Developing your own Real-Time Operating System

For fun, frustration, and definitely not profit. A Project-based course.

Contents

Chapter 1: Introduction

Developing a real-time operating system (RTOS) is a challenge that will make you a better programmer and get you to thoroughly understand the world of embedded software design[1]. In this book and the projects contained within it we are going to take you through the process from blank project to reasonably functional RTOS.

As a point of notation and pronunciation: The acronym RTOS is often pronounced "Arr-Toss", as in you name the letter "R" then pronounce the rest. As such, I will be referring to a singular RTOS as "an" RTOS, since I've never bothered to learn whether it is grammatically correct to use "a" or "an" for an acronym that starts with a consonant whose name starts with a vowel, and neither should you.

What is an RTOS?

Well, it's a real-time operating system, of course! But let's step back for a second. What is an operating system?

An operating system (OS) is a program or set of programs that provide and control access to resources on a computer. On your PC these resources are things like graphics cards, networking, and the hard drive.

The PC's OS's purpose is twofold:

1. Make sure that untrusted programs do not mess up the hardware, for example by erasing the hard drive every time they try to save a file[2]
2. Make sure that the application programmers do not need to know anything about how the hardware works

Purpose 1 has been described as "the difference between computers that crash all the time and computers that crash catastrophically and have to reboot all the time". A modern computer has so many protections in it so that programs written to interact with the user (typically called "userspace" programs) might have to close but will rarely crash the whole computer.

Purpose 2 is known as abstraction. Imagine if you had to write a graphics card driver every single time you wanted to display "Hello, world!" in a new programming language. It would make the task of programming much harder, and it would mean that your code would not be very portable.

The broad organization of an OS

At its core an OS is broken into two parts:

1. The **Kernel**, whose job it is to closely guard access to hardware. The kernel is not optional

[1] For the purposes of this course, "embedded computer" is any programmable computer that is used primarily to run something else and "embedded" in that thing. This contrasts to "personal computer", a category that I choose to allow to include phones and tablets, whose purpose is to run programs that interact with a user. If it bothers you think of it as "an embedded computer is small and doesn't run Windows" and you're mainly good to go.

[2] This is **very** possible if you have the right level of access and is really entertaining when you accidentally do it in a test environment. Not so much if you do it to all of your real files…

2. The **modules**, whose job it is to provide simpler methods to access that hardware (think graphics drivers, user interfaces, windowing systems…). Modules are optional

The word “kernel” is difficult to pin down because it has many different meanings in the world of operating systems development. We are going to work with an unsatisfying definition but one that will suit us for now: a kernel is the core part of any OS, real-time or otherwise. If the kernel is not there, then that specific OS is also not there. Any other code that is not strictly required to run the OS shall be a module.

The kernel is typically as small as possible. For example, while a full Linux install is, as of 2022, many gigabytes in size and includes a massive number of drivers, graphics programs, and other utilities, the Linux kernel itself is only about 5 megabytes. Interestingly, all flavours of Linux use the same kernel, which is known as “The” Linux kernel. As such, we can define a flavour of Linux as “an operating system that uses the Linux kernel”, with absolutely no other requirements. Some flavours have graphics, others use less space, some are networked etc., but all use the same kernel.

So…what is an RTOS?

In an RTOS the kernel is not only small but extremely efficient. Embedded systems have limited resources. It is not uncommon to find a microcontroller that has only a few kilobytes of RAM and operates at only a few tens of megahertz. We can’t expect a kernel that takes millions of cycles to run every second and that uses megabytes of space to be useful.

Writing programs for an embedded computer rather than a desktop or a laptop is similar in many ways and different in others. For one thing, embedded software often allows application programmers to directly access hardware. This is unheard of in the PC world but is standard practice in embedded programming. We might modify Purpose 1 a bit, then, to arrive at the First Purpose of RTOS:

1. To provide safe libraries of code that make it harder for untrusted programs to mess up the hardware

Notice that we are no longer requiring that the kernel protect anything, only that it provides protections should application programs choose to use them. I should mention that it is possible for an RTOS to restrict what untrusted programs can do but this is less often seen in the real world. As such, the separation of userspace and kernelspace is a lot more fluid in an RTOS. This is done both to save resources (it takes a lot of computational effort to run drivers and other layers of code to protect the hardware) and because most of the time an embedded system is supposed to be used for interacting with hardware anyway. If the OS tried to abstract that away it would just bloat things and make it harder to use[3].

The Second Purpose of RTOS is almost the same as that for a regular operating system:

[3] There are exceptions to this rule. The Arduino environment provides exceptional abstraction at the expense of a loss of control – if the Arduino libraries don’t implement something then either you don’t use it, or you have to ditch the Arduino libraries and learn to interact with the hardware directly. Arduino is bloated but its purpose is to encourage new programmers to interact with the microcontroller, not to make high quality real-time applications. For an example of how not to do this see the Atmel Software Framework, which managed to make interacting with hardware so difficult through so many layers of abstraction that I personally will never use Atmel chips again.

2. To provide simple methods of configuring memory and certain hardware on the chip so that applications programmers do not have to know how those specific things work

I'm being a bit cagey here because RTOSs do not usually provide abstractions for all the hardware on a microcontroller. Rather they provide abstractions to the things that are commonly needed in an RTOS. If that bit of circular reasoning makes you mad, here is a brief but non-exhaustive list. At a minimum, an RTOS provides easy ways to access:

- Threading – the ability to concurrently run multiple independent chunks of code, known as threads
- Timing hardware – specifically the use of timers so that threads can run periodically or within known time bounds. Hence the "Real Time" part – RTOSs are intimately linked with timing
- Scheduling resources that determine when each task runs
- Memory protection, especially to ensure that multiple threads do not try to modify the same memory at the same time
- Handling priority access so that important tasks get preferred access to hardware

There is often more that the RTOS does but believe me that this is quite a bit! By the time you finish this course, your RTOS will be able to do all of those things but not much else.

RTOSs also have a kernel, just like an OS for a PC, but its purpose is more limited. Its job is to ensure that threads run when they are supposed to and can access protected memory if they are allowed to. It is otherwise up to the threads themselves to access any other hardware. RTOSs also usually do not have separate application code or other drivers, with the application program being compiled alongside the kernel itself, though this is not universal.

What this course is about

Why are we doing this?

Your RTOS will not be efficient. It will not be able to handle every edge case or truly exotic use cases (for instance, real-time networked control of thousands of drones flying in concert). It will definitely not be secure in any way. It will not be better than one of the many freely available RTOSs out there. So why make one at all?

First, in programming it is often best to learn to do something yourself before you use a pre-made solution. Imagine, for example, someone who wanted to become a chef but had never cooked a meal at home. How would that person be able to function on the job? Similarly, as engineers working with RTOSs we should be intimately familiar with the concepts so that when we settle on a specific RTOS to work with in our careers we aren't baffled by the (often terrible) documentation. Think of it this way: if you are working on a project that requires an RTOS, it's better to be able to ask, "does this RTOS implement mutexes efficiently for my purposes?" rather than "what's a mutex?" Making your own RTOS is one excellent way to gain that knowledge.

Second, creating your own RTOS will lead you to mastery over programming concepts that you may not be clear on even at this stage of your studies. You'll need to understand pointers, memory, computer organization, data structures, and many other concepts just to get it working. This deep understanding will serve you very well in your future career and in interviews for jobs.

Finally, I find that developing software like an RTOS is just a huge amount of fun. When I first learned to program, I was disappointed by how much was done for me. I would type things like `int x = 4;` into a text document and through a process that was basically magic to me the computer ran the program. I found myself wondering what exactly happens between me typing in code and the computer running it. How did the operating system know what to do, and how were those strings turned into instructions? Those questions, while profound, are far too big for this course to answer fully. However, we are going to get you started on the path to answering them.

What are we going to do?

This course is broken into five parts plus one orientation part that will get you set up for success. Each part builds on the previous parts, and we will provide reference implementations at the start of parts two through five that fully and correctly implement the previous parts. They are:

1. Memory mapped IO, pointers, and the stack
2. Context switching
3. Time management and the Round-Robin Scheduler
4. Deadlines, timing, and advanced scheduling
5. Memory protection and inter-thread communication

Memory Mapped IO, Pointers, and the Stack

Before we start writing an RTOS we need to understand certain fundamental things about how the microcontroller works. We will begin with a review of important concepts in C programming, then introduce you to the ARM Cortex M documentation. We will explore how to access the underlying hardware via memory operations, and finally dissect a running C program so that we can understand the key things our RTOS has to manipulate and keep track of.

Context Switching

The whole point of an RTOS is to enable tasks to run within time constraints. Often this also means running concurrent tasks and ensuring that each one gets to run according to its deadlines. Before we can discuss these time constraints, however, we need to understand what tasks are, how they are represented in our OS, and how to switch between them. Executing a successful context switch between two running tasks is a major achievement and is foundational to making an RTOS.

Time Management and the Round-Robin Scheduler

Once we can run multiple tasks, we need to be able to put the “T into our RTOS (or the…RT…?) In this section we are going to learn how to switch tasks based on time, and to force a task to context switch even if it doesn’t yield. We are going to formally implement a scheduler that will serve our purposes and lay the groundwork for more advanced discussions about timing and scheduling later.

Deadlines, Timing, and Advanced Scheduling

The round-robin scheduler is very basic. For one thing, it completely ignores the fact that some tasks need to run within time constraints. In this section we are going to learn how to schedule tasks so that they run within time constraints, and we are going to explore the Earliest Deadline First scheduling algorithm and periodic tasks.

Memory Protection and Inter-Thread Communication

Having multiple tasks running on a single computer gives rise to the problem of data races. What happens if two tasks try to modify the same memory at the same time? Our OS needs a way to protect resources so that this cannot happen. In this part we are going to study the problem through a priority-free lens. Specifically, whichever task asks first gets the resource first, and the OS makes the other tasks wait until it's their turn.

How the lab projects progress

The lab projects in this course are designed to give you a piece of an RTOS each time. By the end of each one you will have code that performs an important subfunction of your RTOS. The projects build in complexity. The first lab project will be like a guided tour of certain functions of the chip, and I will give you quite a bit of code that you can copy directly. The final project will have very little pre-supplied code and will require you to design, build, and test a working system. The reasons for this progress are twofold:

1. In the beginning we are going to do a lot of low-level setup of the chip. This low-level stuff is extremely technical and it's easy to get lost in the details for days. It's not worth our time for you to explore the documentation because this is a course on RTOS for any chip, not just the specific chip we are using
2. Beginning embedded programming is hard and sometimes looks a lot like magic. You can learn from my code so that when it's time to go out on your own you have some examples to go by

This means that you may feel that lab 1 is very cookie-cutter – you will do exactly what I tell you without much independence. I assure you that will change rapidly once we get further in.

Chapter 2: Preliminaries

In this section we will complete a preliminary project that gets you up and running on the microcontroller board, then goes over several important C programming concepts that you may not be familiar with. In this section we are going to create our first microcontroller projects and run some code. It won't be real time but getting this part up and running is important. It's best not to skip this section, but if you are already familiar with the microcontroller and the C concepts you should just skim it and move on.

When you are setting up your projects for the rest of this course, you might want to come back here to remember the exact steps. In this section we will go over:

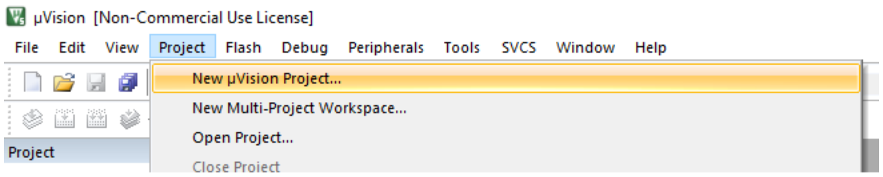
1. Creating a project in Keil uVision 5
2. Retargeting the serial port and using PuTTY to capture serial output
3. Review bit shifting operations in C
4. Review pointers in C
5. Review malloc and free

Preparatory Project 1: Hello, Microcontroller!

When developing software for embedded systems, one IDE is usually capable of programming multiple types of microcontroller. This makes the software very versatile, but we need to make sure that we set it up correctly each time. Any time you are creating a new project in MTE241 using the uVision 5 IDE you must follow these steps to ensure that the software will compile for the correct board. In addition, there are steps we need to take to see any output from the microcontroller. This part may seem daunting at first, but you can always just copy your project folder and rename things so you don't have to do this every time.

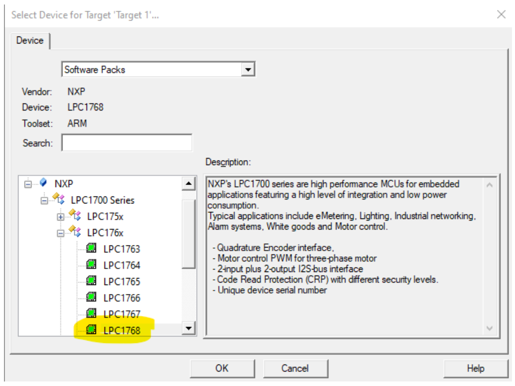
Creating the project and setting the correct board

1. Open uVision 5
2. Click Project->New uVision Project



3. You are now given the option to save your project. We recommend saving it in the "Documents" folder, in some kind of appropriate subfolder. This is a networked folder that will follow you if you use a different computer. Save your project.
4. Once you click save, a new box opens. This is the box we use to select the correct microcontroller. Our microcontroller is an NXP LPC1768:
   a. Expand the "NXP" box
   b. Expand the "LPC176x" box

c. Select the LPC 1768 and click “OK”



Select Device for Target 'Target 1'...

Device

Software Packs

Vendor: NXP
Device: LPC1768
Toolset: ARM
Search:

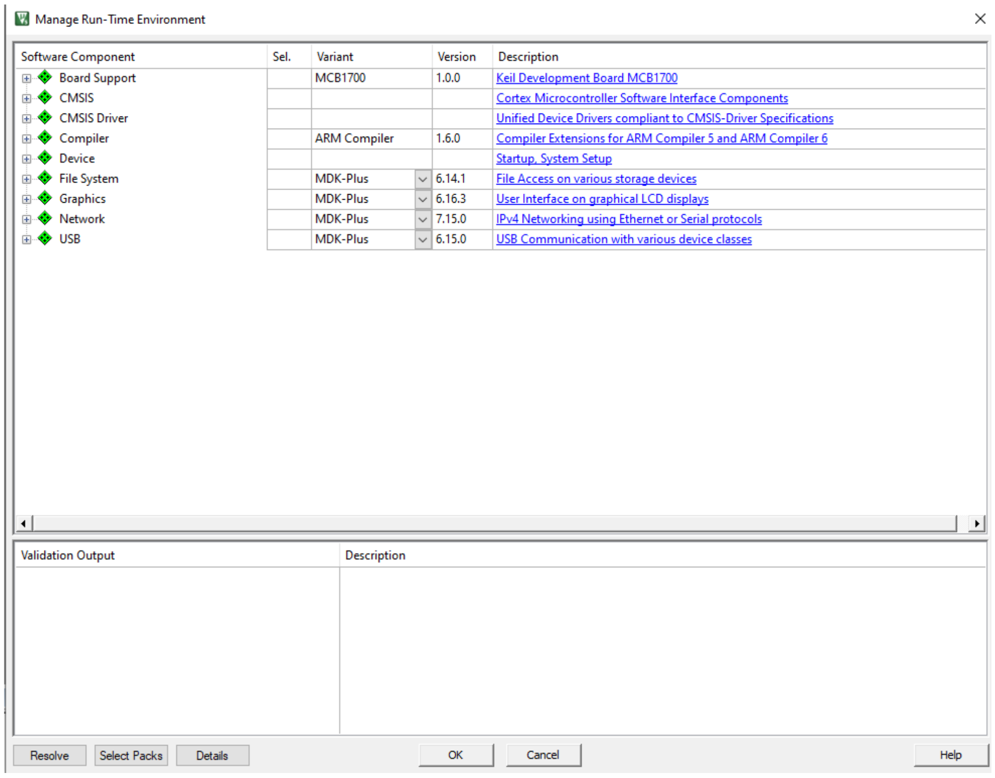
Description:

NXP's LPC1700 series are high performance MCUs for embedded applications featuring a high level of integration and low power consumption.
Typical applications include eMetering, Lighting, Industrial networking, Alarm systems, White goods and Motor control.

- Quadrature Encoder interface,
- Motor control PWM for three-phase motor
- 2-input plus 2-output I2S-bus interface
- Code Read Protection (CRP) with different security levels.
- Unique device serial number

Setting up the Runtime Environment

If you’ve done everything correctly up to this point, you will now see a box that says, “Manage Run-Time Environment”. It looks like this:

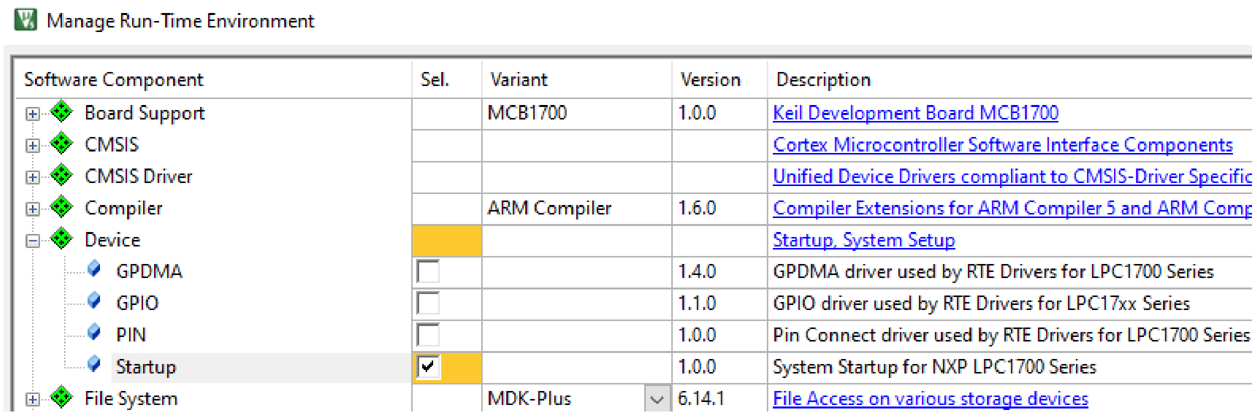


Manage Run-Time Environment

Software Component	Sel.	Variant	Version	Description
Board Support		MCB1700	1.0.0	Keil Development Board MCB1700
CMSIS				Cortex Microcontroller Software Interface Components
CMSIS Driver				Unified Device Drivers compliant to CMSIS-Driver Specifications
Compiler		ARM Compiler	1.6.0	Compiler Extensions for ARM Compiler 5 and ARM Compiler 6
Device				Startup, System Setup
File System		MDK-Plus	6.14.1	File Access on various storage devices
Graphics		MDK-Plus	6.16.3	User Interface on graphical LCD displays
Network		MDK-Plus	7.15.0	IPv4 Networking using Ethernet or Serial protocols
USB		MDK-Plus	6.15.0	USB Communication with various device classes

The Runtime Environment (RTE) is the collection of settings and startup code that we need in the project to make everything work. Unless you’ve had experience working with embedded systems before you’ve probably never had to set up such a thing. On a modern PC the runtime environment stuff is set up for you by the IDE. However, on an embedded system we need to select the correct files and set them appropriately every time, since each project will use vastly different resources. Follow the next few items to set up the RTE so that the LPC1768 is in a minimally initialized state and ready to do simple tasks.

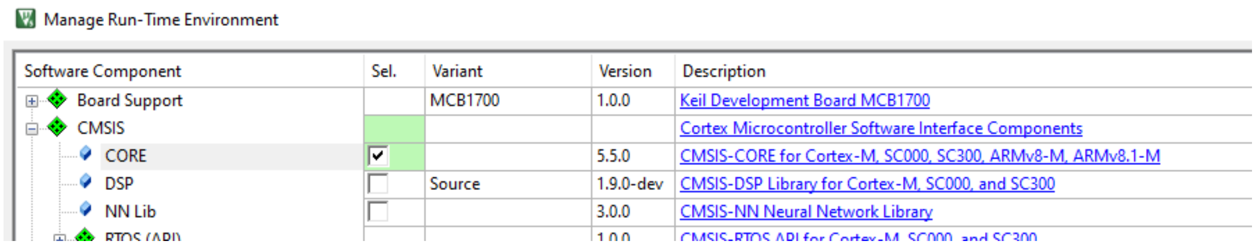
1. Expand the “Device” block and select the box for “System Startup for NXP LPC1700 Series”. This is an initialization file, written in assembly, that sets various flags and other settings in the microcontroller. In general you will rarely, if ever, want to set these things yourself until you are very sure of what you are doing.


Manage Run-Time Environment

Software Component	Sel.	Variant	Version	Description
Board Support		MCB1700	1.0.0	Keil Development Board MCB1700
CMSIS				Cortex Microcontroller Software Interface Components
CMSIS Driver				Unified Device Drivers compliant to CMSIS-Driver Specific
Compiler		ARM Compiler	1.6.0	Compiler Extensions for ARM Compiler 5 and ARM Comp
Device	🟨			Startup, System Setup
GPDMA	☐		1.4.0	GPDMA driver used by RTE Drivers for LPC1700 Series
GPIO	☐		1.1.0	GPIO driver used by RTE Drivers for LPC17xx Series
PIN	☐		1.0.0	Pin Connect driver used by RTE Drivers for LPC1700 Series
Startup	☑		1.0.0	System Startup for NXP LPC1700 Series
File System		MDK-Plus	6.14.1	File Access on various storage devices

Note that the yellow background means that we need to select additional options. We’ll do that next.

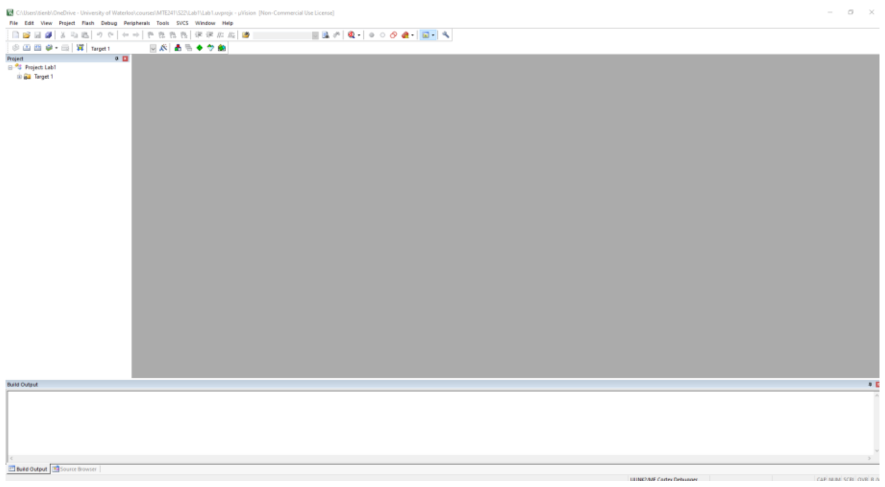
2. Expand the “CMSIS” block and select the box next to “CMSIS-CORE for Cortex-M, SC000, SC300, ARMv8-M, ARMv8.1-M”. You’ll notice that the background colors turn green. This means that we’ve included everything we need for our minimal initialization.


Manage Run-Time Environment

Software Component	Sel.	Variant	Version	Description
Board Support		MCB1700	1.0.0	Keil Development Board MCB1700
CMSIS	🟩			Cortex Microcontroller Software Interface Components
CORE	☑		5.5.0	CMSIS-CORE for Cortex-M, SC000, SC300, ARMv8-M, ARMv8.1-M
DSP	☐	Source	1.9.0-dev	CMSIS-DSP Library for Cortex-M, SC000, and SC300
NN Lib	☐		3.0.0	CMSIS-NN Neural Network Library
RTOS (API)			1.0.0	CMSIS-RTOS API for Cortex-M, SC000, and SC300

3. Click “OK”

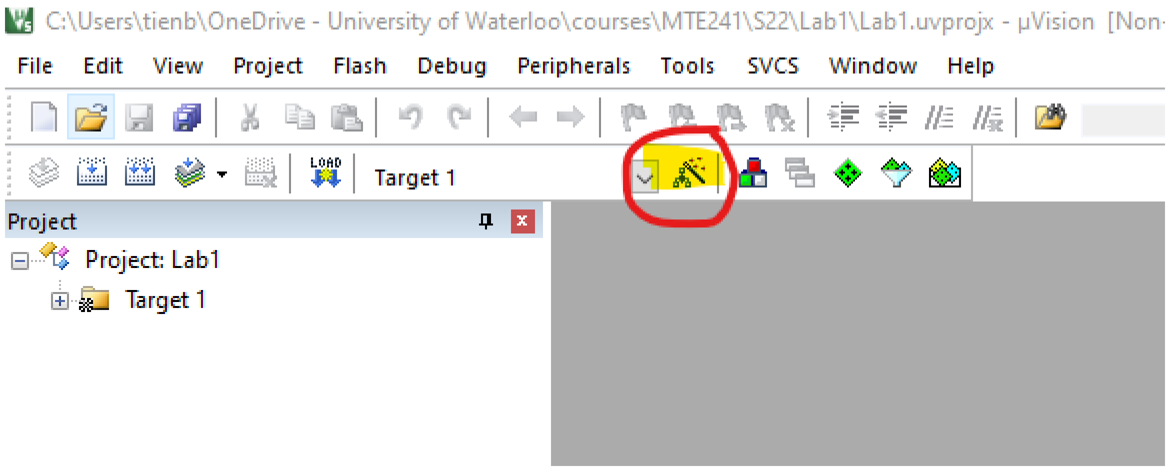
At this point the dialog boxes close and you are left with the IDE window looking something like this:



Setting up the compiler

We're not quite done yet! We need to make sure that the compiler is using the correct settings.

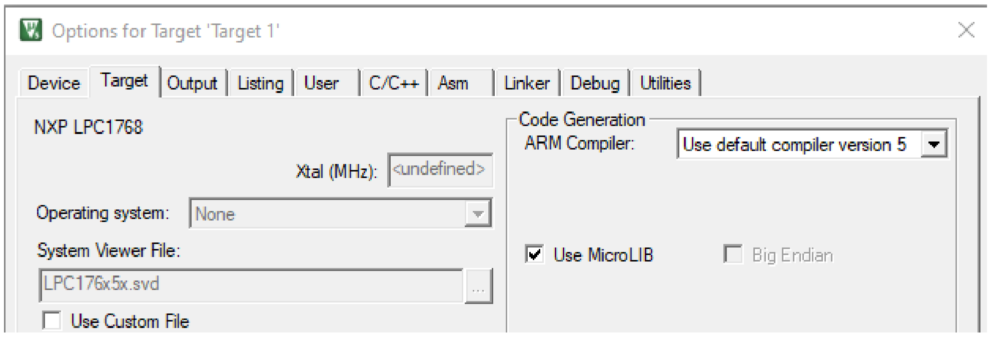
Click the button that looks like a magic wand[4]:


C:\Users\tienb\OneDrive - University of Waterloo\courses\MTE241\S22\Lab1\Lab1.uvprojx - µVision [Non-
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Target 1
Project
Project: Lab1
Target 1

A whole new dialog box opens that lets us set up the compiler.

1. Under the "Target" tab:
    a. Change "ARM Compiler" to "Use default compiler version 5". Some of the code written by ARM has not been optimized to use new language features and if we don't do this we'll see a ton of warnings for no real reason.
    b. Click the box next to "Use MicroLIB". This is a code library that is optimized for our device and which will give us access to some very useful functions later on.

    If you did everything right so far, it will look like this:


Options for Target 'Target 1'
Device Target Output Listing User C/C++ Asm Linker Debug Utilities

NXP LPC1768

Xtal (MHz): <undefined>

Operating system: None

System Viewer File: LPC176x5x.svd

Use Custom File

Code Generation

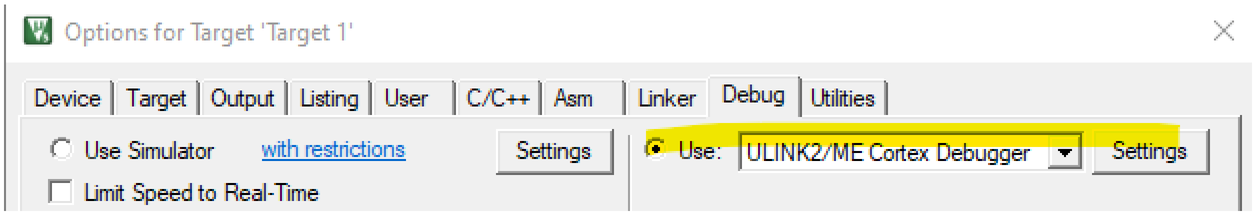
ARM Compiler: Use default compiler version 5

[✓] Use MicroLIB

[ ] Big Endian

2. Under the "Debug" tab, check to make sure that the debugger is set to "Use: ULINK2/ME Cortex Debugger" and not the simulator or anything else. You probably don't need to change this setting, but sometimes it gets reset. If you don't use this debugger you cannot communicate with the microcontroller to debug!

[4] I'm not going to lie here, the magic wand imagery is a throwback to the early 1990s when setting up a computer was so difficult that only a "wizard" could do it. Software companies caught on and designed various "wizard" programs to do the setup for you, and the terminology and symbols stuck. So yes, we are about to perform magic.


Options for Target 'Target 1'

Device | Target | Output | Listing | User | C/C++ | Asm | Linker | Debug | Utilities

( ) Use Simulator with restrictions Settings

[ ] Limit Speed to Real-Time

(●) Use: ULINK2/ME Cortex Debugger Settings

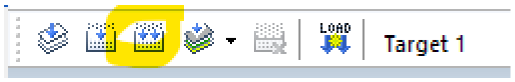
Populating the project with files

Once you have the new project ready to go, you need to add the rest of the source files.

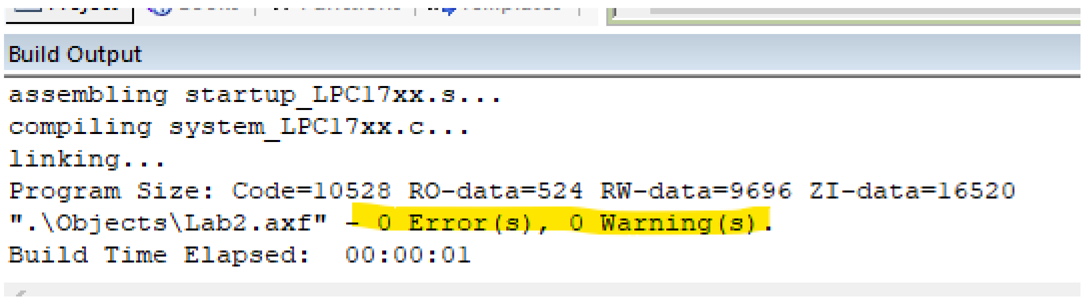
1. Unzip the files in the "lab0_starter_code.zip" file into the same folder where you saved your project – where the project is saved and where the code is saved matters. If they are not in the same folder you will need to go through extra steps, so make sure that the actual code files, the *.c and *.h files, are in the same folder as your project file
2. In the project window of uVision 5, expand the "Project" folder, then right click on "Source Group 1". In the menu that comes up, select "Add Existing Files to Group 'Source Group 1'…
3. Navigate to where you saved the files and add all of the *.c code files. Note that there are also *.h files, but they are included in the project automatically as long as they are in the same folder. Frustratingly the dialog box doesn't confirm that the files are added, but rest assured they are. Close the dialog box and expand "Source Group 1" to see them

Compiling and Downloading Code

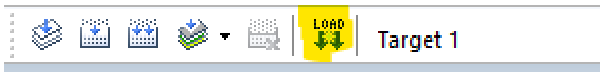
Programming for a PC is a matter of compiling and running the code. On the microcontroller there is an extra step. We compile the code on the PC, then we need to load it onto the microcontroller, and then we can run it.

1. Click the "rebuild all" button in uVision: 

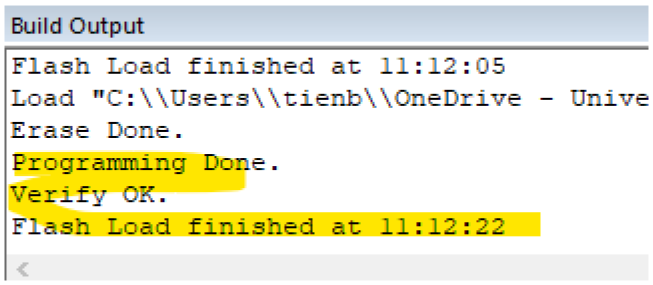
2. Assuming everything went well, you should see the compiler dialog box indicating zero errors and zero warnings, like this (your output won't look exactly the same, but the goal is 0 errors and 0 warnings):


Build Output

```
assembling startup_LPC17xx.s...
compiling system_LPC17xx.c...
linking...
Program Size: Code=10528 RO-data=524 RW-data=9696 ZI-data=16520
".\Objects\Lab2.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:01
```

3. Click the "Load" button: 

4. If all went well, you should see information indicating that the programming is done and the code is verified:



```
Flash Load finished at 11:12:05
Load "C:\\Users\\tienb\\OneDrive - Unive
Erase Done.
Programming Done.
Verify OK.
Flash Load finished at 11:12:22
```

5. Press the reset button on the microcontroller, which will start the code. However…you're not going to see much yet!

We aren't done. The whole point of this introductory project is to print to the Serial port. But at the moment we don't have any programs that are monitoring the serial port. Let's fix that.

Serial Communications

One of the most common, though least sophisticated, ways to debug a program is to print its output. On the PC this is as simple as printing to the console but on a microcontroller there is no console to print to! The Keil board we are using has an LCD screen that can be used to print debug information and display other things, but that display is small and limited so it is often easier to connect to the board using a serial port and software on the PC.

A serial port on a microcontroller is a specific piece of hardware that can send data one bit at a time, hence the name: the data is sent serially, one bit at a time, until it is done. Modern computers don't have the same kind of hardware[5] but we can still communicate over serial with a microcontroller if we have the appropriate interface boards.

The serial hardware on the microcontroller is called the Universal Synchronous/Asynchronous Receiver/Transmitter, or USART (often you'll read about a UART, which drops the "synchronous" part. I'll refer to the hardware as UART in this manual as well since that's what the ARM software we are using calls it). In this manual we are going to learn how to set up a serial connection.

Setting the IDE to recognize UART

This part is probably the least intuitive, so if you aren't sure what's going on be sure to read carefully.

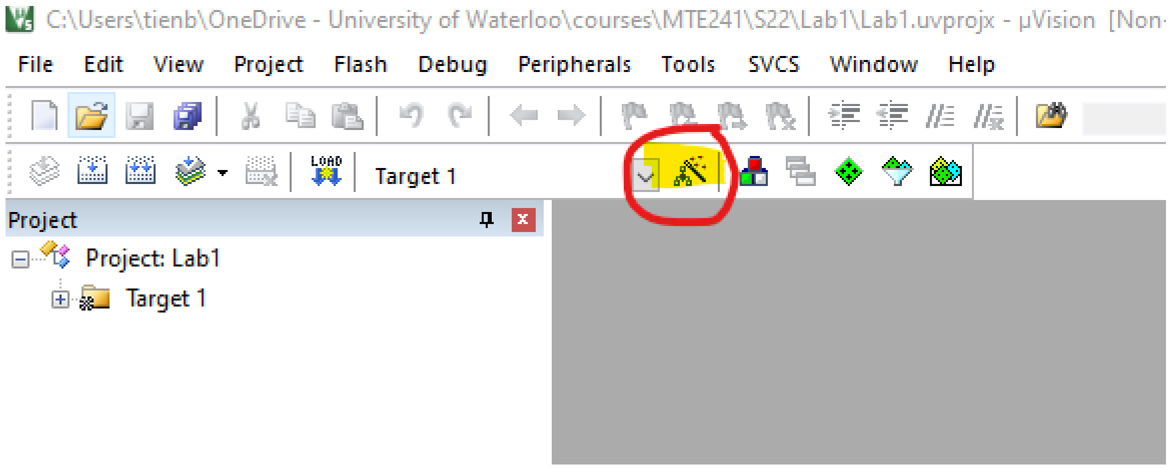
In the C programming language, we use a library called standard io (`stdio.h`), which has many functions for printing and obtaining input. This printing is usually done on a PC to a console or terminal window, which we do not have with the Keil board. However, `stdio.h` is still available for the Keil board we are using. To make it work we need to tell uVision to retarget stdio's output to the serial port.

The actual steps to do this are quite complicated, so we've done a lot of it for you. If a project uses retargeted UART we will provide the appropriate *.c files. However, you need to define an important symbol that this code uses in your project. We cannot do this for you.

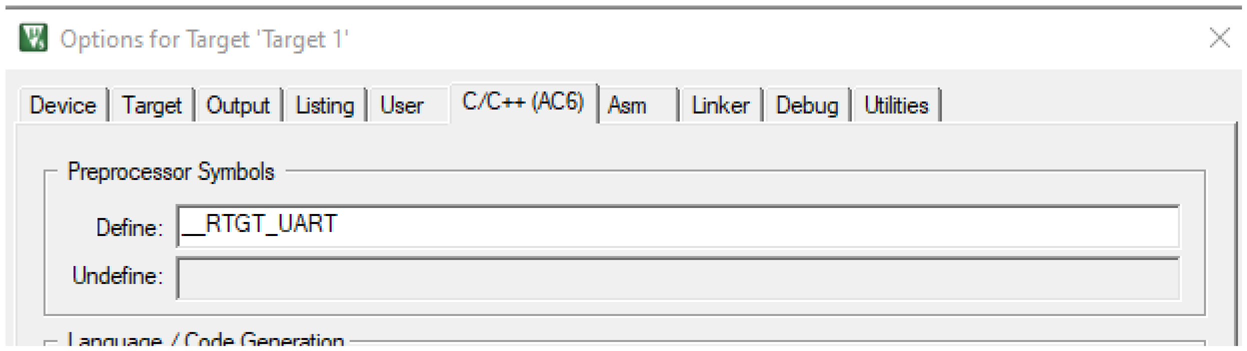
1. Ensure that you have set up your project according to the instructions above

[5] The now ubiquitous USB (Universal Serial Bus) is far more complicated than the serial hardware used on a microcontroller.

2. Open the options for target wizard


C:\Users\tienb\OneDrive - University of Waterloo\courses\MTE241\S22\Lab1\Lab1.uvprojx - µVision [Non
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Target 1
Project
Project: Lab1
Target 1

3. In the C/C++ tab, you will see a box that says “Define”. Type __RTGT_UART in that box. Note that this symbol begins with two underscores, so it should be “underscore underscore RTGT underscore UART”



Define: __RTGT_UART

Undefine:

4. Click OK

This tells our retargeting code that we are retargeting (RTGT, as in “ReTarGeT”) to the UART.

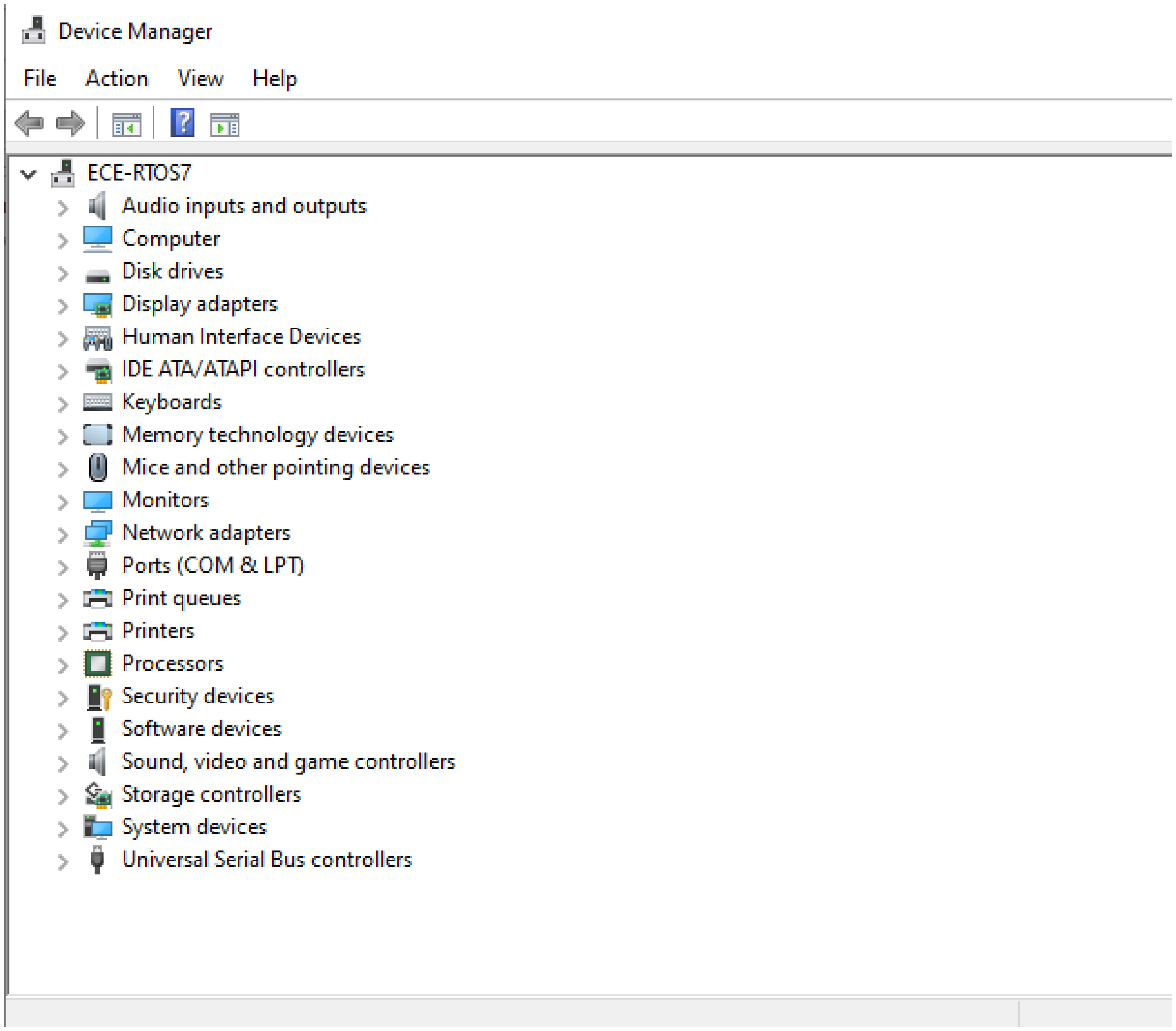
At this point the compiler is ready to start sending serial data. If you haven’t already make sure that your microcontroller has been loaded with a program that prints to the serial port. Now, the PC isn’t going to accept any serial data until we open the appropriate program and set it properly.

Using PuTTY to Communicate via Serial

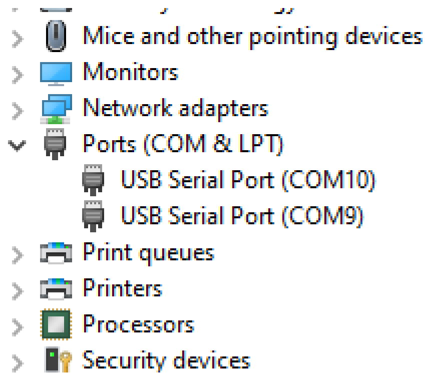
Serial communication on the PC happens via COM ports. COM ports are legacy (i.e.: old) but useful things. COM ports are numbered and assigned to devices when the PC boots, and unfortunately it is not possible to know which port your device gets. As such, we need to find them using the device manager.

Part 1: Finding the COM ports

1. Search for the “Device Manager” on the PC. You may see a popup telling you that you are unable to make any changes if you are not an administrator on the machine you’re using. That’s fine, we just need to know which COM ports we have access so, so click “OK” when it pops up. You should see something like:



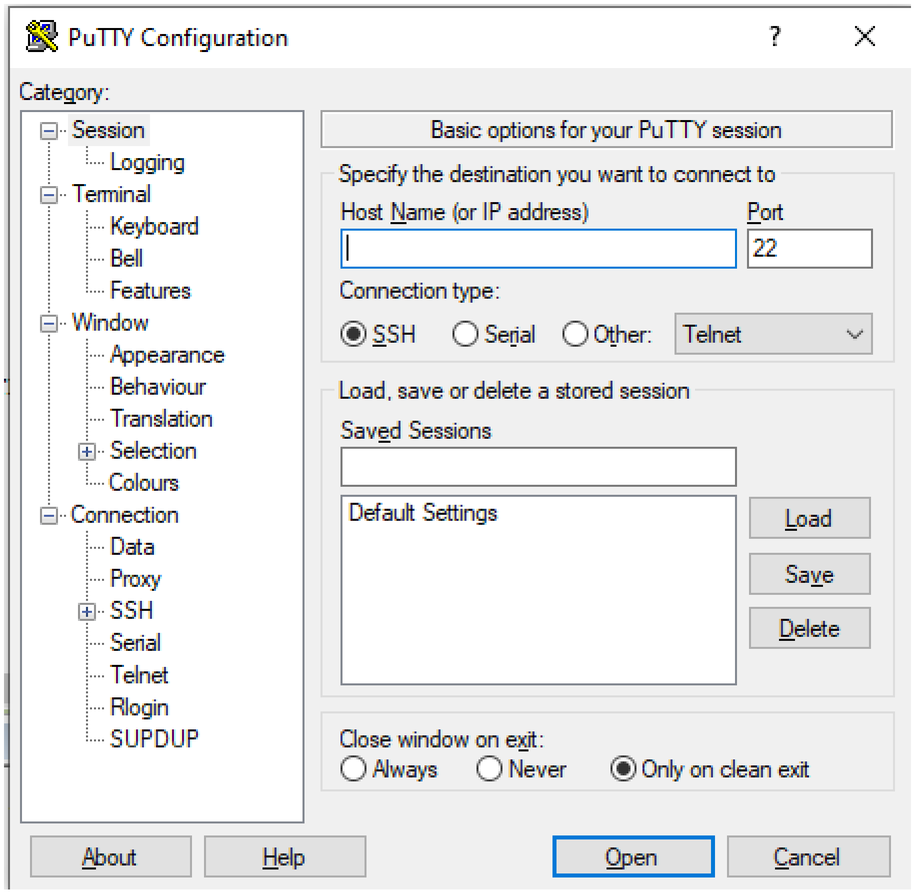
2. Click on "Ports (COM & LPT)" and it will expand. In the lab, you will likely see two COM ports. In the example below I have COM10 and COM9 assigned, but you may see something different:



3. Write down the COM port numbers you have. One of them is the microcontroller, but unfortunately we do not know which.

Part 2: Communicating via Serial

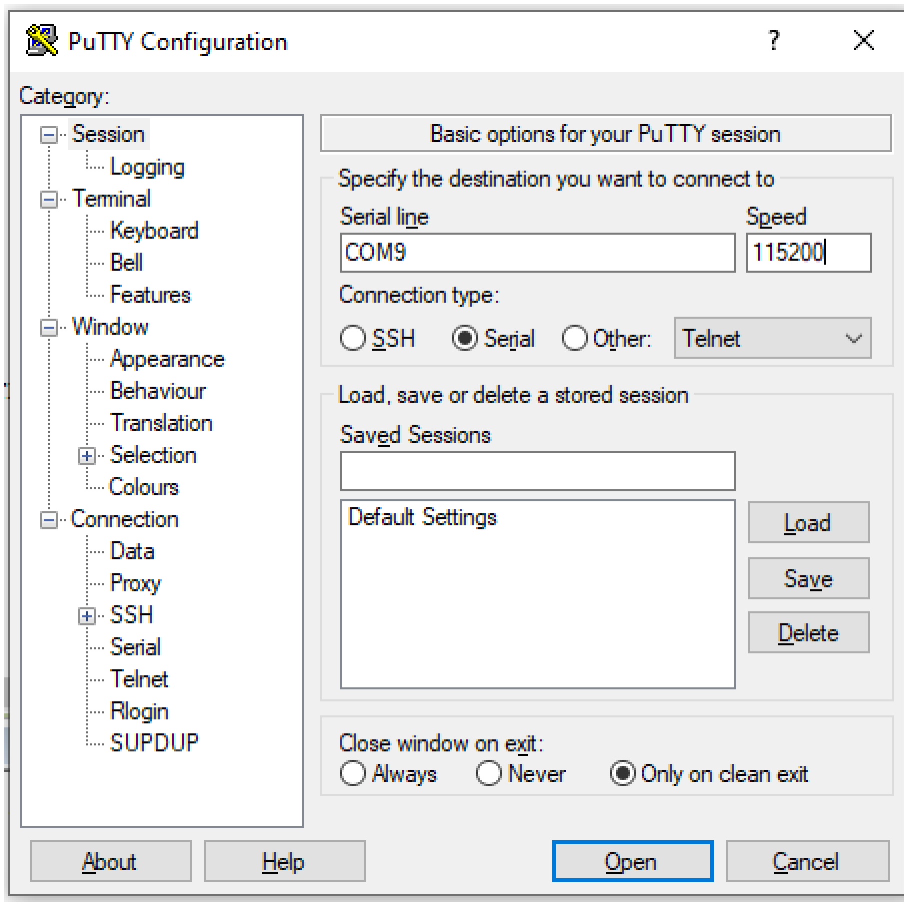
The program we are going to use is called PuTTY[6], which is a free and very useful program for this sort of thing. Open it by searching for it in the start menu. You will see:



Specify the destination you want to connect to
Host Name (or IP address):
Port: 22
Connection type: [✓ SSH] Serial Other: Telnet
Load, save or delete a stored session
Saved Sessions
Default Settings
Close window on exit: Always Never [✓ Only on clean exit]

1. Under “Connection type”, click “Serial”
2. In the “Serial line” box, enter the appropriate COM port. For example, if you are connecting to COM9, you write the string COM9. Note that, since you likely have two COM ports for the Keil board, if one of them doesn’t work try the other one before asking for help.
3. Under “Speed” enter 115200 – this setting is extremely important to get right but you don’t need to know what it does now. If you are interested, ask in the lab! At this point your window should look like this:

[6] You might think that PuTTY’s name has some meaning, but the developers have explicitly stated that the name is meaningless.



Specify the destination you want to connect to

Serial line: COM9

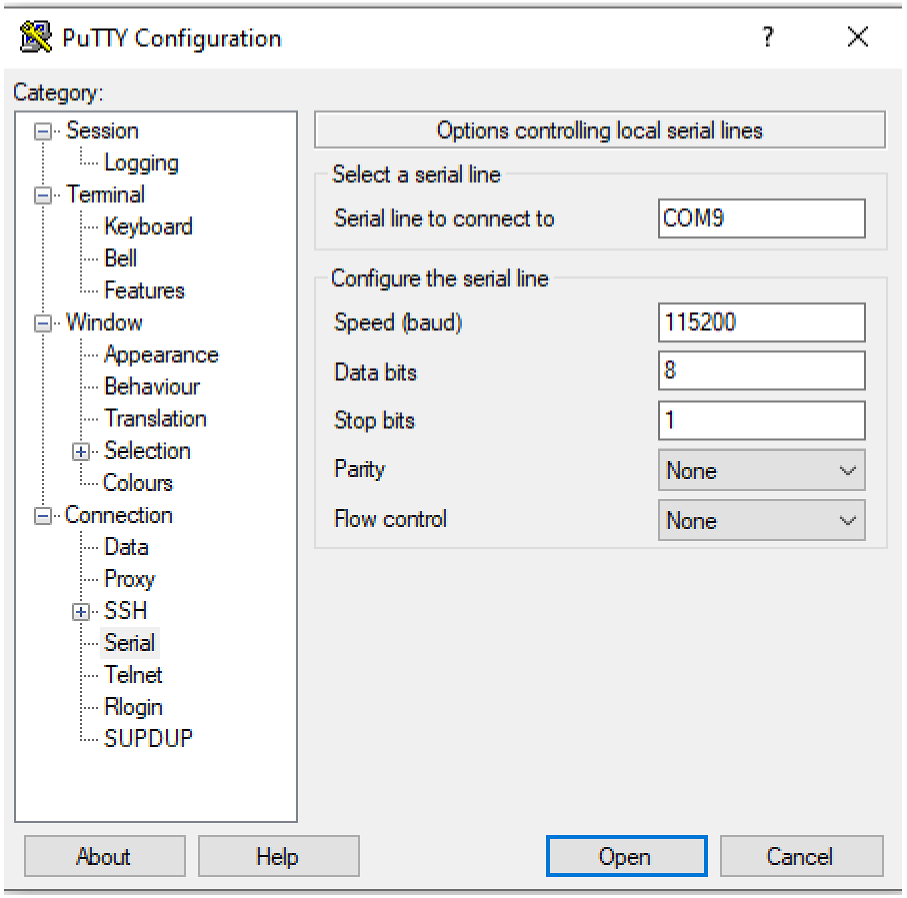
Speed: 115200

Connection type: ○ SSH [● Serial] ○ Other: Telnet

Saved Sessions: Default Settings

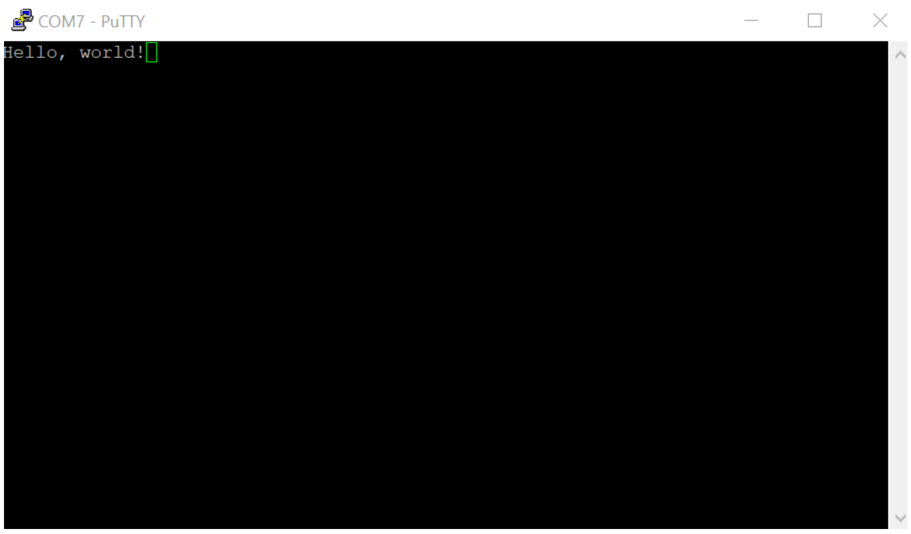
Close window on exit: ○ Always ○ Never [● Only on clean exit]

4. On the left side menu, click "Serial"
5. Under "Flow control", select "None", but leave everything else unchanged.
6. **Optional:** If you intend to disconnect and reconnect a lot, you may want to save the session. Go back to the "Session" tab (click "Session" on the left side menu), enter a name under "Saved Sessions", then click save. If you want to load it again, click the name and click load.
7. If your PuTTY window looks like what is shown below, you may now click "Open" to open the serial connection



Serial line to connect to: COM9
Speed (baud): 115200
Data bits: 8
Stop bits: 1
Parity: None
Flow control: None

8. Reset your microcontroller now with the PuTTY session open. If you see something like what is shown below, congratulations! You've successfully completed the project:



```
Hello, world!
```

Preparatory Project 2: Low-level hardware interfacing

In this project we are going to learn how to interact with the low-level hardware of the microcontroller. You may choose to write this in the main file from Preparatory Project 1's starter code, or you may choose to practice setting up a new project on your own.

Background

At a bare minimum a microcontroller needs a CPU, RAM, and permanent storage such as flash memory to be useful. However, most microcontrollers have many more subcircuits that are useful in expanding their capabilities. These subcircuits are often, but not always, directly on the chip itself and have many purposes. We call these subcircuits "peripherals".

The Keil MCB 1700 board provides several peripherals that you can make use of. You are already familiar with the use of the UART (i.e. serial port) from Project 1. The additional peripherals you will be using in this project are:

- LEDs
- Joystick
- Push button

Interestingly, all these peripherals are external devices that are on the Keil board, of which the microcontroller is only a small part. To interface with them we need to use the microcontroller's General Purpose Input/Output (GPIO) functionality. Setting up and using GPIO on a microcontroller is more complicated than simply printing to a screen. In this project a major focus will be on understanding how to do this.

Multifunction, Memory Mapped IO on the LPC1768

The LPC1768 is very flexible and it's possible to configure many of its pins to perform multiple functions[7]. Since there are only a limited number of pins, some pins must be used for multiple things. If this were not the case, a chip might need hundreds of pins just to support every function it has!

For us to program the pins and set their modes, modern CPUs used memory mapping. What this means is that certain special, reserved pointers are used that can be read and/or written to. These memory locations are set by the chip manufacturer and can never be changed. By writing the correct value to the correct pointer we tell the chip to set up a specific pin to have a specific function. The CPU handles the actual configuration of the circuits after the pointers are written to. These pointers are directly accessible if you know their address, and we'll see how to look up that information in the next chapter. For now, we'll rely on the automatic setup code generated by Keil uVision.

What this means practically is that we are going to write a lot of code that looks like magic. For example,

[7] This isn't unique to the 1768. Many, many chips do this. In fact, one of the distinguishing factors between cheaper and more expensive chips is how many functions you get.

```
LPC_GPIO2->FIODIR |= 0x0000007C;
```

We'll learn what all of this means as we progress. In this specific case, LPC_GPIO2 is a pointer to a struct that contains a number of memory mapped register locations. FIODIR is one of those, and the bitwise operation sets specific bits in that register to 1.

Note that it does take quite some time to learn the functions and names for each of these registers for a particular chip. Usually an engineer would have a copy of the chip's datasheet and programming guides open as they work. It's almost never useful to memorize these things since there are so many and they are so specific to each chip.

In this project we are going to focus specifically on the Pin Select and Data Direction registers of the LPC1768, along with a few others when we get to them.

Pins, Ports, and Registers

The memory bus on an ARM Cortex M chip is 32 bits wide. This means that it is easier to address an entire 32 bit register than a single bit. To simplify the process of setting up what various pins do, the pins are grouped into "ports", most of which are 32 bits wide. The processor can only read or write the full port at once, so to change the value of individual pins you'll have to mask the bits using binary operations. Below we'll go through what these bit operations are.

To further complicate things, the bits are set by writing to one register, and cleared by writing to a different register. This is an architectural choice by ARM, and some chips do not do this. It's just one of those things you will have to learn to deal with. You'll find that as you become more familiar with different chips made by different manufacturers, each one has unique quirks that others don't have.

All of the ports (both the data ports and the various configuration registers) are memory-mapped. The file lpc17xx.h contains definitions for all of these addresses, and should be #included in your source code as follows:

```
#include <lpc17xx.h>
```

The GPIO registers are represented as a pointer to a struct containing a number of fields corresponding to the various registers. For example, we'll be using LPC_GPIO1 and LPC_GPIO2. Within each of those structs are fields named FIODIR (to specify the input/output directions), FIOSET (to set bits on the port to one), FIOCLR (to clear bits on the port to zero) and FIOPIN (to read the current value of the pins)

Knowing the name of the memory locations, or even the address to find them, is just part of the challenge. We also need to know how to manipulate the memory that is there. We do this using bitwise operators, which we will review next.

Bitwise operations: A review

Most CPUs operate with a particular memory bus size, measured in bits. Common values are 8, 16, 32, and 64 bits. The ARM Cortex M processor has a 32-bit wide memory bus. To avoid confusion, we are going to assume that we are working only on 32-bit computers. The contents of this document are still correct for any memory bus size, however.

The bus size represents the smallest unit of memory that can be addressed, loaded, or stored by the processor. Anything smaller than that requires us to do additional operations on the memory once a whole 32 bits has been loaded.

Sometimes the compiler handles this for us. For example, if we create a character array:

```
char ar[3] = {‘a’,’b’,’c’};
```

the compiler will take care of placing these three characters at convenient locations in memory (usually what this means is up to the compiler and is very much outside the scope of this document!)

Modern processors allow us to individually change specific bits in memory by first loading an entire 32-bit memory location, which we often store in an unsigned integer. In C, the operands used to do this “bitwise” modification of a memory location are called the “bitwise operators”.

The Bitwise Operators

We are going to look at only a few of the bitwise operators: bitwise not, and, or, and shift.

Bitwise NOT

Purpose: negate all bits in the memory location. This is frequently used in bit masking (see below)

Syntax: the tilde symbol placed in front of the variable: `y = ~x;`

Example:

```
unsigned int x = 4; //this variable now stores the binary
//0b00000000000000000000000000000100


unsigned int y = ~x; //This variable now stores the binary
//0b11111111111111111111111111111011
```

Bitwise AND

Purpose: take two 32-bit memory locations and perform the logical AND operation on each corresponding pair of bits.

Syntax: a single ampersand placed between two variables: `z = x & y;`

Example:

```
unsigned int x = 5; //this variable now stores the binary
0b00000000000000000000000000000101

unsigned int y = 4; //This variable now stores the binary
0b00000000000000000000000000000100

unsigned int z = x&y; //This variable now stores the binary
0b00000000000000000000000000000100
```

Bitwise OR

Purpose: take two 32-bit memory locations and perform the logical OR operation on each corresponding pair of bits.

Syntax: a single pipe placed between two variables: `z = x | y;`

Example:

```
unsigned int x = 5; //this variable now stores the binary
0b00000000000000000000000000000101

unsigned int y = 4; //This variable now stores the binary
0b00000000000000000000000000000100

unsigned int z = x|y; //This variable now stores the binary
0b00000000000000000000000000000101
```

Right Shift

Purpose: Shift all bits of a 32-bit memory location right (towards least-significant bit) by the specified number of bits, filling in the remaining bits with zero and truncating anything that gets shifted below the LSB. Note: The LSB is bit 0, so a shift of 0 will do nothing.

Syntax: Two chevrons placed between the variable and the bit shift, pointing right: `y = x>>2;`

Example :

```
unsigned int x = 5; //this variable now stores the binary
0b00000000000000000000000000000101

unsigned int y = x >> 2; //this variable now stores the binary
0b00000000000000000000000000000001
```

Left Shift

Purpose: Shift all bits of a 32-bit memory location Left (towards most-significant bit) by the specified number of bits, filling in the remaining bits with zero and truncating anything that gets shifted above the MSB. Note that on the LPC1768 a shift of more than 32 bits will result in a value of 0. This is not true on some other processors.

Syntax: Two chevrons placed between the variable and the bit shift, pointing left: `y = x<<2;`

Example :

```
unsigned int x = 5; //this variable now stores the binary
0b00000000000000000000000000000101

unsigned int y = x << 2; //this variable now stores the binary
0b00000000000000000000000000010100
```

Useful Examples

In this example, assume that we are trying to access and modify a variable, `MEM_LOC`, that contains useful settings. For example, perhaps if bit 16 is set to 1 an LED will turn on, and if it is set to 0 an LED will turn off. This is how the memory mapped IO works in the Arm Cortex M.

Example 1: Setting a bit

Note: the term "set" means "set to 1". It never means "set to 0" unless that is specifically stated. Presume we want to set a specific bit to 1. For the sake of the example, let us set bit 16. To do this we use bitwise OR with a left bit shift:

```
MEM_LOC |= (1 << 16);
```

Note the |= syntax. This line takes the current value of MEM_LOC and bitwise ORs it with 1, shifted by 16 bits to the left. The result of that bit shift will be a 32-bit number whose 16th bit is 1 but every other bit is zero.

If you think about why this works, consider that 0|1 = 1 and 1|1 = 1. So whatever is at bit 16 will be 1 no matter what, and everything else will remain unchanged.

Example 2: Clearing a bit

Note: the term "clear" means "set to 0". It never means "clear to 1", and that is never said.

Let us now say that we want to clear a specific bit. As above, we are going to clear bit 16. This is a bit[8] tricky – we can't bit shift a 0. The way to do this is to bit shift a 1 to the location we want, then bitwise NOT the bit-shifted result, and finally bitwise AND the result with our memory location. It looks like this:

```
MEM_LOC &= ~(1 << 16);
```

Wow. Why? Let's unpack this.

Starting inside the brackets, we have:

```
(1<<16) = 0b0000000000000010000000000000000
```

Now we negate it:

```
~(1<<16) = 0b1111111111111101111111111111111
```

Finally, when we perform the bitwise AND, we note that 0&1 = 0, so whatever is at bit 16 will be set to 0 for sure. The remaining bits will remain at whatever they were – if the bit was 1 originally, then 1&1=1, and if the bit was 0 originally then 0&1 = 0, so the rest of the bits remain unchanged.

Example 3: A fancy way to set multiple bits

Let's now say that we want to set bits 2-6. We could do this using 5 separate bitwise OR operations as we did in Example 1, but that is inefficient. Consider that what we want is to bitwise OR MEM_LOC with a group of 1s in position 2-6. This would be binary 0b1111100 (remember, bit 0 is the LSB, so bit 2 is the third bit from LSB). Some compilers (including the one we use in the lab) do not like binary in the 0bxxxx format. Instead we turn this into a hexadecimal number. Either because you are very good at this or you used an online calculator, the binary 0b1111100 is the hex 0xFC.

We could therefore do this by bitwise OR using that binary constant, like this:

```
MEM_LOC |= 0xFC; //You do need the "0x" there to indicate hex
```

Often this method is used along with a defined macro[9] using the `#define` preprocessor directive in a useful header file. So we might `#define TURN_ON_LEDS 0x7C;` and then the line would look like this:

```
MEM_LOC |= TURN_ON_LEDS;
```

[8] Ha!
[9] C is not C++. Macros are used with #defines rather than constants for this purpose.

This is an extremely useful way to keep track of various settings, because once defined you don't have to remember the numerical sequence, just the name of the macro.

Exercise 4: Reading a bit via bit masking

Bit masking is a fancy way to say "set everything except the bits we care about to zero". Let us say that we want to read the value stored in the 16th bit of MEM_LOC. Unless we are very sure what we are doing it's usually best to store the variable in something temporary first:

```
unsigned int tmp = MEM_LOC;
```

Then, we bitwise AND this new temporary variable with an unsigned integer that is zero everywhere except the 16th bit:

```
tmp &= (1<<16);
```

This has the effect of setting every bit in tmp to 0 and leaving the 16th bit as it is. Finally, we can just check it using an if statement. Remember, in C, everything that is nonzero is considered to be true and zero is considered to be false, so we can determine if that bit is set by doing:

```
if(tmp) //if true, bit 16 is 1
```

Example 5: Bit shifting 1 by 31

This might sound like a weird example, but it isn't. On a 32-bit machine, the most we can possibly bit shift is by 31. If we try to shift by 32, say, the compiler will warn us and the results won't be what we expect when we run it. But there is a problem. The number 1 is interpreted by the compiler as a signed integer. Signed integers are special in that the MSB is the sign bit, so if we do `(1<<31)` the compiler will throw a warning. To get around this it is best to cast 1 as an unsigned integer, like this:

```
((unsigned int)1<<31)
```

Note that when there is no ambiguity, you can force the constant 1 to be unsigned by using the "U" postfix, like this:

```
1U<<31
```

Both work. The explicit cast makes for more readable code, however, and is harder to miss when doing a code review.

Project 2: Exercises

Exercise 1: Setting the LEDs

We'll begin by working with the LEDs, which will give you a good idea of how all of this works. Note that, in real life, you will probably be reading a datasheet to understand exactly what is connected to what. We are going to simplify that process today by just telling you where everything is connected. However, you should know that it took lab instructors some time to read the datasheets and figure this out. A very common confusion among beginning embedded systems engineers is to assume that "everyone just knows this and I don't". That's not true. Sometimes learning which port does what can take days!

There are eight LEDs on the board and they are connected to specific GPIO pins on ports 1 and 2. In order to turn the LEDs on or off we need to do two things:

1. Set the pins' direction to 1, which tells the chip to treat these pins as output pins

2. Set or clear the pins when we want to turn the LEDs on or off

The first three LEDs are on GPIO port 1 pins 28, 29, and 31 (not 30, that would be too logical). The remaining LEDs are on GPIO port 2 pins 2, 3, 4, 5, and 6.

The GPIO ports are accessed via pointers to two structs, `LPC_GPIO1` for port 1 and `LPC_GPIO2` for port 2. These structs contain 32-bit unsigned integers `FIODIR` to set the direction, `FIOSET` to set the bits, and `FIOCLR` to clear the bits.

Setup step

In your main, set the LED pins' directions. For instance, to ready the first LED, you would write:

```
LPC_GPIO1->FIODIR |= 1<<28;
```

In main, set all of your LEDs' directions to 1. Remember to include a while(1) loop at the end of your main.

Exercise

Write a function that receives a single unsigned integer. This function should turn on the LEDs corresponding to the lower 8 bits of this number. For example, if you pass in the integer 5, which corresponds to 0b00000101, LEDs 0 and 2 should turn on with the rest off. Thoroughly test your code.

Exercise 2: Reading the Joystick and Pushbutton

Understanding

The joystick is actually a four-position switch, with each position corresponding to a particular input pin. In addition, you can press in on the joystick to give you a one-bit input that is also hooked up to a specific input pin.

This time we can avoid using the Data Direction registers, since they default to input.
The input pins in question correspond to bits 23 through 26 of GPIO 1 for the four directions, and bit 20 of GPIO 1 for pressing in on the joystick. The pushbutton is read similar to the joystick but is on pin 10 of GPIO2. To read an IO pin we use the `FIOPIN` register using a bit mask (see the Bitwise operations in C document, exercise 4, to understand how to do this).

Note that you will need to determine what values are read on the pins when the joystick or button is pressed or not.

Exercise

Write a function that continuously checks the joystick and prints UP, LEFT, RIGHT, DOWN, JOYSTICK_PUSHED or PUSHBUTTON_PUSHED via UART as appropriate. The function may continuously print while the joystick or button is being pressed in a particular way.

Preparatory Project 3: Pointers, Malloc, Free, and printf

In this final part we are going to tour some of the more confusing parts of the C programming language. Feel free to set up a new project for practice, or keep the old ones going from the previous Preparatory Projects.

Pointers

Never in the history of programming has a topic caused more confusion than pointers[10]. At this stage in your programming career, you don't need to me to make another analogy like "pointers are like the index in the back of a book" because:

1. You've likely already heard most of the analogies and are still confused
2. Present document excluded, who learns how to program from a book anymore?

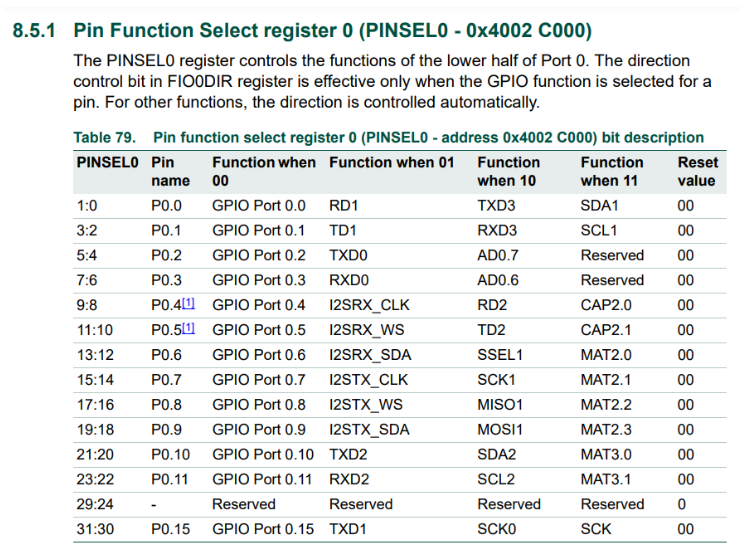
So let's just jump right in and assume that you have seen code like this before:

```
int* y = &x;
```

The variable y is a pointer to an integer, x (declaration not shown), whose address is accessed via the ampersand, &x.

When interacting with a microcontroller we are going to use pointers a lot, since much of the microcontroller's hardware is accessed via pointers – we locate the correct pointer location in the documentation, then access the memory there to change settings for the hardware. In the microcontroller's documentation these are often given as unsigned integers in hexadecimal format. For example, in the documentation for the LPC1768 ARM Cortex M3 microcontroller, the following table describes a register known as "Pin Function Select 0":

[10] That statement is entirely false, but is included for dramatic, hyperbolic effect.



8.5.1 Pin Function Select register 0 (PINSEL0 - 0x4002 C000)

The PINSEL0 register controls the functions of the lower half of Port 0. The direction control bit in FIO0DIR register is effective only when the GPIO function is selected for a pin. For other functions, the direction is controlled automatically.

Table 79. Pin function select register 0 (PINSEL0 - address 0x4002 C000) bit description

PINSEL0	Pin name	Function when 00	Function when 01	Function when 10	Function when 11	Reset value
1:0	P0.0	GPIO Port 0.0	RD1	TXD3	SDA1	00
3:2	P0.1	GPIO Port 0.1	TD1	RXD3	SCL1	00
5:4	P0.2	GPIO Port 0.2	TXD0	AD0.7	Reserved	00
7:6	P0.3	GPIO Port 0.3	RXD0	AD0.6	Reserved	00
9:8	P0.4[1]	GPIO Port 0.4	I2SRX_CLK	RD2	CAP2.0	00
11:10	P0.5[1]	GPIO Port 0.5	I2SRX_WS	TD2	CAP2.1	00
13:12	P0.6	GPIO Port 0.6	I2SRX_SDA	SSEL1	MAT2.0	00
15:14	P0.7	GPIO Port 0.7	I2STX_CLK	SCK1	MAT2.1	00
17:16	P0.8	GPIO Port 0.8	I2STX_WS	MISO1	MAT2.2	00
19:18	P0.9	GPIO Port 0.9	I2STX_SDA	MOSI1	MAT2.3	00
21:20	P0.10	GPIO Port 0.10	TXD2	SDA2	MAT3.0	00
23:22	P0.11	GPIO Port 0.11	RXD2	SCL2	MAT3.1	00
29:24	-	Reserved	Reserved	Reserved	Reserved	0
31:30	P0.15	GPIO Port 0.15	TXD1	SCK0	SCK	00

The register is named PINSEL0, and its address is given as 0x4002C000. We've just learned about bit shifting, so you're ready to hear this: to change a function in this register we need to set the appropriate bits in the memory at this register.

However, this begs a question – We have the address, which is just a number (an unsigned integer, in fact). To bit shift, we need to access the memory at that address, and therefore we need to have a pointer whose value stores that address. How do you make a pointer from an integer?

Naively, you might think that you could just write:

```
unsigned int* PINSEL0 = 0x4002C000U;
```

However, most compilers will warn you that this is not a good idea, and some compilers (such as the ARM compiler we'll be using, at least with its default settings) will consider this to be an error. The problem is that although pointers can be read as though they are integers (for instance, if you just want to print out the address of a variable to see if it is in the right place), integers are not pointers. That is, it doesn't work both ways. We need to tell the compiler that we know what we are doing and that we 100% intended to do this. We usually use a cast for this purpose. Therefore, it is more correct to write the code like this:

```
unsigned int* PINSEL0 = (unsigned int*)0x4002C000U;
```

Why are integers and pointers not the same?

If memory is addressed via integers, why can't we just set a pointer equal to an integer without a cast? The problem lies in how pointers are manipulated by the CPU. To avoid having to write out a big hex number each time, let's consider the following, simpler (and invalid) addresses:

```
unsigned int* ptr = (unsigned int*)0x4;

char* charPtr = (char*)0x4;

unsigned int notAPtr = 0x4;
```

Now, here are three questions for you: presuming that we are programming an ARM cortex M, meaning that integers are 4 bytes long, chars are 1 byte long, and memory is byte addressable, what are the values of the variables after the following operations are performed?

```
ptr--;
charPtr--;
notAPtr--;
```

When we decrement or increment a pointer, the compiler automatically does this in the unit of the thing it points to. Therefore, ptr now holds the value 0x0, since decrementing it decrements its address by the size of one integer. Since chars are 1 byte long, charPtr now holds the value of 0x3, and notAPtr holds the value 0x3 as well because decrementing a single integer by one does not have to comply with pointer math rules.

So now imagine you are a compiler and you encounter the following statement:

```
int* ptr = (0x3-2);
```

What are you supposed to do? What did the programmer intend? Did they want to make an integer whose address is 0x1? Did they instead want to create an integer pointer that stores a negative address, and subtract twice the size of an integer (so 8 bytes) from 0x3[11]? Do either of those two operations even make sense? Since there is so much ambiguity and the potential for abuse, integers must be cast and integers are not pointers!

Arrays and pointers – siblings, but not twins

You may not be aware of this, but arrays and pointers are treated similarly by the compiler. In fact, anything you can do with an array you can do with a pointer, but not everything you can do with a pointer you can do with an array. Let's illustrate:

```
int a[10];
int* b = a; //notice no dereference or ampersand!
int y = b[1]; //totally OK
int z = *(b+1); //Totally ok and z now contains b[1]
int q = *(b++); //Totally OK, q now contains b[1], b is incremented
int w = *(a++); //NO! Compilation error
```

Huh? Why is that last line a problem when everything else seemed to have no issues? Arrays, in terms of memory allocation, are just allocated via a pointer, and the array (in our case, the variable "a"), just stores that address. As such, if we create a pointer and set it equal to the array, what we are doing is just copying the array address into the pointer. At that point we can access and modify that pointer all we want. The third line might be a bit weird, since b was declared as a pointer and not an array, but the compiler doesn't mind and indeed sometimes this is a very useful thing to do. But why did the last line fail?

[11] Which will wrap around to 0xFFFFFFFF, since pointers are unsigned integers and therefore cannot be negative

The problem lies in the fact that a++ would modify a itself. Our array, a, is a variable that is used to access all elements of that array. So consider this:

```
a++;
int y = a[0]; //What is y now?
```

If a were[12] incremented, then y should hold the second value in the array. But then how do we access the first value again? It's ambiguous and may be lost, so we just call it a compiler error and don't allow it.

Void pointers

A void pointer is a pointer that does not have an associated data type. It's best to think of them purely as addresses. You create one like this:

```
void* ptr;
```

Void pointers are extremely useful when you need to manipulate memory but precisely what that memory contains is irrelevant or unknowable at the time the code is written. For example, in our RTOS we will want to create threads that take in a void pointer as a function argument. This is because our RTOS doesn't care what the application programmer will eventually send to their thread, and in fact our RTOS shouldn't be concerned about this. However, our RTOS does need to make it possible for application programmer to pass **something** in.

In order to use a void pointer you usually must cast it. For instance, like this:

```
int x = 4;
void* y = &x; //OK, no worries
*y = 5; //NO! Compiler error
*(int*)y = 5; //OK, we're good!
```

Why did we have to cast it? Think about what a pointer really is – it is a memory location to something. The compiler doesn't know the data type of a void pointer, so it has no idea whether it is valid to set the data there equal to 5 or anything else. For example, what if, instead of an integer, it was a pointer to a char? We could set it equal to 5, but not 23432. What should the compiler do if asked to write a bigger value than should fit? Again, not much – it should tell you there is an error and let you figure it out.

Function pointers!

You may have never heard of function pointers, but we need to know about them to understand our RTOS. Specifically, we will need to write functions that take other functions as input, but these functions are written by the user! Wow.

Let us say that we have a function, `int myFunc(int x);` We create a pointer to that function like this: `int (*funcPtr)(int) = &myFunc;` The function pointer's return type must be correct, as must its argument list, but the arguments are usually not named (so we just wrote "int", rather than "int

[12] Yes, "were", not "was". Look up the subjunctive tense.

x”). In order to use this function pointer to call the function, we write `(*funcPtr)(10);` which calls the function myFunc.

Malloc and Free

In C++ you learned about the keywords **new**, which creates a new object with a given type, and **delete**, which deletes it. C is not an object-oriented language, but we still need a way to dynamically allocate and release memory. The two functions that do this are malloc (Memory ALLOCate) and free (…FREE).

Malloc returns a void pointer but, in C, you don’t have to cast it. This is because the left side of the call to malloc determines the data type. To use malloc you need to use the “sizeof” operator, which is a keyword in C. It looks like this:

```
int* myInt = malloc(sizeof(int));

int* arrayOfSize10 = malloc(10*sizeof(int));
```

Note that to make an array of something you just multiply the output of sizeof with the number of elements you want. Malloc returns NULL if there is not enough memory to allocate.

Malloc allocates memory on the heap – a data region that is entirely dependent on the programmer to maintain. When you are done with memory you’ve allocated, you must call free, which releases the memory back onto the heap. The call to free takes the name of the pointer you previously allocated, like this:

```
free(myInt);
```

In an embedded system we want to avoid using dynamically allocated memory repeatedly. This is for two reasons:

1. Never trust programmers. Programmers often forget to call free when they should, and therefore the memory is never released. This is a memory leak, which is really bad when you have limited memory in the first place.
2. Malloc and free are usually implemented quite simply on microcontrollers, and memory can get fragmented. Essentially, there might be a kilobyte of free memory available, but because you kept allocated and releasing single bytes, the heap is broken into one-byte chunks. It’s possible to defragment, but that is computationally intensive. Therefore, the heap looks full but it isn’t.

In general only allocate onto the heap if you are very certain you need to! A bad idea would be to constantly malloc and free space for each running thread. Threads run many times per second, and you will rapidly fragment memory. A good idea would be to dynamically allocate space for all threads at once then only call free if the kernel encounters and error and has to return to a safe mode. An even better idea would be to set up a linker script so that you know exactly where your threads go at all times and never use malloc in the first place. Do what you need to do.

To close this section, I should mention that there are other functions in addition to malloc. The common ones are calloc (Clear and ALLOCate, which sets the memory equal to zero before giving it to you) and realloc (REALLOCate, which resizes the memory already allocated, copies things over, and frees the original). They are more rarely used so I will not go through them here.

Preparatory Project 3: Exercises

Note: only exercise 1 strictly needs the microcontroller, but you should do all of the exercises on the microcontroller and not a PC with a C compiler. Otherwise the pointer sizes may not be right.

1. Dynamically allocate an array of integers as follows: the user will push the joystick up to increase their desired size and down to decrease it. You may choose to use printf to see the size, or just count. Once the user presses the pushbutton allocate the array and initialize it to the numbers 1 through the size. Print the array using PuTTY. Free the array afterwards.
2. Print the results of the sizeof operating for all integer data types (char, short, int, long int, long long int). Did anything surprise you?[13]
3. Create pointers to variables of all of the integer data types. Use malloc to ensure that they are valid addresses. Print the pointers out using printf by casting them to uint32_t – do not dereference the pointers, we want to see the address they contain. Then, increment each of your pointers (that is, use the ++ operator) and print them again. Compare the addresses. The difference between the incremented address and the original address should exactly equal the results of the sizeof operator in exercise 2. Free your pointers when you are done.
4. This one is fun: create a number of functions. How many is up to you, but it must be at least two. All the functions must return an integer and take an integer as an argument. I recommend doing something in those functions that is simple, like printing "In function 10! The input argument was…" and the output the input argument. Next, create an array of function pointers with the appropriate type, and run all of them using that array in a for loop. The for loop's body should be a single line that just calls the function at the given array index with the index variable as the function's input. Note that you do not need to free your pointers because you did not use malloc.
5. Create a function that takes a void pointer as an argument and returns nothing. In that function, cast your void pointer to a pointer of type int, and, if it is not NULL, add 5 to the integer stored at the address stored in the pointer. Test your function thoroughly in main. If you use malloc in your testing, remember to free your pointers.

Closing thoughts

In this chapter we learned how to set up a project in our IDE and how to write information from the microcontroller to the PC. Then, we explored memory mapped peripherals, and ended with a deeper dive into memory. By this point you should be feeling that you can do bit shifting if you have to and you get the general idea of how to set and clear bits. You should understand that there are special memory locations in the chip that need to be manipulated in order to set certain settings. You should have a good idea about what pointers are and what malloc does. However, you will likely be asking yourself: "How was I supposed to know all of this?" Especially, how were you supposed to know that LPC_GPIO1 controlled certain LEDS or which memory addresses were used to read from the ADC?

[13] You'll often see us using <stdint.h>, which gives us access to the types intN_t and uintN_t, where N is the number of bits and typically may be 8, 16, 32, and sometimes 64. This is a preferred method, since we can be assured that if we need a 64-bit integer we get one.

At this stage you are not supposed to know these things at all. We are about to leave the preliminaries and embark on new phase in our RTOS journey. We are going to learn how the ARM Cortex M chips work on a deeper level, and we’re going to start our first dives into the documentation. By the time we’re done, you’ll have a good idea of where to search for this information for yourself.

Chapter 3: ARM Cortex M Internals

In this chapter we are going to go over the ARM Cortex M in as much detail as is necessary for our RTOS. We will create the skeleton project that will eventually become the RTOS and test some key components of it. Specifically:

1. We will learn about how the stack is organized and begin our study of the rules that govern how the compiler uses the stack for programs
2. We will learn about interrupt driven programming and the difference between Handler mode and Thread mode
3. We will begin a very basic level of assembly programming – unfortunately, we just can't avoid assembly for RTOS development
4. We will learn about two important interrupts: systick, the internal timer interrupt, and PendSV, the pendable software service call interrupt, both of which will be used to switch between threads
5. We will learn about the documentation for the ARM cortex, and start peeling back the layers of how to find out about the chip on your own

This chapter is a bit heavy on the Arm Cortex theory. It is recommended that you read, understand, and ask questions before attempting the lab project.

This chapter will consist of a background section and your first lab project. The background section should be read first, but you may want to follow along with it by writing test code to see what happens.

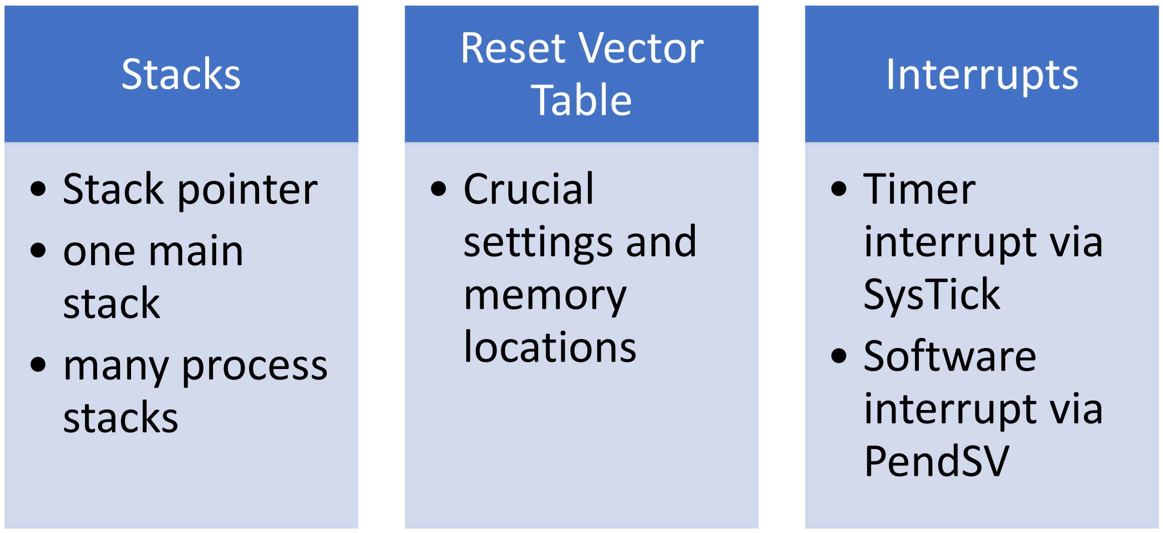
Understanding the ARM Cortex M

Common features and code organization

The ARM Cortex M family of microcontrollers is diverse, but they all share commonalities. ARM the company creates its microcontroller architecture then licences it out to other companies who manufacture the chips. This means that chips made by different manufacturers might differ in certain specific things, for example what exactly pin number 15 does, but since they descend from the common architecture set by ARM they will agree on many more. For this reason, we are going to write our RTOS in as generic a way as possible. This means that, rather than relying on pre-set structures like LPC_GPIO1 (which is specific to the LPC line of microcontrollers and is a macro written by ARM), we will instead define our own memory addresses. Where this is not possible, either because it would be far too much work or because we need to directly interact with certain hardware on our specific chip, I will make sure you know.

An overview of the important parts of the Cortex M chip for RTOS development

The design philosophy we are going to follow for our RTOS is to not do more than is necessary to build a functional RTOS – we are going to keep our RTOS simple and self-contained. Therefore, we need to know only a few things about the ARM Cortex M (hereafter the "Cortex", even though that is technically incorrect). There are three key components that we need to understand for this chapter. One frustrating thing about development of low-level code is that we often need to understand everything at once to make sense of it. I am presenting these topics in a linear order because I can't teach three things at once, but that means some concepts will be used before they are introduced. Be sure to read and come back to parts that didn't make sense before. As you read on many questions will be answered.


Stacks
Stack pointer
one main stack
many process stacks
Reset Vector Table
Crucial settings and memory locations
Interrupts
Timer interrupt via SysTick
Software interrupt via PendSV

Understanding the Stack

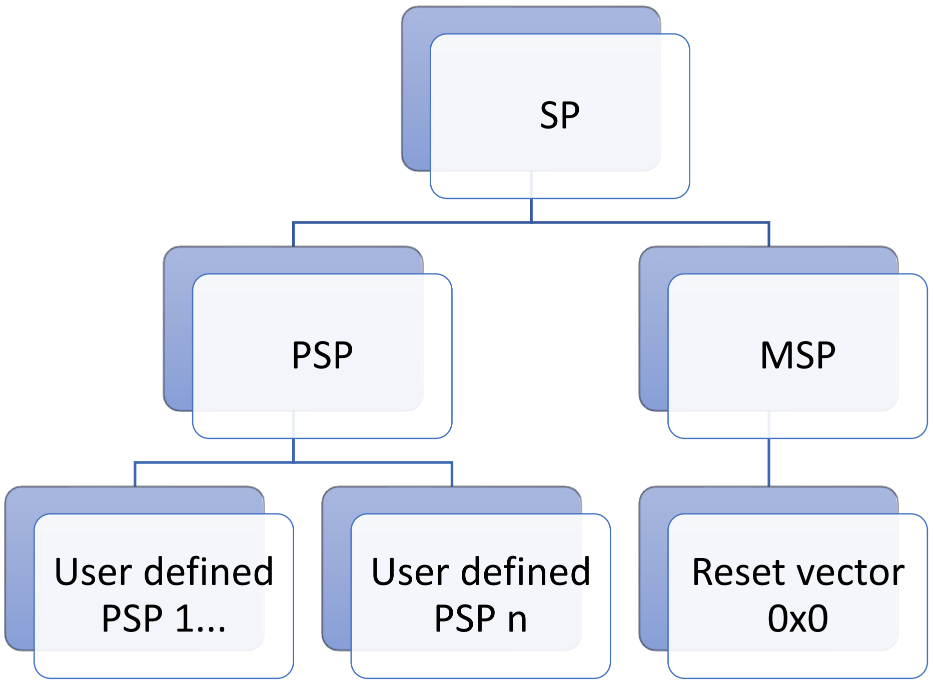
C is a stack-based language and the Cortex M is often programmed in C, so we need a way to access a stack. Interestingly, the stack is not a requirement for programming the microcontroller. A suitably advanced programmer could do the whole thing in assembly without ever once using a stack. However, it is best to not go this deep into the Cortex at this time.

The Cortex is organized in an interesting way[14]. There is a single memory location that the CPU can access to put things onto or take things off the stack. This is called "SP", the Stack Pointer, and it is a processor register that we need to be very careful with. Whenever the CPU runs a PUSH or a POP instruction it does so using SP itself. But that's not the whole story.

SP just stores a location in memory that contains the top of the stack. The Cortex has two other locations, known as MSP and PSP, that are called "banked" registers. These registers allow us to maintain multiple stacks, storing the one that is not in use ("banking" it). One of them, the Main Stack Pointer, is the default. MSP's address is set by the startup code and is stored in the vector table at address 0x0 (see below). When the chip first boots, that address is retrieved and placed into SP. As PUSH and POP operations are performed, SP changes but the original location of MSP does not, meaning that if we ever want to reset the stack entirely, we just point SP to the original location of MSP again. PSP is

[14] "Interesting" in this context means "complicated"

set by user code and initially it contains random data that may or may not be a valid stack location. Graphically this looks like this:



In our OS, MSP will be used only for OS-specific things. Threads, on the other hand, will all have their own stack pointers and therefore their own stacks. This way, threads do not accidentally overwrite the data from other threads or the OS itself. When we want to load a new thread, we need to change PSP and then tell the processor to look at PSP for the new value of SP.

You might be asking why we would do all of this. Why use both PSP and MSP? Couldn't we just create an OS Stack and store that in PSP when we needed it as well? Unfortunately (this is the complicated part) the Cortex uses MSP exclusively whenever we are inside of an interrupt service routine. This means that we need MSP to remain unchanged and accessible, since it's not up to us when the processor uses it. Our job is to do the following steps:

1. Let the microcontroller set MSP on its own but never modify it ourselves
2. Use MSP's address to locate valid addresses for a bunch of stack pointers, one per thread, that will eventually be switched out (see the next chapter about the switching thing. Today we're just going to deal with one MSP and one PSP)
3. Tell the microcontroller to use the stack pointed to by PSP when we start running our threads

However, before we can do any of this, we need to know where to find the initial location of MSP in the first place. We find that in the Reset Vector Table.

The Reset Vector Table

Booting any computer is a chicken-and-egg problem. The computer must execute code to set itself up, but part of the setup is to initialize the computer so that it can run code! How this is solved depends on the architecture, but in the Cortex a special region of memory has been set aside and hard coded to contain various important settings and addresses. This is called the Reset Vector Table (or just the vector table). You can begin your journey into ARM's (terrible) documentation by learning as much as you can about the vector table from this link: https://developer.arm.com/documentation/dui0552/a/the-cortex-m3-processor/exception-model/vector-table

The vector table in the Cortex is always at memory address 0, regardless of manufacturer. Astonishingly, this means that address 0, which is usually referred to as "NULL" by some novice programmers, is a perfectly valid memory address that can be dereferenced on the Cortex. The vector table itself is just an array of unsigned integers (usually interpreted as pointers of various types).

Address 0x0 stores the address of the Main Stack Pointer (MSP, the first stack pointer that is used upon boot) and address 0x04 stores the address of the reset routine, which is the function that runs as soon the processor boots. The Cortex's boot sequence is to first load MSP, then jump to the function whose address is located at address 0x04. If there is no function at that address, say because we set up our code wrong, the processor will enter a fault condition. User code is also usually responsible for filling the vector with other addresses, for example by writing the address of an interrupt function to the appropriate location in the vector.

We are not going to write the code that fills the vector, but you can look at it by opening Device->system_LPC17xx.s (Startup), which is very instructive. The vector definition begins on line 59, and you'll see that it includes the location of the initial stack pointer and a function called Reset_Handler, among other things.

We will not overwrite the Reset_Handler (see line 125 of that file to see what it does – basically it just calls an initialization function and then main), but there are two symbols that are defined that will eventually be of interest to us. They are PendSV_Handler (vector address 0x38, line 73 of the startup file) and SysTick_Handler (vector address 0x42, line 74 of the startup file). The startup code we are given provides functions that do nothing – they literally just return from the functions when they are called. This is necessary because, during startup, these functions might get called, so the microcontroller needs something to be there, but they shouldn't do much because what they do is up to the programmer. When we are writing our RTOS we are going to overwrite them. Speaking of which…

Interrupts

Our final bit of theory in this chapter has to do with interrupts. An interrupt is a mechanism by which a normally running program can be halted in the middle of what it is doing ("interrupted", as it were), another piece of code can run, and then the original program may start from where it left off[15].

There are many types of interrupts on the Cortex and many ways to trigger them. For our RTOS there are only three of any importance: SysTick, which is a timer-based interrupt that we can configure to happen periodically, PendSV, which is an interrupt that can be triggered from software itself, and SVC, a kind of supervisor to PendSV that can be called from software but that has special privileges that make it attractive for ensuring things run on time. For this chapter we are going to focus only on PendSV and leave SysTick to our discussion on scheduling. We will introduce SVC much later when we are ready to do advanced timing.

[15] We say "may" because some interrupts, such as a Hard Fault interrupt, mean that the program caused such a problem that the CPU had to stop what it was doing completely.

System Calls

PendSV means “Pendable Service Call”, and “service call” is an ARM term that literally[16] everyone else in history has referred to as “system call”. A system call is a way for a program to access the kernel. On more complicated operating systems, the way this works is that a userspace program calls a specific function that is always at a known location in memory. That function then transfers data to the kernel, which handles it from there, does what is asked, and returns to the program. For example, a userspace program might perform a WRITE system call to write data that it has to a file on the hard drive. The program passes that data to the kernel and tells it where to write to, and the kernel handles the hard drive interfacing.

The “Pendable” part just means that this is an interrupt that we tell the chip to run only if it is not already doing something else in another interrupt. This will be important for scheduling later, where the periodic interrupt governed by SysTick might itself call the interrupt governed by PendSV. We don’t want to interrupt the interrupt, so we tell the CPU that whenever it’s ready it can go ahead and run the PendSV interrupt routine. We will rely on PendSV to do most of our low-level context switching throughout this entire course, even when we eventually get the higher priority SVC up and running.

Interrupt Service Routines

Interrupts have associated Interrupt Service Routines, or ISRs. These are honestly just functions that we must write a little more carefully than normal. In this lab we are going to write a PendSV handler, but we are going to have to do it in assembly (ugh) because we will eventually be doing things that are simply impossible to do in C.

Before we tell you how to write this, you should know a few things:

1. Keep your ISRs short. The CPU should not be running all the code inside of interrupts, and the application that the microcontroller is running should be the thing running most of the time
2. It is possible to interrupt an interrupt, but we are not going to do that for our RTOS[17]
3. The PendSV ISR is really just a function, but we don’t call it like other functions. Instead, we manipulate the chip’s interrupt registers to indicate that we want the chip to run the function when possible. The chip then jumps to the function for us
4. On the Cortex, no matter what, every time you trigger an interrupt MSP is used. You can never use PSP in the interrupt and trying to do so leads to a guaranteed hard fault. This fact is going to make our lives harder in the next chapter, but for this chapter we can ignore it

If you still aren’t clear about what an interrupt is, don’t worry about it. The important thing to understand is that it is a function that we are going to write, and then call from our threads, that allows us to interact with the kernel.

To understand what happens during an interrupt, we need to understand the Cortex’s registers.

[16] Figuratively speaking, of course

[17] We are going to chain them together, eventually, so that one interrupt immediately triggers another but only once it’s done. This is called “tail chaining” and it is not the same as interrupting an interrupt, nor does it have nearly as many pitfalls

The important registers

Note that this discussion is 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 you really, really shouldn’t mess with them in this course.
- 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. It allows the processor to return from certain types of function calls and the compiler is really the only thing that needs to know how it works
- R13 is known as SP, the stack pointer. We’ve already introduced it and 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 this relevant? These registers will be used by the CPU to store information relevant to a thread. When we do things like switch between threads or call functions, the registers must be preserved. So what does that look like?

What happens when a function gets called?

First, a disclaimer – what is explained here is specific to the way that our compiler (and some others but not all) implements the function call. It is not necessarily a requirement for all code written for the ARM Cortex M3 chip to conform to. However, it is a common way that compilers for ARM chips do function calls. See Appendix A for more information.

When a function call happens exactly eight of the registers are pushed onto the stack in a very specific order. It is important for us to understand this order, because we are going to manipulate the stack a lot. xPSR, PC, LR, R12, and R3-R0 are pushed onto the stack for you by the hardware in that order, xPSR first and R0 last.

When our function runs, the compiler may set the registers however it wants. When the code reaches the end of the function, the registers that were pushed onto the stack are popped back off, restoring the contents of the registers before the function call.

Now, why is this relevant? Well, the important register for returning from a function call is LR. LR stores the return address, so if it gets overwritten, what happens? You might think it means we can never return from our function, which is partially correct. What you'll see is that we want to manipulate LR during a context switch so that, rather than returning to the thread we started from, we go to another thread. This forms a kind of "indirect return" or jump. Now, if we manipulate LR incorrectly…we jump to a random location and get a hard fault. Similarly, if we manipulate the stack incorrectly and accidentally set SP so that it can't find the correct value of LR, we are going to fail to return as well. Either way, we need to be careful with what we do to our registers.

In the following lab project we are going to see that this isn't the whole story, and that we need to carefully manipulate LR in order to return from an interrupt.

Lab Project 1: The Scaffold of our RTOS

In this project we are going to begin creating our RTOS. Specifically, we are going to:

1. Obtain the value assigned to MSP from the vector table
2. Use that value to create one process stack
3. Switch the microcontroller to use the process stack we just made
4. Create a very simple but assembly-based PendSV ISR
5. Trigger our interrupt and view the processor's state in the debugger to investigate what happens when we are in an ISR

Setting up your Project

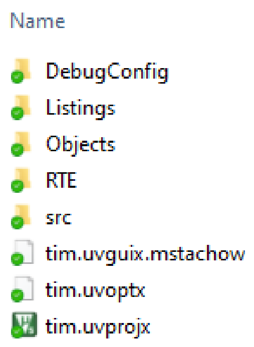
It is a good idea to set up your project carefully and thoughtfully. This project is what we are going to be expanding upon in each subsequent lab project, so organizing it well is a good idea.

I recommend that you begin by first creating a new project in a new folder, or just use the folder structure given to you in Tim_Step0.zip. Do not use the same project you used for the preliminary projects but follow the steps chapter 2 to set up a new one. This time, however, create a new folder, "src", that contains all the source code. If you made your own project, copy only the source code from the "Tim_Step0" project into your src folder[18].

You can name your RTOS anything, but usually people give them slick, 90's era dot-com sounding names like "FreeRTOS" or "CheetOS". I will call my RTOS Tim[19].

[18] It is common to create many sub-folders for larger projects. You may, for instance, have a "lib" folder for any common libraries you write, a "bin" folder to store precompiled binaries, and many more. Our RTOS will not be too many lines of code so we will stick to a single "src" folder.
[19] This is not a Monty Python reference, I swear. The person who taught me the most about embedded systems was named Tom, so I've chosen to indirectly honour him by naming the OS something close to his name. Also, I can keep calling my OS an RTOS even if we are doing nothing with timing, it just becomes a Real Tim OS instead.

Presuming you've done everything correctly, your folder structure will look like this:



```
Name

DebugConfig
Listings
Objects
RTE
src
tim.uvguix.mstachow
tim.uvoptx
tim.uvprojx
```

Next, your main function should be written to do the following:

1. Call the SystemInit() function
2. Print Hello, world! (This is for more than just testing. The printf function must be called in main before we start doing weird things with interrupts)
3. End in an infinite while loop

If you don't feel like doing this part, you can use the step0 source code. It contains a set up, but empty RTOS project that does nothing but initialize itself and print Hello, World.

As we progress in these labs you will be adding to your previous projects. I will provide source code that fully implements each lab so that, if you don't get it working one week, you aren't completely stuck. Since this project will be ongoing throughout the entire course, I will be referring to files written in previous labs as we move forward.

Following best practices for the labs

This course is not just about writing and understanding an RTOS. It is also about developing good programming practices that can be used to manage relatively large software projects. In this section I will describe a minimal set of best practices that you should follow.

First, I strongly, strongly recommend that you use version control software for this lab. GitHub is good enough, but more sophisticated git repositories or anything else you'd prefer should be used. Do not, and I cannot stress this enough, work in a single folder without version control. You will, for sure, mess something up at some point, have no idea what it is and no way to roll it back. Slightly better but still bad is to save older versions of your code (like main_old1.c). In theory, yes, you can roll it back, but version control lets you do this for an entire project all at once, which is far better and clogs up your directory tree a lot less. If you are wondering, Tim was developed using a private repository, since I don't want to release the whole thing to you until you've done each project.

Second, comment your code. When in doubt, comment too much. The various pre-made zip files will be extensively commented so that you can learn from them, and you should probably comment around that level at first until you fully understand what is going on.

Third, organize your code. In this first lab we are going to do a lot of the hard parts for you. By the time we are at the last lab you will be fully in charge of figuring out which files and functions you need. Organization is a great idea. I will explain each of the files and functions I am creating and the rationale behind them, but you are free to create as many helper functions and files as you want. There are several philosophies on how to do this:

1. Every function gets its own *.c file – this is probably overkill, but the huge benefit here is that it makes it much easier to find a specific function in your source code and debug it. This is how many implementations of the C standard library are written
2. Every group of functions that make sense together get their own *.c file – this is the approach I will be using. For example, I'll be writing a "kernel" library and a "thread" library
3. Everything is written into a single file – don't. You will 100% be unable to find and debug things, and it will make compilation and unit testing impossible

Fourth, stick to a consistent style that is reasonable and professional. I am not going to force you to use a specific type of bracket convention or anything else, but you must be consistent and professional.

Finally, test as you go. Test the functions you write with good inputs – that is, make sure that the function does what it is supposed to when everything is perfect – then test with reasonable bad inputs and edge cases. Don't try to move on unless you are certain that your individual functions work.

Part 1: Creating two Stacks

At this point you should have an empty project whose main function prints a suitably welcoming message to the screen. Our first exercise will be to create the two stacks, MSP and PSP. We are going to approach this in two steps:

1. Create an RTOS library, _threadsCore.h and its associated _threadsCore.c file, that will be used to create these stacks
2. Test in main that it works

Creating your _threadsCore library

You may wonder why we are using underscores. The simple answer is that, although we don't need to, adding an underscore usually makes it so that any files created by ARM or another third party don't conflict with what we are doing. "threads" is a fairly common name for a library, so calling it "_threadsCore" will reduce the chances of us overwriting or conflicting with other libraries already present in our IDE.

- Create two new files in your src directory: _threadsCore.h and _threadsCore.c
- In _threadsCore.h, use include guards to prevent multiple inclusion

To create a library like this you first write the function declarations in the *.h file, and then the function definitions in the *.c file. The *.h file typically also includes things like struct definitions and, sometimes, inclusion of other libraries. You then add your *.c file to the appropriate source group in your project.

_threadsCore.h should include <stdint.h> for now, since we're going to need some of the important integer definitions that header files provides, and <LPC17xx.h>, since we're going to need a few register access functions.

Write the following functions:

- `uint32_t* getMSPInitialLocation(void); //Obtains the initial location of MSP by looking it up in the vector table`

- `uint32_t* getNewThreadStack(uint32_t offset); //Returns the address of a new PSP with offset of "offset" bytes from MSP. Be careful with pointer arithmetic! It's best to cast to an integer then back if you're not sure.`

- `void setThreadingWithPSP(uint32_t* threadStack); //Sets the value of PSP to threadStack and ensures that the microcontroller is using that value by changing the CONTROL register`

Here are a few things to know about these three functions:

1. The ARM Cortex's stack grows down. Therefore, getNewThreadStack should *decrement* MSP by the offset value, not increment it
2. You may want to include error checking in getNewThreadStack. For instance, it should not be possible to create a new thread stack whose address exceeds the thread stack size initially set in your setup step
3. Setting PSP should be done via the _set_PSP(uint32_t newPtr); function, available automatically by including LPC17xx.h. Note that this function takes a uint32_t, not a pointer, so you need to cast your new thread stack pointer accordingly.
4. Setting CONTROL should be done via the _set_CONTROL(uint32_t newValue); function, also available in LPC17xx.h. To switch to threading mode, CONTROL must have a 1 in its 1st position (so, bit 1, the bit after bit 0).

Stunningly Important Caveat: This paragraph will look like magic in this chapter. I'll explain it in more detail later, but for now trust me. In your getNewThreadStack function you need to ensure that the stack pointer is divisible by 8. This is an internal ARM thing: the stack simply must be on an 8-byte boundary. If you do not do this you will have to do some truly exotically weird things to get context switching working, and there is a chance that your OS will have random bugs that you don't know how to fix. The simplest way to do this is to first compute a potential new stack location, check if it is divisible by 8 using the modulo operator, and if not add to the offset. Luckily, the stack will either be divisible by 8 or by 4, so you only need to offset the stack by an additional sizeof(uint32_t).

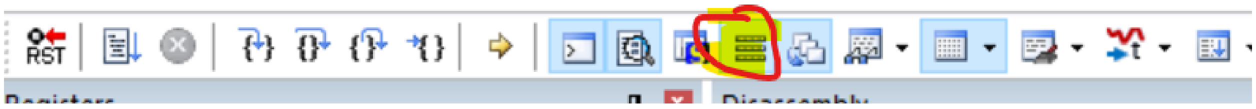
Testing your _threadsCore library

Perform at least the following two debug tasks:

- In main, begin by printing out the value of the initial MSP. This can be achieved just using printf.
- Then, create a new thread stack 512 bytes below that value of MSP and set the processor to use it. Use the debugger to confirm that this worked (see below).

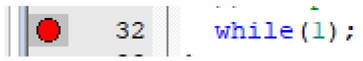
Using the Debugger

Unfortunately, some of the settings we are using are not obvious and cannot easily be printed out. We need to use the debugger. First, ensure that your code is loaded onto the microcontroller, then begin a debug session by pressing the "Start/Stop Debug Session" button, . Then, ensure that the registers window is open. If not, when the debug session starts, click the button highlighted below:



The register window shows you…the registers! If they aren't already expanded, expand the "Banked" list (which shows MSP and PSP), the "System" list, which shows CONTROL (and others), and the "Internal" list, which doesn't show registers but does show important information about the state of the processor. MSP, PSP, and CONTROL are the only three registers we are going to care about at the moment. Presuming that you did everything correctly so far, you should see CONTROL is 0x00, PSP is likely a random memory location, and MSP is set to some valid value, most likely starting with 0x1…

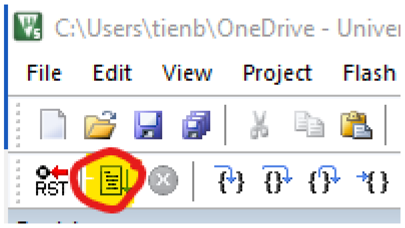
Set a breakpoint in main at the start of the infinite while loop by clicking next to the line number on which the while loop is written. It will look like this:



```
32    while(1);
```

This will stop the debugger at this point. In our case, this means we want to run our entire main and then stop so that we can examine memory (you cannot examine memory while the code is executing).

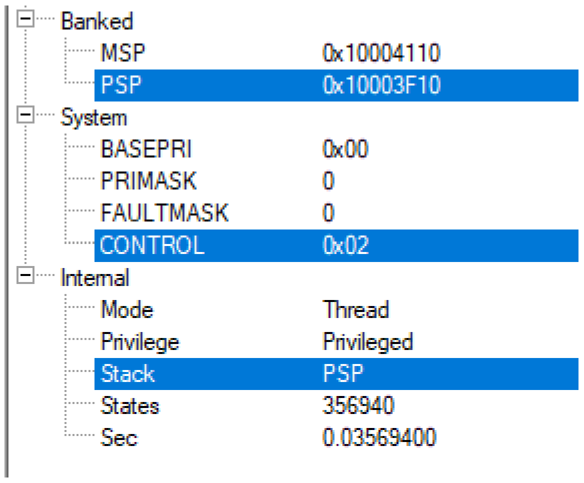
Finally, click the "run" button:



This will run your code until it encounters a breakpoint. Examine the registers now. You should see a few things:

1. PSP will be set to a reasonable value, also starting with 0x1…
2. If you subtract MSP-PSP, you should get a value less than 512. Remember these are hex values, so you'll need to convert
3. CONTROL should be set to 0x02
4. The "Mode" in the Internal list should be "Thread"
5. The stack should be "PSP"

Here is what my window looks like. Note: your memory locations will almost surely be different, because it depends on how you wrote your main.



```
Banked
    MSP          0x10004110
    PSP          0x10003F10
System
    BASEPRI      0x00
    PRIMASK      0
    FAULTMASK    0
    CONTROL      0x02
Internal
    Mode         Thread
    Privilege    Privileged
    Stack        PSP
    States       356940
    Sec          0.03569400
```

If you've got that far, success! You've successfully created a valid PSP and switched your microcontroller to over to using it.

Part 2: The PendSV Interrupt

Buckle up, it's time for some assembly. Not much, I promise. Think of this part of the lab as a guided tour of some assembly information that we need to know. You won't need to come up with the assembly yourself. In the next lab project I will reduce the amount of support I give for the assembly stuff. For now, just copy the lines and make sure you understand them.

Create a new file, "svc_call.s", and save it in your src directory then add it to your project[20].

The assembly file preamble

We need to add a few things to the start of that file. I'll show you each line you have to add, then explain it. Add these lines in order, starting on line 1 of the file. Frustratingly, the ARM assembler requires us to indent some things and not others. Each of the lines I'm about to show you must be indented by at least 1 whitespace character. I recommend the tab character.

```
	AREA   handle_pend,CODE,READONLY
```

This line defines a new "AREA" in the executable. This is a way for the linker to know that what exists in this file is important and must be placed into a single, contiguous block of space. I named this area "handle_pend", but you can name it anything that is a valid variable name. The area holds CODE, and should never be overwritten (so it is READONLY).

```
	GLOBAL PendSV_Handler
```

We are about to write a function to handle the PendSV interrupt. This function must be declared globally, because it will be referenced by other files we have written.

```
	PRESERVE8
```

This line is required because we are going to be messing with the stack eventually. What it does is tell the linker that we promise the stack will always lie on an 8 byte boundary. This is one of those "you sort

[20] You can do this in one step: right click the source group, add a new item, name it, choose its type, and set its location

of have to know the chip super well" things to even know that it exists, so I'm just telling you: any time you are messing with the stack in assembly, always PRESERVE8[21].

Now, we need to create a symbol that tells the linker where our PendSV handler function will be. We've started this with the GLOBAL definition above, but that only tells the linker that a symbol of that name is forthcoming, not that it has been created. We must name it "PendSV_Handler". This name is not arbitrary. It is the same function name that is used in startup_LPC17xx.s to define the PendSV interrupt handler. We must be sure not to change this name unless we also want to modify the startup file.

Remove your indent so that you are writing on column 0, and type:

```
PendSV_Handler
```

On the next line, indent again and stay indented for the rest of this file. Anything we write now will be part of the PendSV handler function.

The PendSV Handler

In this lab we aren't going to do much. In fact, we are going to enter the PendSV handler and return from it. However, this will set the groundwork for our context switches in the next lab. First, though, you need to understand a bit about what exactly is going to happen.

Once we trigger the interrupt (which we'll do in C later on) the chip will be in Handler mode and will be using MSP, not PSP. We are going to verify this with the debugger once everything is ready to go. In this lab we are only going to restore LR and return.

Remember above that I said LR needs to be set in a special way to return from an interrupt. This is because we need a way to inform the chip to go back to Thread mode and to stop using MSP and start using PSP. We do this by loading a very special value into LR: 0xFFFFFFFD (that is, seven F's and a D). There are other constants that do other things, but this one specifically tells the chip:

1. You are returning from an ISR
2. You will stop being in Handler mode and go into thread mode
3. You will use PSP instead of MSP

To load LR, we use the "MOV" instruction, which MOVes a constant into a register. The code to do that is:

```
    MOV LR,#0xFFFFFFFD
```

The "#" there indicates to the assembler that what comes next is a constant. If we didn't include it, we'd get a warning.

Now it's time to return. To return from a function we use the BX instruction. B stands for "Branch", which means "branch to a different place in this program". X ensures that we maintain certain important processor state information. In theory we could just use the "B" instruction (ignoring the "X"), but then we have no guarantees that the processor will remain in a state we want.

[21] When I started this project, I most certainly did not know the chip super well. The assembler gave me errors about PRESERVE8, and I had to figure it out. You're welcome.

To use BX, we supply a register. That register must contain either a return address or a code like 0xFFFFFFFD. We will BX using LR, which contains our All-Important Code. The code for this is:

```
BX LR
```

Finally, on the last line of the assembly file, type:

```
END
```

This indicates that there is no more code to process in this file.

Performing the interrupt

At this point we have our interrupt handler written, but we have no way to trigger the interrupt. In general interrupts can be either hardware based (such as the timer or an input button) or software based (which is what PendSV will be). The procedure we will be using to trigger a software-based interrupt is:

1. Write a C function that prepares things then tells the chip that an interrupt is coming
2. Write directly to memory locations reserved for interrupts
3. Allow the interrupt to complete
4. Return from our C function if necessary

Point 4 might sound weird, but you're going to see that once we introduce multithreading, we will never return from the interrupt calling function. It isn't as odd as you think, but we're going to have to wait until lab project 2 to understand how and why we would do this. In this lab we are just going to return from that function.

However, before we do this, we need to be aware that interrupts can…interrupt each other! This is usually desirable but to make it work properly we need to set the priority of the interrupts we are using. If we fail to do this then an important interrupt might get missed because another, less important interrupt had an incorrectly strong priority.

It is counter-intuitive, but we need to give PendSV the *weakest* priority[22]. This is reflected in its name. "Pend" means "pendable", as in "the interrupt is pending", which tells the chip to wait until all other interrupts are done before this one gets triggered. We want it to be a weak interrupt because we do not want to run this interrupt if the chip is already in an interrupt. Why? Because first, we would be in Handler mode, not Thread mode, and second the interrupt that is already running might need us to keep the currently running thread unmodified. By making PendSV weak we can minimize the chances of this happening.

To make this work we are going to start writing the very first part of our kernel, which will be responsible for both setting up and triggering the interrupts.

[22] I am using the terms "weak" and "strong" to refer to interrupts that can be superseded and interrupts that do the superseding, respectively. This is because ARM chips use low numbers to refer to strong priority. Therefore it is ambiguous to say "lower priority". Do I mean "a lower number as its priority" or do I mean "other interrupts may supersede this one"?

Create a new pair of files: _kernelCore.c and _kernelCore.h. These are going to store any of the functions that the kernel oversees. In this lab we are going to write two functions, which we will be expanding upon throughout the course:

```
-  void kernelInit(void) //initializes memory structures and interrupts
                          necessary to run the kernel

-  void osSched(void) //called by the kernel to schedule which threads to
                                            run
```

The kernelInit function will do a lot of setup eventually, although in this lab it is just going to set the interrupt priority. The osSched function is going to be a very important one later. Its purpose is to figure out which thread needs to run next, then trigger the appropriate interrupt and switch between threads. In this lab it is only going to trigger the PendSV interrupt.

An important note: the name "osSched" implies that it is used for…scheduling, and it is. However, we are going to see in Lab Project 3 that we probably want a separate function that is "The" scheduler. It makes sense at that time to re-name this to "osYield" or something similar. I am going to do the renaming later when the name no longer suits, but you may choose to rename it sooner rather than later.

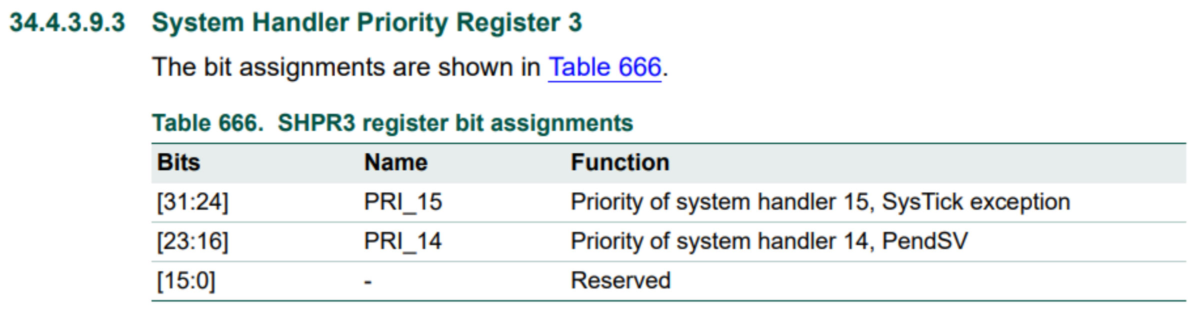
Writing kernelInit (part 1)

First, in _kernelCore.h, we want to define the location of the PendSV priority register. This is a memory location that, when properly manipulated, will enable us to change the priority of the PendSV interrupt. It is by no means obvious, but if we look into the documentation for the chip we see that this memory location is named SHPR3 (System Handler Priority Register 3). Add the following line in your _kernelCore.h:

```
#define SHPR3 *(uint32_t*)0xE000ED20
```

This memory location, 0xE000ED20, is a pointer to a uint32_t. This define will allow us to set the bits of this register with bit shifting operations.

Next, we can go to the documentation to see what the bits in this register do. Here is the table from the documentation explaining it:



34.4.3.9.3 System Handler Priority Register 3

The bit assignments are shown in Table 666.

Table 666. SHPR3 register bit assignments

Bits	Name	Function
[31:24]	PRI_15	Priority of system handler 15, SysTick exception
[23:16]	PRI_14	Priority of system handler 14, PendSV
[15:0]	-	Reserved

We can see that we need to set bits 23-16. This is 8 bits, and we want to set them to the highest number we can, which is 0xFF. So we need to bit shift 0xFF by 16 and bitwise OR SHPR3 with this value.

Now your kernelInit function will consist of only one line:

```
SHPR3 |= 0xFF << 16;
```

This will set the priority of the interrupt correctly.

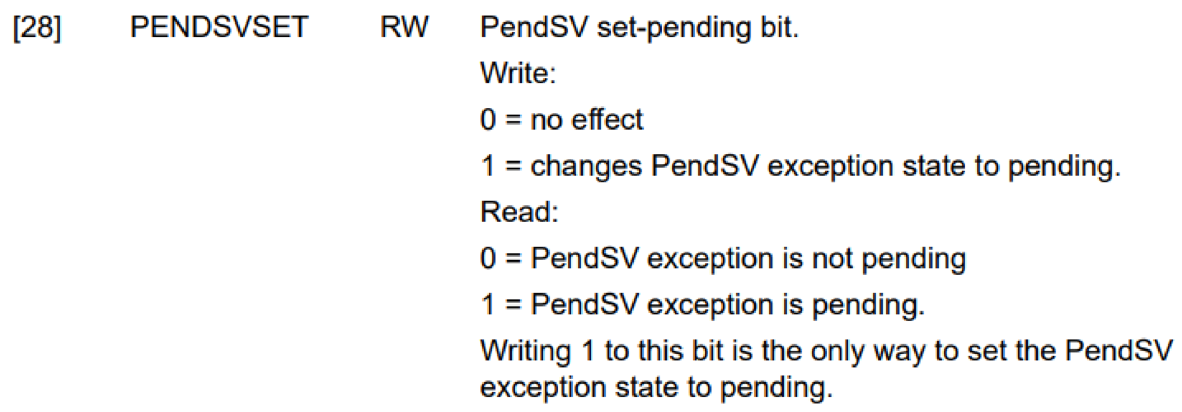
Writing osSched (part 1)

We are going to be spending a *lot* of time in this function, but for now we just need to set a register called the Interrupt Control and State Register (ICSR), then flush the pipeline of the processor.

Once again going to the documentation, we see that the ISCR lives at memory location 0xE000ED04, so add the following line to your _kernelCore.h:

```
#define ICSR *(uint32_t*)0xE000ED04
```

From the documentation we can see that Bit 28 of this register controls what PendSV does:



[28]	PENDSVSET	RW	PendSV set-pending bit. Write: 0 = no effect 1 = changes PendSV exception state to pending. Read: 0 = PendSV exception is not pending 1 = PendSV exception is pending. Writing 1 to this bit is the only way to set the PendSV exception state to pending.

If we "change the PendSV exception state to pending", what we are doing is telling the chip to run the PendSV interrupt when all other interrupts are done. This is exactly what we want! We need to set this bit to 1, so add the following line to your osSched function:

```
ICSR |= 1<<28;
```

Now we aren't quite done. The Cortex is a pipelined chip. This means that it tries to load instructions into memory that are going to run in the future. This is done for many reasons that are well beyond the scope of this course, but the problem that we are going to have is that we need to be sure the pipeline is clear before we try to trigger an interrupt. This way we don't enter the interrupt a few cycles too late after some strange instructions have run. The only way to achieve this is via assembly, and I promise this is the only inline assembly we're going to write in this course. Add the following line to the end of your osSched function:

```
__asm("isb");
```

Note there are two underscore characters, not one. This tells the compiler to run the "isb" instruction using assembly. This instruction just flushes the pipeline. If you didn't understand any of that don't worry. It is necessary but not important to remember. Long story short: keep that line below your write to ICSR and never think about it again.

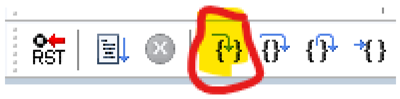
Testing our interrupt code

The time has come to test our code. Theoretically all we need to do is call osSched in our main function and we're done, but since our interrupt currently doesn't do much we won't notice anything has changed. We need to use the debugger.

Modify your main so that it calls kernelInit and then osSched right before the infinite while loop, then remove any breakpoints in your code left over from the previous exercise. Next, add a breakpoint to the first line of your PendSV_Handler function (where you set LR to the value 0xFFFFFFFD). Compile, download, and debug your code, then run it to the breakpoint.

If all went well you should be inside of your interrupt. View the registers window and verify:

1. The stack is now MSP
2. The mode is Handler

Step once using the "step" button: . Verify that LR contains the important constant. Keep stepping until you return to main and eventually get to the infinite loop. If everything went well your processor will now stay in that loop. If something went wrong you will likely notice fairly quickly and the processor will go into the "Hardfault_Handler", whose purpose is to just keep looping with no chance of escape.

Lab Project 1: Submission Guidelines

1. Submit your code, saved as a pdf, to Crowdmark. You will be evaluated on consistency of style (10 marks), usefulness of comments (10 marks), and functionality (20 marks)
2. Separately, submit a brief report. The report should include:
   a. A discussion of how you used the debugger in Part 1, including a screenshot of the registers window proving that the microcontroller is using PSP (10 marks)
   b. A discussion, in your own words, of how the interrupt code works. Include a brief description of each line of code in the functions that we wrote and what they do. You can rely heavily on this lab manual for that, or seek it out elsewhere. Any external documents used must be cited in proper IEEE format. Include a screenshot of your debugging, stopped in the PendSV_Handler function, showing the processor mode and stack (10 marks)
3. The remaining 30% of the lab grade is allocated to the lab quiz

Where are we now?

It may not seem like it, but we've just done a lot! We have a basic framework set up that we can expand in future lab projects, and we are able to do things using interrupts.

Chapter 4: Threads and Context Switching

Introduction

In this chapter and its associated lab project we are going to build upon the framework from before and start executing multiple threads. In a sense, once this chapter is complete, our RTOS will “work” – it will be able to run and switch between multiple tasks, which was about all the very first class of multitasking systems were able to do. We are not going to implement anything about timing yet (so it’s more of a “Task OS” rather than a “Real Time OS”), but it will still be quite the achievement.

The context switching code itself will be written in assembly and I will write a sample for you – it’s way too easy to get lost in the weeds. But your life isn’t going to be that easy. This chapter requires you to make the first of many far-reaching design decisions. In the lab project it is going to be your responsibility to design your threading system, and to answer important questions like: “What is a thread in my OS?” “What data must be stored in a thread for it to work?” and “How does my OS store multiple threads?” There are many ways to answer these questions, so how your OS works will be subtly different from my own. This is exciting! Let’s dive in.

What are threads in commercial RTOSs?

Commercial RTOSs use threads that follow the same basic format: a function that never returns and whose main body lives inside of an infinite loop. Depending on how the scheduling works, that thread either must inform the OS that it is ready to yield control (typically via a call to a function called “yield” or something similar) or the OS itself interrupts the thread and runs another one. However, it can’t be that simple and it isn’t! Consider the following problem:

Let us say that we have a thread[23]:

```
void thread1(void* args)
{
	while(1)
	{
		printf("Hello");
		osYield();
	}
}
```

This should seem odd to you already. When the thread calls yield, the OS switches it out for another one. But what gets “switched out”? Even worse, 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

[23] This thread code is from Keil RTX5, a commercial RTOS based on the open source CMSIS RTOS2.

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"[24].

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".

What's in a context?

We learned in lab project 1 that the CPU has 17 registers: R0 through R12 (some of which are special, but for our purposes we are not going to worry about that), SP (stack pointer), LR (link register, for return addresses), PC (program counter), and xPSR (Program Status Registers, which we *really* do not want to mess with). Registers are used by the CPU to store intermediate results, so it makes sense that the registers' values for one thread are going to be different from those of another. At the very least, the PC register has to be different or else all threads run the same code! So, these registers form part of the thread's context, which must be saved for the current thread and reloaded for the new thread each time we do a context switch.

In addition to that, we also need to store any OS-specific stuff that is relevant. This is the point where you should start thinking about what sorts of things you might need to store. In my sample implementation, for example, I am going to store SP separately, because of how it gets manipulated in my context switches. This is by no means a requirement for you, but I have personally found it easier to save this information. Your OS may eventually need to store quite a bit more than just the registers listed above.

The basic idea of a context switch

Note: When you start writing the code to do this, 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. If you do not exactly realign your stack it simply won't work. However, in general a context switch is "simple", in that describing what you must do is easy but actually doing it is hard.

You begin a context switch with a call to some function. Typically, this is called "yield" if it's the thread's job to determine when a switch is necessary but sometimes it's an interrupt. In this chapter we are going to write yield, and when we get to priority and time-based switching, we will expand what we are doing to use an interrupt approach if needed. Either way, the 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)

[24] Yes, the same idea as "booting" your computer. A PC has to solve 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? See Appendix B for a taste of how this is solved

4. Trigger the PendSV interrupt – we're going to start by using only PendSV. In future chapters we are going to use timer interrupts to *start* the context switch, but PendSV is still going to execute it.

As I stated in the previous chapter, when functions are called, xPSR, PC, LR, R12, and R3-R0 are pushed onto the stack for you by the hardware in that order. So, saving the context means you only need to save registers R4 through R11.

As soon as the function gets called and the eight registers get pushed, your stack pointer is 8x4 bytes lower than it had been before you called the function. Keep that in mind – calling a function modifies what SP is before the function even begins executing.

Let us now assume we are in the PendSV interrupt. It is our job to:

5. Put R4 through R11 onto the stack
6. If you haven't already, now is a really, really good time to save the thread's stack pointer
7. Switch PSP so that it points to the new thread's stack pointer – now when we manipulate the memory stored there we are using the previously stored value from the new thread
8. Pop R11 through R4 (notice the reverse order)
9. Return from the PendSV handler. The microcontroller will pop the rest of the registers out for you.

Unfortunately, there are a few caveats:

1. When you are in the PendSV_Handler the CPU is also in Handler mode, so MSP is what's being used as the stack. Therefore, you shouldn't use PUSH and POP because that will push and pop onto MSP's stack. 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. In my sample solution I will use a C function and show you how to call it from assembly.

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, and I stored it in a global thread variable (see below). This meant that my thread's data structure stored the current SP minus an appropriate offset. 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. This is where the "frustration" part of the title of this book comes in!

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. It's a very good idea to make this a uint32_t, but uint8_t can be used if you would prefer (slower) byte-level stack access[25]

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. I recommend taking a staged approach – start with the minimum, get it working, then add more things as it becomes important. This isn't a great way to do it, but anything more complicated would rely on you understanding everything in this book before you started designing your OS, and the whole point of this book is to understand how to design an OS!

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, or your initial context switches will not work.

For the following, let's presume that we have a pointer, sp, which points to the current location on the thread's 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;
  ```
- Wait…why did we decrement sp? Remember we need to ensure that the stack is 8-byte aligned, so we decrement before we store something. If you do not do this step then you will potentially break the 8-byte alignment and either have to fix it in *very* weird ways later, or it won't work in the first place. Note that 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; //threadFunction is a function
  pointer
  ```
- 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

[25] The ARM Cortex M is a 32-bit microcontroller, meaning its address bus is 32 bits long. Therefore, to address a byte, you must first load the closest 32-bit address that contains it, then bit shift. The processor does this for you, but the extra steps take more CPU power, hence it is slower

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.

Fundamental Design Considerations

It's time to start making decisions. Before you can implement a context switch, you need something to switch between. So before you start writing code, let's consider the following questions:

1. How are threads represented in your OS?
2. What data structures will you use to activate/deactivate/store threads?
3. How will you context switch?
4. How do you initialize your OS and start the first thread?

There are many ways to go about answering these. In this section I'm going to give you a few ideas, but they are by no means the only options.

Representing a single thread

As I stated above, a thread requires at least a stack pointer and a function pointer. Your stack pointer is probably going to be a pointer to a uint32_t, but what about your function pointer? We can go back to the definition of RTX5's threads for a clue about a good design. Thread functions are void functions that take a single void pointer as an argument in RTX5. This gives several benefits:

1. Since it is void, the thread never has to return, and if it does it doesn't have to return anything useful
2. By taking in a void pointer, we are giving future programmers the freedom to pass in whatever arguments they want. The downside is the amount of work that the user has to do. The user must create a location in memory, store their data there, and then pass in the memory address of the start of their memory region. Then they need to extract the data in their thread function.

However, you might consider whether you want your threads to return a value. Returning a value to the kernel might be a good idea if, for instance, we want to indicate an error has occurred, or perhaps that the thread has completed and can be removed from the scheduling queue. However, that would require you to do a bit more heavy lifting when writing your scheduling algorithm.

Adding a void pointer as an argument is probably a good idea but again your design may consider adding something else. Don't pass in nothing – it's better to have the argument and not use it rather than need it and not have it – but if you, for example, need your thread to know the location of its thread local storage or something you may consider adding in additional input arguments to your thread. Now, in this book I am ignoring the input values, but if you are wondering how you access them – you put them into the part of the stack that stores R0 when you are setting up your initial stack (that is, when you initialize your thread the first time). If you do that, when the function gets called you can just access the input argument pointer right away. See Appendix A for why. If you want to ignore this that is fine, it's totally optional.

Since your threads are more than just function pointers, it is logical, therefore, to store thread data inside of structs. For this chapter, perhaps something like the following will do:

```
typedef struct thread_struct{
      uint32_t* threadStack;
      void (*threadFunc)(void* args);
}timThread;
```

As you can imagine this isn't your only option. Perhaps parallel arrays are the answer, one array for the stack pointers and one for the function pointers. The benefit of that is that it automatically answers the question: "what data structures will you use to activate/deactivate/store threads?", with the downside being that if you want additional things in your threads you are going to need more arrays.

The key here is to be intentional in your design choices. Think about what you're comfortable with and what you think will scale well. Remember, these are choices that are going to persist for the next three lab projects!

Storing threads

Your OS will need to access multiple threads and switch between them. How do you intend to do that? This depends on your choices above. If you are already storing things in arrays, you're probably done. But if you are using structs, now is the time to think about how your OS will internally store the threads.

There are several considerations here, but perhaps the most obvious will be whether you want to dynamically load threads.

To dynamically load, or not to dynamically load

An OS that dynamically loads threads does not know how many threads it has at compile-time. This is harder to do but, in some instances, it may make sense. For example, if you are programming a game, then each enemy will likely get its own thread. As enemies are created and destroyed, it may make sense to create and destroy the thread that controls them. The benefit is that if you do this well your OS will use only the minimum amount of memory at a given time, with the downside being that you need to carefully manage that memory and incur potential overhead costs.

In dynamically loaded environments you will likely want to add more stuff to your thread, but that will depend on how you choose to implement the dynamic loading. If you use something like a linked list[26]

[26] That will be a HUGE overhead in terms of searching for threads…

your threads will need next and maybe prev pointers. If it's a dynamic array[27] perhaps the thread needs to know where it is in the array. Who knows? You. That's the point.

On the other hand, if you choose to statically load your threads, this means that the OS *does* know how much memory it needs at compile time. You can simplify this even further by choosing a maximum number of threads that will ever run – say 8 – and then use an array of a set size to load your threads. Alternatively, you could have a #define statement indicating how many threads the user program will use. Setting a maximum limit will limit you and make it slightly harder to port your code – you will need to change the maximum for a new chip, whether bigger or smaller. Allowing the programmer to choose this instead will save memory if fewer than the maximum threads are being used, but adds another task to the programmer, who may forget or change the setting incorrectly.

Waiting and running

We are working with a single core chip, so the maximum number of threads that can be running at a given time is 1. The rest of the threads need to go into a waiting state. How this is handled depends on how functional you want your eventual OS to be.

If you are using only a single array to store your threads, then it may be sufficient to store the index of the currently running thread. Any thread not addressed by that index variable is therefore waiting. This is a good idea for simplicity, but it does limit you a bit. Specifically, it is not possible to have a third state. This third state, which will be relevant to you in upcoming lab projects, typically is called "blocked". A blocked thread isn't just waiting its turn, it's waiting for something to happen before it can ask for its turn to run. This is commonly done when two threads require access to the same resource in memory but only one thread can use it at a time. The OS blocks one thread while the other is using the resource, then switches them.

You can put off your decision about how to represent running and not running threads for now (I certainly will), but keep it in the back of our mind.

How to initialize your RTOS

Now you need to think about what the application programmer has to do in order to get a multithreaded application to run. It is typical to do this in a three-step process, all of which are done by the programmer in their own code:

1. Initialize the kernel, usually by calling a kernel_initialize function. This function's purpose is to change various chip settings (for example, setting the PendSV interrupt to its weakest possible state, as we did in the previous lab project), and initialize important memory.
2. Create your threads, usually by calling a create_thread type function. This function's purpose is to initialize any thread-specific memory (for instance, the initial stack settings like I talked about above), then add the new thread to whatever data structures you are using to store and run threads. This function should also perform error checking
3. Start the kernel, usually by calling a kernel_start type function. This function's purpose is to initialize anything that the first thread needs before it gets going (or that the kernel needs before the first thread runs), then switch to thread mode, switch SP to PSP, and finally start the

[27] That will be a HUGE overhead when allocating a new array…

first thread. The easiest way to do that is to call the scheduling function, but exactly how you start threads is a consideration for your OS

Lab Project 2: Multithreading

Note: the major part of this project is the design of your scaffold. The context switch, while important, is not actually critical and requires some knowledge of assembly. For that reason I have provided you with sample code that you can use for the context switching part. You may feel free to use this code directly (replace the contents of your pendSV handler assembly function with it) and just go with it, or you may choose to write your own from scratch. In the addendum to this lab project below, I've also explained every single line of that code. Please do not spend hours trying to write your own context switching code. That not the point of the lab!

Organizing your code

In the previous lab project we put the definitions for a couple of registers directly into _kernelCore.h. You may find that both the kernel code and the thread code require access to certain constants or other information. For this reason, I recommend writing a single "osDefs.h" header that contains useful definitions. For example, I have made my own osDefs.h that stores the typedef of the thread struct so that both the kernel and the thread code know what threads are and how they are stored. I also have all of my global OS defines in there, including the register definitions, stack sizes, and various thread-related constants. This way, both _kernelCore and _threadCore can include osDefs and they don't need to do something awkward where they attempt to include each other. This is the approach taken in RTX as well. You are free to ignore this advice, but you will need to think harder about how to resolve or avoid these double-dependencies if you do.

Part 1: Design

Begin by unambiguously answering the following questions. In your writeup you will be required to submit design documentation.

1. How are threads represented in your OS?
2. What data structures will you use to activate/deactivate/store threads?
3. How will you context switch? – You are free to use my context-switching code as an example, but it is unlikely to work for your OS right out of the box. You're going to need to tailor it to how your stuff works
4. How do you initialize your OS and start the first thread? – You are free to use the ideas I have described above for this, but you will need to explain the exact steps that your specific OS has to take. Everyone's will likely be subtly different

Part 2: Implementation

Implement your design. Think about good ways to organize your code, what should go into which file etc. I have provided a sample context switcher assembly file, which I will describe below. You may choose to use it or write your own. There are lots of resources on the Internet for how to do a context switch, including an interesting one written by someone named Miros (and who has the absolute best OS name ever, which is MiROS). You are free to use any code you find as a *reference*, but you must still implement it yourself. Any time you use code that is from an outside source you must cite it. Note that this means you are free to collaborate between groups, but you are not free to just download and submit someone else's code.

Part 3: Testing

Prove that your context switching code works by running at least two separate threads. These threads must clearly indicate which one is running. Printf statements like "in thread 1!" are just fine.

Lab Project 2: Submission Guidelines

1. Submit your design documentation that clearly outlines how this stage of your OS is designed. It should be written so that it is detailed enough that someone who is familiar with the board and OS design in general should be able to implement it. This means you need to clearly define any files, functions, constants, variables, and structural considerations (for example, how many arrays, why, etc). Describe your thread implementation. You do not need to spend time describing the context switcher, since I wrote that, but you must mention that it exists where appropriate. This document should not be longer than 10 pages, and I expect many of you will be able to do it in far fewer pages than that. (40 marks)
2. Submit your code, saved as a pdf, for review. It will be reviewed for style and good programming practices. (30 marks)
3. Submit evidence that your testing is successful. This is not for marks, but we will give any code submitted without evidence a 0.
4. The remaining 30% of the lab is for the lab quiz

Addendum: How does the sample context switching code work?

My context switching code is based on what we did in the previous lab project. I'll let you go back to that project to understand the various directives like AREA and the like.

The first unfamiliar line might be: `EXTERN task_switch`. In that line I am telling the linker that I have written another function called task_switch and stored it elsewhere. This is actually just a C function that, currently, just sets PSP appropriately. It looks like this:

```
int task_switch(void){
      //set the new PSP
      __set_PSP((uint32_t)osThreads[osCurrentTask].taskStack);
      return 1; //You are free to use this return value in your
                //assembly eventually. It will be placed in r0, so be sure to
                //access it before overwriting r0
}
```

The reason I do it this way is because I want to use the built-in __set_PSP function rather than having to do this in assembly. Primarily this is so that I don't need to access the array that stores my threads in assembly.

After that there is nothing new until we get into the body of the function.

```
MRS r0,PSP
```

This line loads the current value of PSP into r0. Since we are in Handler mode the stack pointer being used is MSP. I'd prefer to just use PUSH (and POP) instructions, but if I do that I'm going to push the stuff onto MSP instead, which means we can never recover it once another task switch happens. So I'm loading r0 with PSP's value and then treating r0 like it was PSP.

```
STMDB r0!,{r4-r11}
```

This line uses the STore Multiple Decrement Before (STMDB) instruction. This is essentially PUSH but not using the stack. I need to be careful here, because the "Decrement Before" thing means it decrements the address in r0 before doing any writes. This is important because PSP currently points to valid data, not an empty space, so I need to decrement to get to the next empty memory location.

```
BL task_switch
```

Then I branch to the task_switch function. Once we return from that, I reload r0 with the new value of PSP, and make sure that my interrupt return value is placed into LR:

```
MRS r0,PSP
MOV LR,#0xFFFFFFFD
```

Now we start the process in reverse. First I load the new PSP into r0, then I chose to move the "return from interrupt into threading mode" constant into LR at this point. I'm going to need to do that eventually, so I figured I'd do it first.

```
LDMIA r0!,{r4-r11}
```

This is the literal opposite of STMDB. It means "LoaD Multiple Increment After". It loads multiple registers (in this case r4-r11), and it increments after each load (the reverse of decrementing before each store). LDMIA sorts out the order of loading as well so that it completely reverses what we did in the storage step.

```
MSR PSP,r0
```

This is the opposite of loading PSP into r0 (note that this instruction is MSR, not MRS). This loads the location of r0 into PSP. Again, we have to do this because, in Handler mode, we cannot use PUSH and POP. Since r0 stores where the stack should be now, we are putting its address back into PSP

```
BX LR
```

Returns us from the interrupt and back to thread mode.

Stack Alignment

When we want to save the stack pointer location, PSP stores the value of the next thread's stack pointer. This is where that alignment thing is so important: I am reloading it from the thread, which means I had to store it correctly. I chose to store it just before triggering the PendSV interrupt. At that point, the actual PSP was 16*4 bytes higher than it is supposed to be when I restore my stack, because I haven't done any pushes yet. So when I stored my thread's stack I had to account for that by subtracting 16*4 bytes from the value that was stored.

If you find that aligning your stack is hard, the debugger is your friend. What the exact alignment offset is will depend on how threads are stored in your OS and when the context switch happens. 16 is the most common value if you are saving the thread stacks before you enter the context switching interrupt.

If you are already in an interrupt when you save the thread stack the offset is likely going to be 8, since function calls (including interrupts) push 8 registers onto the stack for you.

Chapter 5: Managing Time

Introduction

At the end of the last chapter our RTOS could switch between multiple tasks. It is now a fully-fledged OS, but it is not a *real time* OS. In fact, we've ignored any discussion about timing so far, which is an interesting choice for a course about a time-based system! Let's remedy that.

The OS so far can perform what is known as co-operative multitasking. In a co-operative scheme each thread has to yield control whenever it is done doing something. If a thread fails to yield then it will run forever and no other threads can run. Co-operative multitasking is an important step in the creation of an RTOS because:

1. It's a lot easier to start with co-operative multitasking and then extend it
2. There are times, even with more advanced multitasking, when a thread has to tell the OS that it needs to wait. For example, if it is waiting on an input, it should still be able to yield and let other threads go ahead without blocking them.

This all means that we are going to build our next multitasking system around what we've already done. We aren't going to throw anything out.

Pre-emptive multitasking

In pre-emptive multitasking the OS is doing a lot more work than in a co-operative multitasking system. The basic idea is that a timer-based interrupt is used to periodically jump back into an OS function from any running thread. The OS then evaluates any scheduling constraints and chooses which thread gets to run next. How this choice is performed is a topic for the next chapter, so we are going to stick with the method we've used so far – just keep cycling through the tasks. The focus of this chapter is on how to get the timer-based task switching working in the first place.

The Grand Vision

At this point we need to think about broad architectural choices in our RTOS. You've already designed your threads, and you very likely have an array or some other data structure that stores multiple threads, only of one of which can run at a time. Let's expand this model a bit. We are going to think about the state a thread can be in. This state model we are going to create will be built upon in the next chapters as well.

Thread states

Our threads must be in one of the following states:

- *Running*, if it is the thread whose code is currently being executed. There is at most one such thread on a single-core processor in this state
- *Sleeping,* if it is done what it has to do and needs to wait a specified amount of time before it can run again

- *Active*, if it is done running and is ready to run whenever the OS can give it time to do so

There is one other state, “blocked”, which we will implement later, and a final state, “dead”, which enables dynamic loading/unloading and which is beyond the scope of this course.

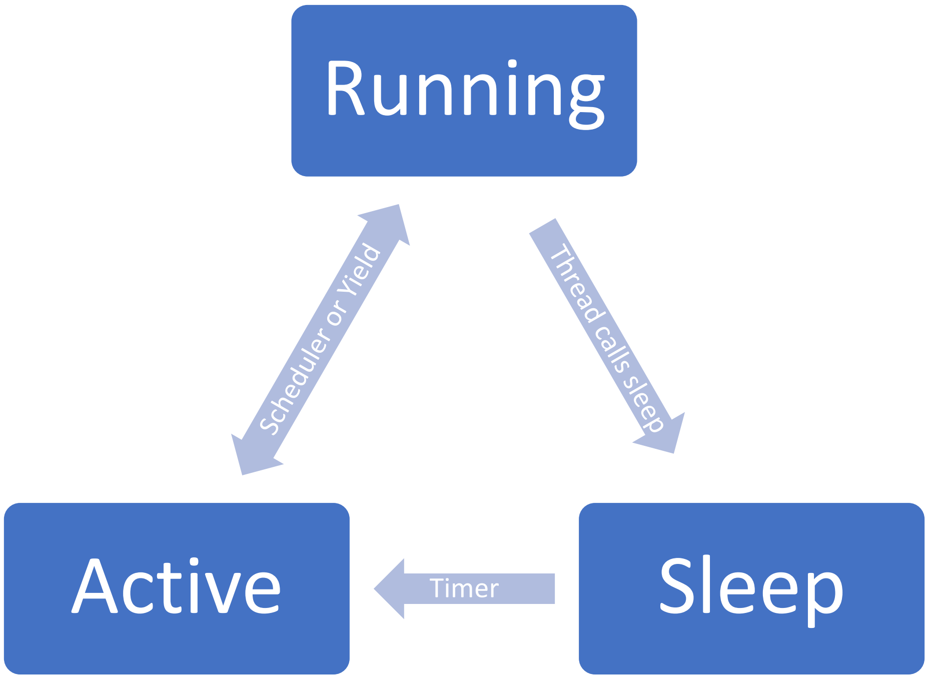
A thread may transition from Running to Active in one of two ways:

1. The OS determines that its time to run has exceeded. The OS then pre-emptively switches it out, even if it was going to yield at some later time
2. The thread is finished what it must do, and it yields

In either case, the OS then determines which task to run next. It does this using its scheduler, which is an algorithm that chooses what to run and for how long.

The Sleep state transitions a thread into a state where it must be “woken up” after a set period of time. This means that it is placed into a separate memory area and has an associated timer. When that timer is up, the thread moves into the Active state (note: not the Running state since some other thread may be running at the time). The OS is responsible for moving the thread into the Active state, but the thread is responsible for entering the Sleep state.

This implies that our OS should be organized as follows:



This gives us an idea of the data structures we need:

1. A pointer or some other way to access and run the single running thread
2. An array or some other data structure of threads waiting their turn (perhaps you want to store this in the thread structs themselves?)
3. An array or some other data structure of threads that are sleeping, along with some way to keep track of when each thread must wake (perhaps you want to store this in the thread structs, too?)

This also implies that we need to figure out a way to time things. Enter the SysTick timer.

The SysTick Timer

We are trying to make an OS that requires as little modification as possible when we move to a new chip. There isn't really much reason for this other than to model good practice – our code should be generic so that if the chip's internals change, only a minor change is required for our code to work. Many Cortex chips have several timers on board, but these timers are specific to the chip you buy. The one that isn't is called SysTick. It is a timer that must be present on all ARM Cortex M processors. As such, it makes sense to use this timer as our core clock.

SysTick is a 24-bit timer, which limits how many ticks it can work for. If you are finding that SysTick isn't working correctly, check to see that the timer value you set up isn't too big for a 24-bit number.

To set up SysTick in Keil uVision 5 you must do two things:

1. Configure the timer, usually in main, using the function `SysTick_Config(ticks);`
    - The value of "ticks" is the number of processor ticks that the timer counts for before triggering an interrupt. Keil uVision has a special macro, `SystemCoreClock`, that tells you how many ticks per second the CPU is running at. Therefore, to run the SysTick interrupt once per millisecond you would do this:

      ```
      SysTick_Config(SystemCoreClock/1000);
      ```

      Note that you will likely want to run your OS at a core frequency of around 1ms, then perhaps run your own OS-timer to change that frequency. This allows you to run several OS timers at once, all of which use this core frequency. It does mean that you cannot increase the frequency from the core clock, but honestly running faster than 1ms will put a lot of load onto the CPU as it will be nearly constantly interrupting itself
2. Write a SysTick handler function. This can be written mostly in C if you use PendSV to do the actual context switch (see below). In Keil uVision 5, this function must have exactly the function header:

    ```
    void SysTick_Handler(void)
    ```

    See below for what this function must do.

Using SysTick to Context Switch

The nuts-and-bolts of the context switch using the timer is not much different from what we've already done. We use the built-in timer, SysTick, which counts down to zero and triggers its interrupt with a period that we specify. Once the interrupt triggers, we can replicate many of the same steps we used to context switch before – push the important registers, swap PSP, and pop the registers back into place.

Since it is so similar to how the context-switch works using PendSV, it makes some sense to just use the PendSV interrupt itself to handle the context switch. This might look something like this:

1. The timer counts down and SysTick's handler function gets called
2. The handler function somehow determines which thread runs next and sets up the necessary data to do so. This may include a call to a scheduler function, but it's up to your design

3. The handler function pends a PendSV interrupt, then exits. It is PendSV (and the code you've already written to context switch) that does the real context switching

Ok, but now what? If the handler function exits, how can we be sure that the PendSV interrupt runs next? The Cortex has a functionality called "tail-chaining" that lets us do some very advanced things with interrupts. Here's how it works:

If we are already in an Interrupt Service Routine (ISR) and another interrupt triggers, the chip evaluates the priority of the two interrupts. If the currently running interrupt is of a higher priority then the chip will wait until it is completed until triggering the second interrupt. However, this happens immediately after the first ISR returns[28]. The word "immediately" here has a very special meaning. It is in fact so special that I am going to put it below in larger, bold font, because if you ignore it your context switching will stop working:

In a tail-chained interrupt, the processor does not push the 8 hardware-saved registers onto the stack again. They are already there because of the first interrupt.

So what? Well, let's think about how we did the context-switch using PendSV alone. We had to save the PSP and, when we restored it, we had to put it back accounting for the fact that we had pushed it 16x4 bytes lower than when we triggered the context switch – the 8 hardware-saved registers plus the 8 we have to save in a context switch. However, when the interrupt is tail-chained we do *not* have to account for the extra hardware-saved registers. Therefore, you will likely need to restore your PSP to a different offset than you restored it using PendSV alone.

This is a big deal. It means:

1. If you can context switch using PendSV then you are 80% of the way to context switching using timers
2. If you keep the stack aligned as though it was a pure PendSV interrupt you will almost surely not succeed!

Personally, I have to offset my stack pointer by 16x4 bytes in a PendSV interrupt but only 8x4 bytes using SysTick[29].

[28] If the new interrupt has a higher priority then the currently running ISR gets interrupted. This is not how we are going to set ourselves up in this course, but it is important to know that such interrupt nesting is possible. And yes, that's why it is called the *nested* vector interrupt controller – interrupts can interrupt each other in a process called "nesting" that depends on the priority of each interrupt.

[29] Figuring that out took *hours* because I had foolishly read about then promptly forgot what tail-chaining is.

The Scheduler

In this chapter we are going to use a very basic scheduler – a round-robin[30] (RR) scheduler. In an RR scheduler each thread is assigned a time slice – a set amount of time that it may run for – and allowed to run for at most that long. The next thread is then run, and threads are chosen in a circular fashion. This is almost the same as what we did in the previous lab project – we cycled through each thread, but the only way to change threads was for another one to call a yielding function.

I recommend writing a separate scheduler function at this point, even if you didn't in the previous lab project. This scheduler function's sole purpose is to choose which thread to run next. There are two reasons for this:

1. In future labs you will be writing a more advanced scheduler, and it will be a simple matter of overwriting this function without needing to change anything
2. In this lab you will need to use the scheduler for both co-operative and pre-emptive multitasking, so it will be easier to just call one function for both

Running the scheduler works as follows:

1. Something indicates to the OS that a context switch needs to happen. This may be a thread (yielding or sleeping) or a timer interrupt that has determined that this thread has had enough time to run
2. The scheduler gets called and whichever thread is to run next is determined
3. The context switch happens and the new thread runs

Note that it is not the scheduler's job to actually do the context switch. That is handled separately.

The Great Re-Naming

A bit dramatic, but long ago we wrote a function called "osSched". It handled both the choosing of a new thread and the context switch itself. Now that we have two separate things that can call the scheduler, I am going to rename the osSched function to osYield, which is more in line with how most RTOS's name and use that function. Its purpose is to initiate a co-operative context switch, but it relies on the scheduler to set up the thread to switch to. If you use the solution files for the next labs, you'll see this re-naming.

Fundamental Design Considerations

The whole point of this chapter is to get your OS to a point where it can run both co-operative tasks and pre-emptively scheduled tasks. The key word there is "both" – it should be possible for our threads to yield if we want, but also to be pre-emptively switched out to allow other tasks to run. In addition, we want threads to be able to sleep – to wait for a set amount of time before being allowed to run again. Sleep is an extremely important function for periodic tasks or tasks where waiting a specified time is

[30] If you're curious, "robin" is actually a corruption of the original French term, "ruban", or ribbon. It refers to the act of signing a document with the signatures in a circle, so that it would not be clear who was "first" and therefore all signatories were of equal importance. If you weren't wondering you're lying.

important (for instance, to lock out a keypad because the wrong password has been entered too many times).

Continue your design by thinking about the following questions:

1. How will you store the state of your threads?
2. How will you distinguish between a sleeping thread and a thread that is just waiting its turn?
3. How do you distinguish between a thread that yields and a thread that has to be pre-empted?
4. How do you store and manage your threads in their various states?
5. How do periodic tasks work?

Thread state

You will likely be modifying the thread struct that you created in the last lab project. So:

- What is the thread's "state"?
- How is it represented?
- What needs to be added to the thread struct to enable state-based decisions to be made?

Storing threads

Design question 2-4 above are just part of one big question: how do you store threads in your OS? Let's consider some possibilities.

First, we need some way to know which thread is running. Since we can run only one thread at a time, we need some way to indicate that one of them is "the" running thread. In the previous lab project you likely already solved this problem, but now is a good time to re-visit the design choices you made. If you store your threads in an array, it may make sense to store the array index of the currently running thread. Otherwise, you might consider storing a single thread as "the" running thread and then copying important information (probably just the thread's stack pointer) into it to do a context switch.

Next, you need to think about how to differentiate between a thread that can be scheduled and one that can't. A thread that cannot be scheduled is one that is sleeping – we have not yet implemented a way to completely remove a thread from scheduling. Threads that are sleeping also need to be woken up, which means you will need to think of:

1. What are the conditions that cause a thread to sleep? (Is it just a function call? Do you have some kind of "periodicThread" data type you are using? Both? Something else?
2. What makes a thread wake up?
3. How does the OS know that it can wake a thread?
4. How does a thread move from the sleep state to the active state?

Finally, you will need to think of what happens if a thread yields. The key thing to understand is that even if a thread will *eventually* call a function to yield, it may still be pre-emptively switched out. A thread that yields will do so when it has determined that it has no further need to run. However, the OS may have a reason to make it yield early (for one thing, what if that thread was supposed to yield but for whatever reason it just never did?) Therefore, you have to design your system so that a thread that yields on purpose and a thread that yields because its time slice ran out are both handled seamlessly. In fact, it is entirely possible for the OS to force a thread to yield at exactly the point in its execution that it was about to yield anyway. The OS doesn't care – the thread ran out of time, so it yields, even if the very next instruction it runs when it gets another time slice is to just go back and yield again.

Periodic tasks

The final consideration is what to do with periodic tasks. A periodic task is just a task that sleeps every time it has to yield, and it sleeps for the same amount each time. But here is where the problem lies: if a periodic task's period is up, does it take priority over the RR scheduler? What happens if two periodic tasks are supposed to run at the same time?

We will be diving into these problems in much more detail in the next chapter, but for now it's a good idea to consider them. You may assume in this chapter that a periodic task is simply put into the active pool once its period is up, or you may decide to think about a more sophisticated way to handle that case. We'll accept anything that works for this lab project.

It is a very good idea at this time to create some kind of running timer in your OS. This timer's purpose is to determine how much time has passed since some events have occurred, so that you can keep track of when to run various tasks. A single timer may be sufficient, but I have personally found that giving each thread a separate timer has helped a lot. This timer is just an integer that it set to a value of time I want the thread to run or sleep for. In my SysTick handler function I decrement it, and if it reaches zero I take appropriate action. This is only one possible way to solve this problem, but the point is your OS has to know what time it is!

Pitfall: Race Conditions

Now that we are building a truly multitasking OS we are going to come face-to-face with a set of problems called "race conditions". A race condition happens whenever the behaviour of a system depends on the timing of events that it cannot control. Essentially, the system "races" to finish what it's doing before something happens to prevent it from doing so. We are going to focus on a single class of race conditions here – attempting to access the same resource at the same time. This will serve as an introduction to the topic but be aware – avoiding and mitigating race conditions is an active area of research!

Let's consider the two methods we will develop for context switching: a thread that yields (or sleeps, which is just yielding for a set period) initiates a context switch on its own. If it does not control the SysTick timer, what happens if SysTick interrupts the in-progress context switch? Remember that we've previously set the PendSV interrupt to have the weakest priority, so SysTick will pre-empt it. So now what? Unless we are very careful it is likely that the switch will result in a hard fault. Perhaps the registers will be pushed onto the wrong stack, or the PSP will be set to the wrong offset.

This type of error is extremely hard to debug because each component works just fine separately and, since calls to yield happen at essentially random times, you may see the error only in very specific situations. What is particularly fun is that since these errors happen as the result of timed sequences of instructions, the bugs may seem to illogically disappear when you add things that are seemingly irrelevant. As an example, when I was developing the canonical solution to this section, I put a printf statement into my context switcher to see why I was hard faulting. This printf statement delayed the execution of the context switch just long enough that the problem didn't occur. Using the debugger caused similar problems[31].

Sometimes, trying to solve a problem the obvious way leads to worse problems, so let's get something out of the way now. The most obvious solution would be to disable interrupts every time you enter your yield function, then re-enable them just as you are about to do the PendSV context switch. This should at least reduce the chances that SysTick interrupts something critical. There are two issues, though, only one of which we can fix:

1. If we disable SysTick then we throw the OS's timing off. If the OS is trying to measure time and wake things from sleep, then not triggering the interrupt means we lose track. We can fix this problem below
2. Eventually we have to do an interrupt to do a context switch. You simply can't do it from thread mode. So even if we prevent SysTick from interrupting the part of yield that doesn't depend on PendSV, we still need to allow interrupts to trigger during a PendSV ISR. In this chapter we will simply not be able to solve this problem and must accept that our OS will have a race condition built in at this point. We'll fix it properly in the next chapter

Also, just setting PendSV to have the higher priority won't work – yet – because our yield function calls PendSV only after setting up various stacks and calling the scheduler, and getting interrupted while doing that will still cause the problem.

In the next chapter we're going to solve this problem in a much safer way than here, but this chapter's focus is on getting the timer working. I'm therefore going to defer the discussion of how to properly organize this to the next chapter and instead introduce another key concept that will be helpful later on anyway.

A Taste of Mutual Exclusion

Mutual Exclusion, or Mutex, is a key concept in the protection of resources from race conditions. If we have multiple concurrent components, each of which must access the same resource, we can create a way for these components to signal when they are using the resource. All other components that want to use the resource check this signal and only access the resource if the signal indicates that they can.

There are many sophisticated ways to create a mutex, but I suggest using a simple global Boolean variable for now. The procedure is:

1. Whenever yield is called, the Boolean is immediately set to false inside of yield
2. Yield does everything it needs to except triggering the PendSV interrupt. This includes running the scheduler and any other tasks your yield function has to do
3. Yield then sets the Boolean to true – we aren't going to return here, so we have no choice but to do this now. This means that problem #2 above may still happen, but it will happen far less
4. Yield triggers the PendSV interrupt

[31] This is a very entertaining situation known as a "Heisenbug", which is a bug whose behaviour is altered when you attempt to study it. It is named after the Heisenberg uncertainty principle and is one of the few puns I have carried around with me since undergrad in the hopes I could use it in a professional context. For this, reader, I thank you.

In the SysTick handler, then, only initiate a context switch if that Boolean is true. When it is false we are in the middle of doing something with yield, and should not be interrupted.

I strongly recommend that you get the SysTick based context switch working first, then test your OS thoroughly. Once it starts failing randomly, come back to this section and implement the changes.

Pitfall: The Idle Thread

Another issue you will need to consider is: what happens if no thread can run? This can happen either because the user didn't program any threads (and this is especially a problem if you allow threads to be dynamically loaded) or because all of the threads are sleeping. Either way the problem is the same – the CPU is always doing something, so if you have no threads to run you're in trouble.

The simplest solution to this problem is to create a thread whose sole purpose is to loop, do nothing, and context switch. Such threads are present in many commercial RTOS's[32] and their existence is hidden from the programmer. This means you need to figure out a way to run this thread when appropriate.

The reasons that it is a challenge, albeit a slight one, are:

1. If this thread is running constantly, then it takes up time slices for no reason. Your scheduler should be aware enough to only schedule this thread when there are no other options
2. This thread takes up memory. You should ensure that its existence doesn't unduly affect other threads. For example, don't give it a giant stack frame
3. This thread has to reside somewhere, and if you put it into user-accessible space that means that your user can run one fewer threads than they want. It's best to place it in its own special area and initialize it/start it with the kernel itself. The user should never know it exists

Lab Project 3: Timing

In this lab project you should focus on functionality, not necessarily efficiency. You must show your OS can work with the following cases:

1. Run three threads concurrently, one of which yields, one of which sleeps, and one of which simply loops without yielding or sleeping that is forced to context switch by a timer. What exactly each thread does is up to you
2. Run at least two threads that both sleep for at least one second. Their sleep times cannot be multiples of each other (easy solution: use a prime number of seconds to sleep each one). The Idle thread should be running whenever both threads are sleeping. What exactly each thread does is up to you

Lab Project 3: Submission Guidelines

1. Submit your code, saved as a pdf, to Crowdmark. You will be evaluated on consistency of style (10 marks), usefulness of comments (10 marks), and functionality (20 marks)

[32] I believe that the apostrophe is wrong, so it is supposed to be RTOSs, but that looks weird. I shall choose instead to go with something that *sounds* weird. I make zero apologies for this.

2. Separately, submit a brief report. The report should include:
    a. A detailed design document outlining the state of your OS at this point. If you used the Lab Project 2 solution code as a starting point you will still need to explain the various components that I made as well as what you did
    b. Evidence that your OS is capable of handling both of the above required cases
3. The remaining 30% of the lab grade is allocated to the lab quiz

Where are we now?

Wow! At this point you have a very basic OS up and running. We can't quite claim it's an RTOS yet – the timer code is likely too imprecise for that – but it can run tasks and pre-emptively multitask. Our next step is to re-evaluate how our OS works with timing constraints. We are going to solve the problem of two interrupts trying to context switch at the same time, and we are going to make an advanced scheduler that respects deadlines.

Chapter 6: System Calls

In the previous chapter we created an OS that could switch tasks based on timing, yielding, and sleeping. However, we had a few kludgy[33] hacks that we had to implement just to get it working. In this chapter we are going to address those hacks first and develop a far more general and powerful model of interacting with the kernel – the system call. Once our OS can handle system calls, we are ready to tackle the problem of advanced timing.

Note: Lab Project 4 lives in chapter 7. This chapter is a guided prelab assignment. You need to modify your OS according to what is said in this chapter or chapter 7's Lab Project is going to be much, much harder.

The Concept of a System Call

Every OS has a problem – the kernel has certain functionality (like context switching) that the threads must access. We can't just write this functionality into each thread, because:

1. That eliminates the purpose of the OS, which is literally to provide this functionality
2. Certain things, like context switching, affect multiple threads, so our thread code would be very complex if we were to implement it like this

The OS has to have this functionality and it must provide a way for threads to access it. However, we're similarly stuck – how can the thread locate and call a function that isn't compiled into it?

The answer is the system call. System calls are just functions, and often they are just interrupts[34]. The special thing, though, is that they are always accessed in the exact same way by every thread. Usually this means either that we keep the functions at a very specific and known location in memory, or we rely on the chip's interrupt functionality to trigger them. We've already been using PendSV, which is a sort of system call, and we triggered it via an interrupt. PendSV will still be used in our OS, but it is going to be subordinate to another type of call known as SVC. SVC and PendSV are almost identical except:

1. SVC runs as soon as it is triggered, assuming that no higher priority interrupts are pending. This contrasts to PendSV, which ran only when all other interrupts have ended
2. SVC allows us, through careful manipulation of the stack, to send in arguments. This is huge because it allows us to distinguish between different types of system calls. For instance, we

[33] You're studying to become an engineer, so you need to know the word "kludge" (typically pronounced "kloodge"). It refers to a solution that is not only hastily put together but with so many corners cut it's shocking that it worked at all. Kludges abound in modern technology and are usually temporary, but not always. For example, one of my colleagues at the University of Waterloo didn't like that his computer went into sleep mode every 5 minutes of inactivity, causing him to have to log in again every time he walked out of his office. Unfortunately, he didn't have the admin rights to change it. Instead, he wrote a small script that ran every 4 minutes and had the effect of moving the mouse cursor by 1 pixel, which the operating system registered as activity and reset the sleep timer. Problem solved!

[34] In modern PC operating systems on the x86 family of processors, system calls are triggered via the INT or SYSENTER instructions, but honestly, they work almost identically to what we are about to do in our ARM-based RTOS

might have one system call to initiate a sleep state and another to simply yield, while a third handles mutually exclusive resource access.

SVC is powerful and very useful, and we are going to slightly re-organize our OS so that SVC is in charge.

The Internals of an SVC Call

If our system calls only did one thing we might as well stick to what we already have and just use a Boolean to protect the various interrupts from each other. What we need is a way to encode more than one behaviour into our system calls so that we can add more as our OS grows more complex. To do this we need to set up our framework to enable it. The key is the ARM assembly instruction that triggers SVC in the first place. It is:

```
SVC #IMMEDIATE
```

Where IMMEDIATE is a number from 0 to 255 and the number sign, #, is not optional. For instance, to trigger system call 4, we would write:

```
SVC #4
```

To trigger this in a simple C function, we would write:

```
void triggerCallFour(void)
{
      __ASM("SVC #4");
}
```

Once this assembly instruction gets called, the chip will immediately look in the vector table (remember that?) for the SVC_Handler function's address. We are going to write this in assembly. In fact, I am just going to tell you what it needs to be because it is very obscure. I'll explain every line below. A good idea is to write this in your existing assembly file that contains PendSV_Handler so you don't have to set up the various AREA directives in the file, but you may choose to write a new file. The assembly code to begin an SVC call is:

```
SVC_Handler
            EXTERN SVC_Handler_Main
            TST LR,#4
            ITE EQ
            MRSEQ r0, MSP
            MRSNE r0, PSP

            B SVC_Handler_Main

            END ;Include this if this is the end of your file, otherwise
            ;don't
```

First, we define an external function called SVC_Handler_Main. We are going to write most of the actual system call framework in C, but we can't do that until we set ourselves up properly.

Next, we test the value stored in LR. Remember that, since this is an interrupt, LR will contain a magic constant. Specifically, it will tell the chip whether it was called from thread mode or from handler mode, and whether PSP or MSP was being used at the time. The reason this is important is because when we provide the immediate value to the SVC instruction (like #4, above), it gets put onto the stack. We want to recover that value, so we need to know which stack it was put on. TST will TeST the value of LR, then ITE is an If-Then-Else construct that will load R0 either with MSP or PSP depending on the result of the test. Finally, we jump to our new C function.

Now, to get the value of the system call's immediate out in our C function we must write that function very specifically. These are the first few lines:

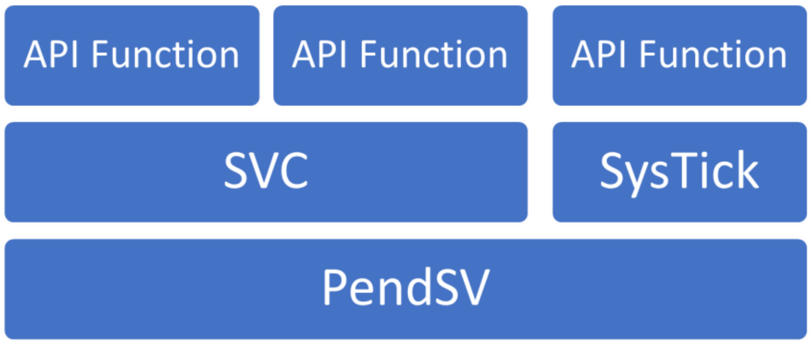
```
void SVC_Handler_Main(uint32_t *svc_args)
{
	char call = ((char*)svc_args[6])[-2];

	//Now your system call stuff looks at the value of "call" and does what

	//ever it needs to based on that information

	...

}
```

Huh? It's sort of best not to question this too much, but what's going on is we are "cleverly"[35] accessing the entire stack frame that was pushed onto the correct stack when the SVC interrupt happened. Whenever we call a function from assembly, the first input argument is always put into R0. In our SVC_Handler we put a pointer to the stack that contained the appropriate immediate value. However, there are other things on that stack that get pushed during an interrupt call by the hardware, so we need to cleverly search that stack for our immediate. It turns out that it is 2 characters behind the actual value of the stack pointer. If that hurts your brain, you are free to completely ignore it, copy what I did as the first few lines, and just start writing your system call framework now.

An Extensible System Call Framework

Wow! Are you now clear on why we didn't put this into the previous lab project? It takes a while to understand!

Our framework will depend on three levels of components:



The user will program their threads using various API functions, for example osYield or osSleep. Those functions are all that the user is exposed to. At the second level we have our two main interrupt methods. SVC will directly interact with the API through calls like osYield, while SysTick will interact with

[35] That is, using the most obfuscated code possible, because ARM isn't great at documentation

the API when we get to periodic timers and other things for which timing is relevant. Finally, both SVC and SysTick will use our existing PendSV context switcher.

Our API functions then are going to be very simple – they are just wrappers for the assembly instruction that triggers the SVC and supplies the appropriate interrupt number. What each number corresponds to is entirely up to you, but it is probably a good idea to just number them in order. In my OS I am going to use 0 for yield and go from there. Interestingly, some of the API calls may themselves be wrappers for other API calls. This is more up to you than it is up to me, but as an example, my osSleep function first sets up the sleep timer on the thread then calls osYield to do the context switch. Therefore I don't need a specific Sleep system call, my SVC framework is simpler, and I re-use components at the potential expense of an extra function call.

The Great Re-Organization

So far, the functionality for our various interrupts have been written directly into the functions themselves, so osYield for example would contain all of the scheduling code and then trigger PendSV itself. It now makes sense to pull that code out and put it into SVC (or have SVC call helper functions if it's getting too big). I call this the Great Re-Organization, and if you didn't do anything horribly complicated in the last lab project you will only need to modify osYield and osSleep. Here is an example of what my osYield looks like now:

```
void osYield(void)
{
	//Trigger the SVC right away.
	__ASM("SVC #0");
}
```

That's it! I moved all of the stuff that osYield did into the SVC_Handler_Main, including the PendSV trigger. So this is what my SVC_Handler_Main looks like:

```
void SVC_Handler_Main(uint32_t *svc_args)
{
	char call = ((char*)svc_args[6])[-2];
	if(call == YIELD_SWITCH)
	{
		if(osCurrentTask >= 0)
		{
			osThreads[osCurrentTask].status = WAITING;
			osThreads[osCurrentTask].timeout = RR_TIMEOUT;

			//Notice the offset of 8 now, not 16, since I'm already in
			//an interrupt
			osThreads[osCurrentTask].taskStack =
			(uint32_t*)(__get_PSP() - 8*4); //we are about to push a
			//bunch of things
		}

		//Run the scheduler
			scheduler();

		//Pend a context switch
			_ICSR |= 1<<28;
			__asm("isb");
	}
}
```

This gives me the freedom to add the sleep system call and any others I want. One thing this will be especially useful for is in the final lab project when we implement mutexes properly, rather than just Booleans being accessed by multiple interrupts. These mutexes will be accessed via system calls, and the various switching functionality will be handled in SVC.

Interrupt Priorities

We now have three interrupts that are available to us: PendSV, SysTick, and SVC. We need to re-organize our interrupt priorities, and then we can finally get rid of that pesky Boolean.

Recall that the entire point of the Boolean variable was to protect a call to yield from being pre-empted by a SysTick interrupt since the problem with SysTick is that it runs no matter what. SVC and PendSV are both triggered by user code, but SysTick triggers when it's ready. We also want to ensure that PendSV doesn't pre-empt SVC, since PendSV's purpose is to wait until the processor is ready before attempting a context switch. Therefore, we are going to organize our interrupt priorities as follows:

1. SVC will have the strongest priority of the three. We will set its priority to 0xFD. This way it cannot be interrupted by SysTick or PendSV. Note that this is still a fairly weak priority that allows us lots of room for stronger interrupts. We won't be using any other interrupts intentionally[36] in this course, but you should be aware that you have lots of wiggle room to add more interrupts. If you feel that SVC needs to be stronger at any point, you can increase its priority
2. PendSV will have the second strongest priority. We will set its priority to 0xFE. This way it cannot be interrupted by SysTick. Since PendSV is being used by both SysTick and SVC, it is not possible for it to interrupt the other two, so we don't have to worry about race conditions when using it
3. SysTick will have the weakest priority, 0xFF. This way it cannot interrupt a context switch already taking place. This eliminates the need for our Boolean mutex from the previous chapter

SVC's priority lives in the SHPR2 register, bits 24-31. SHPR2's address is 0xE000ED1C. Define a macro for it like we did for SHPR3 so long ago. Assuming you did this the same way I did, your kernel initialization function needs to contain the following three lines:

```
//set the priority of PendSV to almost the weakest
SHPR3 |= 0xFE << 16;
SHPR3 |= 0xFFU << 24; //Set the priority of SysTick to be the weakest

SHPR2 |= 0xFDU << 24; //Set the priority of SVC the be the strongest
```

Prelab Project

Implement everything above and test it using your Lab Project 3 main file. Don't forget to remove our Boolean kludge that protected the yield function! If everything went well you should notice absolutely no difference between this implementation and the one you did for lab project 3.

[36] For instance, the hard fault interrupt always has the strongest priority no matter what, but we never call the hardfault hander intentionally.

Do not proceed to the next chapter until you are done that part, or you’re going to have a bad time.

Chapter 7: Advanced Timing

We are about to do something rather remarkable. We are going to modify our OS so that we use a new scheduling algorithm called Earliest Deadline First (EDF). Once this modification is done, we will be able to run various types of tasks just by setting up our threads correctly. Specifically, we can run:

- Periodic tasks that require real-time scheduling
- Co-operative tasks that yield to each other
- Tasks that sleep for random amounts of time, and may never be periodic

The key thing to understand is that we are going to create a single system that a user can use to run these three types of tasks or any combination of them, but all tasks will fundamentally be the same. What this means is that we are not going to create a separate struct for periodic tasks that is somehow different from the struct we used for co-operative tasks. Instead, we are going to ensure that a single thread struct, and any associated functions and data structures, are flexible enough to enable us to create any type of thread we want.

The Pre-Emptive Earliest Deadline First Scheduler

To understand what we're about to do, we need to take a brief step into the world of the pre-emptive EDF scheduler. Note: at this point in your course, I am assuming that you have seen the EDF scheduler in lecture. As such, this is going to be a quick refresher rather than an in-depth analysis. Also, this is the point where we must admit that our OS is not going to be as efficient as it can be. I am going to use an implementation that is simple but inefficient. I will outline why that is as we talk about what we're doing.

The Deadline

The whole point of an RTOS is the RT part – we need to make sure that tasks happen with known timing[37], and that if we need a task to happen by time X then we need to either be sure that it will or have proof that it won't (indicating we need to redesign something). However, so far, our tasks have just been threads that either yield on their own or run continuously until a timer-based interrupt tells them it's time to wait. Worse, we've scheduled our tasks based on the order in which they were created, and completely ignored whether one task has to run first.

Now, there are many ways to specify this priority, but we are going to use the concept of a thread's deadline so that tasks get their priority set for them automatically – a thread has to finish running by a certain time, it's deadline, or the system has failed. If the scheduler must choose a task to run, it has to choose the task with the earliest deadline first, hence "Earliest Deadline First" scheduling! Whenever

[37] Let's not get into the complexity of "hard" vs "soft" real-time. We aren't going to do enough proving to guarantee hard real-time performance for our OS, but we are going to be as strict as possible.

two tasks have the same deadline, we say there is a tie, and in this course we are going to break ties arbitrarily[38].

Now, the above paragraph glosses over some very important implementation details! Let's think about what a functional EDF scheduling system has to look like.

The flow of EDF

When an EDF scheduling system is working, threads are in one of three states – they are either ready to run, running (this is only one thread that has the CPU at this time), or not ready[39]. If a thread is not ready to run we say it is "waiting". Threads that are waiting are made ready to run once some time has elapsed that is specific to the thread. For instance, a thread that has a period of 1s will be moved from waiting to ready once per second. Because of this our OS needs some mechanism to determine when a thread is done waiting and gets to move into the ready state. In addition, it is possible that a thread that is done waiting has an earlier deadline than the currently running thread, so it needs to run first. Finally, some threads may run continuously and never yield. If this is the case, we can handle it in one of two ways:

1. Assume that this is the desired behaviour and that this thread is supposed to use 100% of the CPU. If this is the case then no other threads can run. If we assume that this is desired, we leave it up to the user to design better code. My OS will do it this way
2. Give each thread a maximum time that it is allowed to run for, and pre-emptively switch it out after that time is exceeded. If we do it this way then our OS will be sensitive to what this time slice is. RTX5 does it this way[40]

Either way, in normal operation of our OS threads will yield, and there are multiple ways for that to happen:

1. A thread voluntarily yields
2. (Maybe, depending on design) a thread exceeds its maximum runtime and is pre-emptively forced to yield
3. A waiting thread has finished its waiting period and has an earlier deadline than the currently running thread, so the currently running thread is pre-emptively forced to yield anyway

All of this means that we need:

1. A method by which threads can voluntarily yield (we have this already!)
2. A way to specify the maximum runtime of a thread (optional)
3. A way to specify how long a thread has been waiting for, and to swap it out if needed

[38] It doesn't have to be arbitrary. For example, we might assign each thread a secondary priority that is only relevant when there is a tie, or perhaps assign some other rule. For our purposes, we are going to assume that two threads with the same deadline have the same priority, and which to choose over the other is not relevant.
[39] We are ignoring a fourth state that will be important next chapter. A thread may also be "blocked", meaning that it is waiting for some external event to happen before it can be made ready to run. For instance, a thread may be blocked waiting on user input. That is an extra layer that will need a bit of careful thought, so I am leaving it for later.
[40] And, as an aside, it really shouldn't. What if the programmer wanted that one thread to use 100% of the CPU? Perhaps it is safety critical and must have full control.

For points 2 and 3, I suggest adding to your thread struct. Personally, I added two uint32_ts: one for the thread's period and one to count down from that value to zero. If the count reaches zero when the thread is ready it means that the thread has missed its deadline. If the count reaches zero when the thread is waiting it means that the thread must be moved to the ready state and its count down timer is reset to the thread's period.

Co-operation, Sleeping, and Mixed Tasks

In the introduction to this chapter, I mentioned that our new EDF scheduler can be used to emulate a number of different modes of operation for our RTOS. I'd like to show you how that might be done.

Emulating a co-operative multitasking system

Our very first multitasking system was a co-operative, round-robin system. We can emulate this by setting the period of all threads to the same value. Typically, this value needs to be small, perhaps only a few milliseconds, but it depends on the user – if they want to run things once per second or once per millisecond that's up to them. The threads also must yield like they did in our co-operative system.

Why this works is that a yielding thread is put into a waiting state and other threads are allowed to run. Since all threads have the same deadline and ties are broken arbitrarily, the order that the threads run in will be fixed[41]. Therefore, we run the first thread, it yields, and then the second thread runs, yields and so on until all threads have run. If no threads have emerged from the waiting state, we run the idle thread until the waiting period has elapsed. Logically, the first thread to run will have its waiting period elapse first, so it runs first again and so on down the line of threads.

Sleep

An important thing for threads to do is sleep – to wait for a specified amount of time no matter what. Sleep is a more general concept that periodicity, since a periodic thread can be thought of as a thread that sleeps for a set period, reawakens, then sleeps again for the same amount of time. A sleeping thread does not have to be periodic. It might sleep for ten seconds one time and ten minutes another[42].

Interestingly, we've already implemented sleep in the previous lab project and for some of you there won't be much more to do – the sleep function might set the waiting timer of the thread and then force a context switch. For others, you may need to modify how sleep works if you wrote it in a way that makes it harder to implement with the new EDF scheduler.

An interesting idea is to use your sleep function as the only thing that ever causes context switches. Since every thread has to have a waiting period, it may be useful for you to just always use the same function to set that period and context switch. Personally, I chose not to do it this way in case in the future I choose to extend my scheduler to handle multiple use cases.

Mixing it all together

What happens if you have a task that sleeps, a task that yields, and task that is supposed to run periodically all in the same system? If you design it right, it should just work. Throughout this chapter I

[41] Unless you do something untoward and choose to use a random number generator or something, which is…just not how it's done.

[42] This is a lot like how some smartphones will lock you out if you enter the wrong password too many times. The first lockout might be only a minute, then the times get progressively longer.

have been encouraging you to think about all the ways a task can run as special cases of a single, general case – a thread that runs, then waits for some time, and becomes ready again.

- From the perspective of a co-operating task, the "wait for some time" just means that other tasks need the processor. A co-operative thread has no sense of how many other tasks are running and therefore when it runs next is not up to it (well, not up to the programmer…).
- We've already noticed that periodic tasks are special cases of sleeping tasks with a set period

The big test of your OS is to try to run the same threading code as you did in the previous lab project but change only the scheduler. Other than some minor specific things, like the exact order that the tasks run in, you shouldn't notice a difference.

Fundamental Design Considerations

As with the previous labs, you now need to make some decisions. At the very least you are going to need to figure out:

1. How do you represent time, especially time to deadlines?
2. How do you store waiting and ready threads?

Representing Time

You are almost certainly using SysTick, but there are other timers on board the chip. Regardless you are going to encounter a problem. If everything on the chip is a uint32_t, and your timers are going to run fairly fast (probably around 1ms at least, faster if you want), you are going to overflow your timers a lot if you decide to count *up*. This is one of the reasons that timers in microcontrollers count down then reset themselves and count down again.

Your threads need to know how long they have until they are ready again, and your scheduler uses this information to choose the next thread to run. Therefore, if you implement a timer in your threads you likely want to make it count down to zero and, when it is zero, do something important. Don't try to keep a running timer for your OS so your threads run at 1 second, 2 seconds and so on, since eventually that will overflow!

Storing Threads

If you've been using my implementation of the OS you'll know that I am using an array to store my threads. I am choosing to do this for simplicity, but it does limit me somewhat. In fact, in the next section I'm going to discuss these design decisions, because now is the time to talk about efficiency. Some of you may be using a linked list.

The questions you need to answer are: how many data structures do you use to store threads, and what are they? Is it just one, with threads marked as "ready", "running", and "waiting"? Is it two – one for ready/running threads and one for waiting threads? Are you using an array? A linked list?

One interesting choice would be to use a priority queue. Priority queues are designed for this exact purpose – the top element is the element that has highest priority. Some priority queues also sort their elements based on priority, but this is actually not required. You may consider implementing a priority queue, which is a very specific type of data structure, as this would be the most efficient implementation if you intend to scale up to a larger chip with more memory.

Wait, What about WCET?

The Worst-Case Execution Time (WCET) of a thread is often discussed when we bring up EDF scheduling. Should we consider it in our implementation? The answer is no. WCET is a way for a user to analyze their system to ensure that CPU utilization is below 100% and, almost the same thing, to ensure that the EDF scheduler can meet all of the deadlines. However, the pre-emptive EDF scheduler doesn't have to know about WCET because it is *pre-emptive*. Let's consider something:

Say we are currently running task A and task B emerges from sleep. Task B has an earlier deadline, so we switch to B. Now, what happens if task A's WCET is so high that, by the time B is done and A is switched back in, there is no time left to finish task A? This actually means that the schedule is not possible anyway.

To understand why, let's assume that A and B are the only tasks running. For concreteness, say that A and B have the same period, T, but they started at slightly different times so they don't tie, that the WCET of A, $C_A = 0.62T$, and the WCET of B, $C_B = 0.41T$. In this specific case, processor utilization is above 100% and we're stuck!

The interesting (and hard to prove) thing is that in this simple two-task system there is no way to have two running tasks in an EDF scheduling mode that simultaneously cannot meet their deadlines and keep the CPU utilization below 100% (ignoring context switching). Either CPU utilization is too high or one of the task's deadlines has to be far enough away to allow everything to run. This result extends to multiple tasks, so WCET is the user's problem, not the OS designer's.

The practical upshot of this is that you, the OS designer, should not worry about what the WCET of each task is, and it definitely isn't something that the tasks themselves need to know.

A note on Efficiency and Good Design

A long time ago I mentioned that your OS won't be efficient. I also included "definitely not for profit" in the subtitle of this document. But why?

I chose to focus our discussion on the fundamental components that your OS should have to be minimally functional. I am using an array of tasks, rather than a dynamic data structure, for instance, because I do not consider dynamic task loading to be part of that minimally functional set.

This is the first time where efficiency considerations may have serious consequences for your OS design. I have chosen to keep my OS simple, and therefore I have chosen to not use a priority queue for my tasks. This has the consequence that, every time my timer ticks, I modify or at least examine the timer on every thread. So, if I have N threads my timer tick is always running at least a O(N) algorithm. For a small number of threads this isn't important, and my OS is meant to teach rather than shock you with its awesomeness[43].

[43] I'm already awesome enough, no need for further shock.

If I were to implement my OS using a priority queue, however, I could speed up this part considerably. If everything is done properly a priority queue can reduce the algorithm to O(ln(N)). That's amazing! If I had 100 tasks that means that instead of 100 comparisons and modifications, I'm doing at most 5. Implementing a priority queue wouldn't actually be that hard, but it would require knowledge of data structures that is beyond the scope of this course.

There are other choices I have made in the design of my OS that strictly violate good design. For example, I really should have organized my code so that there was a core library that was specific to the chip, and then a broader OS library that calls those chip-specific functions. This minimizes the difficulty of porting my OS.

So, as we near the end of our OS journey, I want to make sure that I mention these things. I encourage you to take each part of this OS that you made and simplify it, increase its efficiency, and really push what it can do. It's an interesting project. However, it is also well beyond the scope of our course!

Why am I not saying this is a hard real-time system?

For all we know our OS is operating in hard real time, but the issue is that we *don't* know. First, whether yours is or not depends on how you built the OS, but second we've done no analysis to figure this out. For instance, we know that we can tell the OS that a task has to sleep for some number of clock ticks, but does it really? How long does context switching take? How accurate is our system clock? Is there any other overhead that eats into that time? Does it wake sooner, or later, or right on time?

We've also made our OS so that our threads *emerge from sleep* periodically. They do not actually run periodically. As long as their WCET is low enough and our constraints are lax enough this isn't a problem, but it's something to be aware of.

We have created a system where we can be reasonably sure that timing is respected. I would not put these OS's onto a safety critical system, nor would I use them on something where exact timing matters (like audio processing). However, a system like a robot, which uses feedback control anyway, or a low-importance system like a GUI, can use our OS just fine. If you want to call it a hard real time system, though, you've got a lot of work to do to prove it.

Lab Project 4: Pre-Emptive EDF Scheduling

Implement a pre-emptive EDF scheduling system. Since this new scheduler should be able to work with the same cases as in the previous lab project, you may notice that the cases below are similar to what you had to do in Lab project 3. You must show your OS can work with the following cases:

1. Run three threads concurrently. One must have a period of 256Hz, one with a period of 100Hz, and one with a period of 12Hz. What exactly each thread does is up to you
2. Run three threads concurrently, one of which yields, one of which sleeps, and one of which is periodic with a frequency at least 200Hz. What exactly each thread does is up to you
3. Run at least two threads that both sleep for at least one second. Their sleep times cannot be multiples of each other. The Idle thread (or equivalent in your OS) should be running whenever both threads are sleeping. What exactly each thread does is up to you

Lab Project 4: Submission Guidelines

1. Submit your code, saved as a pdf, to Crowdmark. You will be evaluated on consistency of style (10 marks), usefulness of comments (10 marks), and functionality (20 marks)
2. Separately, submit a brief report. The report should include:
    a. A detailed design document outlining the state of your OS at this point. If you used the Lab Project 3 solution code as a starting point you will still need to explain the various components that I made as well as what you did
    b. Evidence that your OS is capable of handling all of the above required cases
3. The remaining 30% of the lab grade is allocated to the lab quiz

Chapter 8: Mutual Exclusion

This is the final topic we are going to cover in this course, and it's an important one. In single threaded programs we don't have a problem with memory protection – the CPU can run only one instruction at a time, and the instructions run sequentially. If we have a global variable, for example, it is not possible for a single thread to attempt to simultaneously read it and write to it. All that changes when we introduce multithreaded, concurrent programs.

Let us consider a program in which thread A writes to a global array of sensor values and thread B reads them and filters them. If the system is pre-emptively multitasking, it is possible that thread A is interrupted while it is in the middle of writing to the array. If thread B now runs, what happens? It accesses some data that is new and some that is old, and potentially this can cause problems. We need a framework for these two threads to tell each other that they need the resource, and then to lock that resource while a particular thread is using it. This is known as mutual exclusion – once one thread is using the resource it excludes the others entirely from accessing it. We usually create a struct called a "mutex" that enables this functionality.

What Mutexes Do

Consider what happened when you implemented cooperative multitasking by using a yield function. This function told the kernel that the thread was done a critical computation and can be put to sleep until there is time for it to run again. The kernel then activates the next thread via its scheduler, and that new thread similarly completed a calculation and then yielded control.

Mutexes take this one step further. When a thread simply yields it is giving control to the kernel to schedule it when the kernel feels it is necessary. When we use a mutex the thread is asking the kernel to let it yield and never wake it up until the resource it is waiting on becomes available. Theoretically this may never happen if the other threads using the resource fail to release them[44].

It is important to understand that the mutex struct does not actually contain any information regarding how the thread is supposed to access the resource. It is simply a way to communicate via threads that a resource is available. It is still up to the underlying software to actually access the resource. An analogy may be helpful here. Consider a traffic light at an intersection. The light simply tells drivers that they should or should not proceed into the intersection. It is the mutex. Whenever one direction gets a green light the other gets a red light and there is never a time when both directions get a green light. However, drivers are physically free to ignore this. A driver can run through a red light or stop at a green one, and the lights themselves do not prevent or enable them to do so. They are simply signals that tell the drivers what their expected behaviour is.

In some systems, an important use of mutexes is when our threads have different priority. In a priority-based system, a higher priority task is able to take control of a mutex if a lower priority task has access. If mutexes were unable to do this then the lower priority task could block the higher priority task. This is known as "priority inversion", where the lower priority task gets higher priority by accident. This must

[44] We will discuss how to address this problem later in this chapter.

be handled by the kernel – threads are unable to communicate between themselves to negotiate this – and is probably one of the biggest advantages to using mutexes. In our RTOS, however, priority is inherited from deadlines and the running task always has the highest priority. For this reason we are going to ignore priority inversion in our RTOS **but we are going to be very aware that we are doing so**. Don't think that we have somehow circumvented the problem of priority inversion. Rather, we are living with the simplifications that we made in previous labs because to do otherwise would require a massive amount of work.

Using Mutexes

To use a mutex we need at least two threads. The kernel is aware of the mutex but has no idea which resource is being controlled by it. It isn't relevant. A thread calls a function to acquire the mutex, usually called something like osMutexAcquire. This initiates a call to the kernel where one of two things happens:

1. If no thread currently has acquired the mutex then the thread asking to acquire the mutex is given the access and the mutex becomes unavailable. No other threads may access it.
2. If some other thread already has acquired the mutex, then the thread asking to acquire it is put into a "waiting" queue. In the simplest case this is a FIFO data structure, where the thread that has been waiting the longest gets access to the mutex next. Note that mutexes in a system using an EDF scheduler introduce the problem that a thread may wait so long for a mutex that it misses its deadline. We are consciously ignoring that problem for this lab project.

Threads that are waiting are never reawakened unless something kicks them out of that state (for instance a timeout on the mutex or the mutex being released).

Once a thread is done with the mutex it releases it, typically by calling a function called something like "osMutexRelease". This releases the mutex **but the thread that releases it is not automatically put to sleep**. When it is time for the kernel to run a new thread, it will determine if a waiting thread can be run next, and if so it will run that thread, take it out of the waiting queue and assigning the mutex to it.

Design Decisions: What's in a Mutex?

Fundamentally a mutex is a Boolean variable controlled by the kernel, so our mutex struct requires at least a Boolean that indicates whether the resource is available or not. But that's not enough. For one thing, it is possible to have multiple mutexes in memory at a given time, so we need some way to identify the mutex being requested by each thread. Therefore, our mutex requires an ID of some sort.

Thread ownership and the waiting queue is the next design decision you have to make. Does the mutex know which thread owns it? Something has to – what happens if a thread calls osMutexRelease for a mutex it is not currently holding? Whether the kernel has some kind of data structure that contains this information or the mutex itself holds this information is up to you. Furthermore, what about the waiting queue? Is that part of the mutex or is it part of another OS-specific data structure? You will need to determine this as part of your lab project, and there is no single right answer.

The OS needs some way to keep track of mutexes much like it keeps track of threads. This can be a simple array of mutexes like we have for threads, or a dynamic data structure. Again, the choice is yours and there is no single best option.

The Waiting Queue

The reason we used an EDF scheduler in the first place was to create a scheduling algorithm that had many benefits, among them that if a thread had to run at a specific time, it either would run or the CPU would be overutilized. Mutexes can break this, and for this lab project we are just going to be OK with that. However, we are going to give threads the ability to break out of the waiting queue if the mutex isn't available within a specific amount of time.

When a thread enters a mutex's waiting queue, it should specify a timeout value. This is like the sleep value – a set amount of time that it waits until it is ready to run again. If the mutex is not available by that time the thread should be informed but woken up. We will also define a constant, typically called osWaitForever, that indicates that the thread will wait until the mutex is available no matter how long it takes.

The existence of osWaitForever means that a thread that has a deadline is no longer guaranteed to meet that deadline if it waits too long. But if that's the case, think about it this way: what if that thread simply can't do any meaningful work without the resource that is protected by the mutex? What's the point of running?

All of this means that our EDF scheduler will not be optimal if we are using mutexes. You should be aware of that, but don't think too hard about it.

The Minimal Set of Functions

Mutexes must be created, acquired, and released. Optionally they may be destroyed. You must create your mutex API that allows for at least the following functionality:

- **Create:** the mutex is created. Although it is possible to call this in a thread, it is more often called before the kernel starts. It is up to you and your design to figure out what this function needs in order to run. It is a good idea for this function to return an ID that can be used by threads to acquire the mutex. This function should allocate memory if needed, set up the mutex according to your design, and make it available for the OS to assign ownership
- **Acquire**: a thread attempts to acquire a mutex. If that is possible, the mutex must be assigned ownership and the function returns. If that is not possible the thread is placed into the waiting queue. The thread must be able to specify which mutex it wants to acquire and how long it is willing to wait. The function should likely return something useful to indicate to the thread whether the acquire was successful or whether it timed out. If the thread enters the waiting queue a context switch must occur and other threads are allowed to run.
- **Release**: a thread attempts to release a mutex. If that thread has ownership of the mutex the mutex is released and the OS is free to schedule another thread in the waiting queue. The thread releasing the mutex is not immediately put to sleep but may continue running until either a pre-emptive or co-operative context switch occurs. You must decide how you are going to schedule the next task. The simplest solution is to give the mutex to the next thread in the waiting queue then put it into the OS's ready queue. It then gets scheduled whenever the scheduler chooses it but it has the mutex. Other options are possible and are up to you

We are not going to require that you destroy mutexes, but you are free to implement that if you wish.

Lab Project 5: Mutual Exclusion

Implement mutexes in your OS as described above. Demonstrate that they work in the following two scenarios:

1. Three threads, all attempting to print their name (as in “Thread 1”, “Thread 2”, and “Thread 3”) using UART via the printf function. UART is protected by the mutex. The three threads co-operatively yield as soon as they release the mutex. No timing considerations.
2. Three threads. One thread increments a global variable, x. Another uses that global variable to set the LEDs to x % 47. The third thread sets the LEDs to 0x71. Two mutexes must be used – one to protect the global variable and one to protect the LEDs.

Lab Project 5: Submission Guidelines

1. Submit your code, saved as a pdf, to Crowdmark. You will be evaluated on consistency of style (10 marks), usefulness of comments (10 marks), and functionality (20 marks)
2. Separately, submit a brief report. The report should include:
    a. A detailed design document outlining the state of your OS at this point, including detailed documentation of how your mutex API works. If you used the Lab Project 4 solution code as a starting point you will still need to explain the various components that I made as well as what you did
    b. Evidence that your OS is capable of handling both of the above required cases
3. The remaining 30% of the lab grade is allocated to the lab quiz

Chapter 9: Wrap-up

Are you impressed yet? Your OS has many of the foundational structures in place that a commercial OS would have. I hope that you have learned a lot about computer organization and programming, RTOSs, debugging, and managing a large-scale software project. At this point I'd like to spend some time going over the design decisions we made, the simplifications that we chose to use, and how you might consider expanding your OS to get around them. This chapter is optional and serves mainly to inspire you to continue exploring this interesting topic on your own.

Threads

I encouraged you to use a static array of threads, and indeed my RTOS uses static arrays for just about everything. The reasoning for this was that we were not interested in learning about high-efficiency data structures and dynamic thread loading. Here are a few things you might consider if you want to go further with this OS:

1. Implement a dynamic load – using a static array means we only have so many threads we can use. Dynamically loading them every time you create a thread means that you are no longer stuck with this limitation. However, you will need to update your scheduler to look through whatever data structure you choose, and the OS will have to do more bookkeeping to know which threads exist and how to access them. You will also have to include memory protections, because at this point it is entirely possible for you to create so many threads that you run out of memory to store them.
2. Implement a dynamic unload – if you can create threads, why not destroy them? You can use the static array for this or a more interesting data structure. Dynamically unloading tasks means you can build projects that run in stages. Perhaps the first stage sets up sensors and tests them while the second stage uses them to complete a task?

Scheduling

The EDF scheduler is OK but we haven't done anything to prove that it actually meets deadlines. We are pretty sure that for simple enough tasks it will, but we don't really know. How long does the scheduler take to run? What about the WCET of the tasks? What about mutexes and the waiting queue?

I encourage you to take more steps to prove that your OS can meet hard real-time deadlines, and modify it if it cannot.

Memory Protection

Mutexes are a fairly simple form of memory protection that allows us to either access a resource or not. The next step up from the mutex is the semaphore – a count of how many copies of a given resource exist, and a way to keep track of who has access to a copy and who is waiting. Semaphores are a generalization of mutexes and some commercial RTOSs just create a single semaphore API and use it to emulate mutexes. You may consider doing the same thing, or writing something separately.

Another next step is to implement inter-thread communication via message queues. This further generalizes and expands on the abilities of your semaphore library. Rather than just signaling that a resource is available, threads should be able to pass messages between each other to co-ordinate their use of resources.

Use a Commercial or Open Source RTOS

Writing your own is always fun, but there comes a time when you want to explore what a real RTOS can do, how it solves the problems you struggled with, and whether it is better or worse than your own. There are many free RTOSs available online, and it is always a good idea to choose one to become familiar with.

Appendix A: The Application-Binary Interface (ABI)

Note: this section is optional but will be helpful for understanding various operations. Throughout the text at very rare times I will mention something like how to pass an argument to a C function through assembly. If you don't know what a calling convention is then it will seem like magic. As such, I'm including this section but, if you skip it until you get to a point in the text where it's needed, then come back, you'll be alright. I will assume that you have arrived at this appendix after being confused reading the main text. Therefore if I say things like "stack pointer" and "R0" I'm assuming you know what that means. If you've arrived here out of curiosity before doing at least lab project 1 you may be confused, but please do read on.

I've mentioned before that, although C is a stack-based language, we do not need a stack to use a microcontroller. The simplest way around this would be to program it purely in assembly and just avoid any stack operations at all. We could, for instance, directly program the registers we want and organize memory however we please. This begs a question: if the microcontroller doesn't require certain things to exist in order to work, who designed them?

Things we take for granted, like the stack, are rigidly defined in the ABI. The ABI is something that most programmers know nothing about – they don't care how the variables are stored, they just care that we can access them when we need them. But we are not most programmers. We are writing an operating system and doing so requires a deeper understanding. We are, for one thing, going to be writing in two languages – assembly and C – and we are going to need to manipulate memory very carefully.

ABIs are generally very dense documents written the goal of being precise and unambiguous rather than easy to read. They are not teaching tools, but if you want to dive in you can access ARM's ABIs for our processor here: https://github.com/ARM-software/abi-aa/releases . If you check out that link you will see that there are many documents that comprise the ABI. This is because the ABI has to specify many things, among them:

- How are functions called and returned from?
- How are binary files created, loaded, and run?
- How does the debugger work?
- Lots more

For someone very serious about learning the internals of how C becomes assembly and then machine code in an ARM Cortex-M, you'll need to read all of the documents. For us, though, it is sufficient to understand a few key parts of the procedure call standard (PCS). I'm going to summarize the key points below, but if you want to read the whole thing it is available here: https://github.com/ARM-software/abi-aa/releases/download/2022Q1/aapcs32.pdf .

A Tiptoe through the Arm Procedure Call Standard

There is a lot in the PCS. What concerns us most is how the stack works and how functions obtain, maintain, and return data.

The Stack

In the Arm PCS, the stack is defined to be Full, Descending. This means that the stack pointer points to valid data on the stack (in comparison to an Empty stack, where the stack pointer points to the next valid place to put data), and that the stack grows down, so that when we push two values, A and B, in that order, B has a lower address than A.

This will be crucial when we perform our first context switch because we need to know where our stack is going to end up when we store important information on it. Since we know it is Descending we know that we'll need to store an address below where we started. Since we know it is Full we know that we should always decrement the stack before trying to put something onto it.

This is not the only way to do it. A stack may be Full or Empty, and it may be Ascending or Descending. Full, Descending is quite common, though and is used in the x86 family's ABI from Intel.

Passing Arguments into a Function

Registers R0 through R3 are called the "argument" and "result" registers. If you've ever wondered how a function knows where to find its input arguments, or where to put its return value, these are the starting point for that. Each of the registers can hold a single uint32_t or smaller data type, so if you call a function like this:

```
myFunc(3,4);
```

Then the 3 goes into R0, the 4 goes into R1. These are always filled in order R0 through R3. If instead you pass an argument that is larger than a uint32_t, it gets stored in a combination of registers. So sending in a uint64_t means you use up both R0 and R1.

In fact, that code above compiles to the following assembly code:

```
MOV R0,#3
MOV R1,#4
BL myFunc
```

Practically, this means that if we want to call a C function from assembly, with an argument, we do it like I just showed you – put stuff into R0 through R3 and branch-and-link (BL) to the function. This is also helpful for you if you want to pass arguments into your thread functions. When you are setting up your threads (as we do in lab project 2), you can put a pointer into R0 and your thread function can access it when it runs the first time.

If you pass in doubles or floats the same rules apply, but now the registers are just interpreted as 4-byte memory locations. Floating point on the Cortex M3 is always software emulated and very slow – there is no native floating point hardware – so it's best avoided unless you really need it.

Now, what happens if we have more than 4 arguments, or more accurately, more than 4xsizeof(uint32_t) bytes to pass into a function? The compiler will start to put things onto the stack. The registers are used only to make access to the arguments faster during a function call, but if they are used up there is no choice but to use the stack.

This means that registers R0 through R4 are special and have to be maintained during function calls. If you were to call another function inside of an already running function, then it must store its arguments in these registers. This will overwrite what is already there, but the original calling function may not be done with them yet! That is why these registers, along with other important registers like PC, are stored for you by hardware. If they weren't then function calls would be impossible, so it is not the user's problem to figure them out.

The Variable Registers

Registers R4 through R11 are used by the compiler to speed up certain things. They are called the "Variable Registers", in that local variables are sometimes stored there. However, the registers R9 and R11 are sometimes special and up to the compiler to figure out. These registers are not guaranteed to be maintained between function calls, so they are typically used by the compiler for only very temporary calculations. These temporary calculations are assumed to be valid only between function calls, and once a new function gets called you may assume that these registers are going to be overwritten. This is why these registers are not stacked for you during function calls – they simply don't need to be, and the compiler doesn't care – but it's also why we needed to save them when we were doing a context switch. Since context switches can happen at any time between function calls, when we switch back, we should be reasonably sure that our temporary calculations will be restored until the code is done with them.

Return values

In C we can return only a single thing, and no data type is larger than 128 bytes. Although there are rare cases when this isn't the case (and yes, the PCS has a way to handle that) we aren't going to concern ourselves about that here. When we return a value from our function it gets stored in R0 through R4. This means that we can call a C function from assembly, have it return a value, then examine the registers to figure out what it was.

Who keeps track of all of this?

Almost always it is the compiler's job to do this. For instance, if a function has too many input arguments to fit in the registers, the compiler puts some onto the stack then accesses them as appropriate in the function. This is complicated and highly optimized, which is why if we choose to pass arguments into a thread, we should keep it to a single input argument of type void pointer. This allows us to not worry about what the user intends to do with their inputs – it is the user's job to manage memory and correctly access it in the threads – and we only need to worry ourselves with giving the user the ability to access that memory when necessary.

Appendix B: Desktop Operating Systems

Our RTOS and a desktop OS perform similar functions for some things but are wildly different for others. This brief appendix is going to outline some of the major differences. I am going to try to keep this as generic as possible, but whenever I need a specific example I will use Linux unless there is a good reason not to. I will limit this discussion to x86 operating systems, and most of what I say here will be valid whether the chip is an x86 or x86-64. This section is purely for your own interest and is optional.

The Boot Process

The Cortex M has a relatively simple boot process: read the vector table, extract the stack and start function addresses, then run that function. The start function typically just sets various vector table entries, configures important system hardware like the clock, and transfers control to main. It is very possible to write all of this from scratch with a bit of effort and the documentation.

A desktop PC boots very differently. First, most of what we think of as "booting" the OS is multiple stages down from the real boot. There are several interesting problems that have to be solved on a desktop PC that microcontrollers do not have. The biggest such problem is that although microcontrollers have on-chip permanent memory, so you can just look for executable code right away, desktop PCs do not. Desktop PCs have to first find and configure their hard drive (or other boot disk), then search it for the kernel. This is a huge circular problem – you need code to initialize the hard drive, but the hard drive is what you use to find code to run! Desktops solve this problem with a multistage boot sequence.

The very first thing to run, typically from a ROM like flash memory at a very specific memory address, is the manufacturer's boot code. This used to be a simple OS called BIOS (Basic Input/Output System), but it has since been superseded by UEFI (Unified Extensible Firmware Interface). UEFI does a lot. It discovers hardware, especially RAM and the hard drive[45], it finds the boot disks and sets them up to the point where the OS can start loading, and it provides information to the future stages of the boot process. Interestingly enough, UEFI can also be used by the user to configure important system settings (like boot disk search order), and it looks a lot like BIOS did, which is why most people just refer to this initial stage as the "BIOS".

In the very beginning, once UEFI has found the boot sector on the boot disk, a simple program is loaded. This program, in Linux, is part of the GRUB2 (Grand Unified Bootloader) package. I won't refer to GRUB2 much, though, because other OSs boot using similar processes but different bootloaders. This is typically called the stage-1 bootloader and it is tiny. Many modern systems pack this entire program into the first sector of the hard drive, which is usually 512B and contains other metadata. In fact, this metadata takes up so much space that the entire first stage has 442B to work with. The point of this program is to locate and load a temporary program that does quite a bit of work.

This temporary program's purpose in most operating systems is to initialize the hard drive to a point where the bigger bootloader can be loaded. Interestingly, this is usually a two-step process as well. Since modern hard drives are so large, the next stage (sometimes called "stage 1.5") was created simply to get

[45] Although modern operating systems can boot from many things, let's simplify our lives and assume that the hard drive is what we are booting from. The steps don't change.

the hard drive ready to load more stuff. This means quite a bit of setup code has to run – the hard drive has already been located and minimally configured by UEFI, but all that was done back then was to let us read sectors. There is no concept of a file system, so searching the drive is all but impossible. Stage 1.5 fixes that for us, initializes a file system, then locates and runs stage 2. Stage 2 is what finally loads the kernel.

In Stage 2 we have a file system where we can do lots of stuff – stuff that depends strongly on the OS you are trying to run. This stage's purpose is to locate the kernel and load everything it needs to get started. This can be a very long process – what services does the kernel need? What files does it need? How should RAM be set up? Are there other hardware interfaces that need to be set up? Loading files is a slow process, which is why booting a system that has been turned off can take some time. Modern operating systems have sped the process up considerably, but booting is probably the most time consuming part of starting the computer.

At this point the final part is very anti-climactic – a function, called either just "main" or "kernel_main", gets called, and then it's the OS's problem from there. This is equivalent to calling "main" in the microcontroller.

Scaffolding a Desktop OS – Getting to Running Processes

In our first lab project we developed a scaffold for our OS that focused a lot on the context switch. For an RTOS this is about as much of a scaffold as you need, but for a desktop OS there is a lot you need to do before the concept of "context" even makes sense.

Perhaps the biggest difference at this stage of the OS development process is that a desktop OS typically has to discover the hardware on which it is running. Some hobby-level OS projects don't need to do this, since the developer knows the exact configuration of their machine, but any OS worth using is able to discover and configure hardware on its own. To do this the OS relies on the hardware itself. All configurable hardware in use on a modern PC has some method of identifying itself and, if necessary, its features[46].

Although it is true that UEFI and the multi-stage bootloader has done some of this work, these programs have really just set the hardware up so that the kernel is able to customize it. As a brief example, modern hard drives communicate through one of only a few common interfaces (typically AHCI[47] or NVMe[48], but there are others) and these interfaces are stunningly powerful. For instance, you can write gigabytes of data by just telling the hard drive controller where the data starts in RAM and how big it is, then telling it to interrupt the kernel once it's done. The CPU is freed from the task of managing the process and you may not even realize that a huge file write is going on in the background. The kernel has to initialize the hardware to take full advantage of what it can do, and it can only do this once it knows exactly what kind of hardware it is talking to. The previous stages use more generic but suboptimal settings just to get things started.

[46] OK, well maybe not "all", but most hardware you're ever going to encounter

[47] Advanced Host-Controller Interface, one of the worst documented standards to ever exist

[48] Non-Volatile Memory Express, originally developed for solid state drives. Now that hard disk drives are becoming obsolete NVMe will likely supersede AHCI. One of its major advantages is that it is better documented and simpler.

To do anything meaningful, the kernel needs to load modules and drivers, and to run several of them at once. It is at this stage, much like we did in our RTOS, that the idea of "context" becomes meaningful. However, we cannot just load programs at this point. There are three things that need to be set up:

1. **The Global Descriptor Table (GDT)**: This is a table that tells the OS where it is allowed to put programs, data, and other things in memory. It is a legacy system from the early 1990s and nowadays doesn't do much, but without the GDT the x86 family of chips won't let you do the next two things, so you have to set one up no matter what
2. **The Interrupt Descriptor Table (IDT):** This is what the Cortex calls the vector table and the nested vectored interrupt table. This contains all of the interrupt function pointers, and it references the GDT (which is why you need to set up the GDT). This lets us do many things, like interrupt driven IO, but more importantly it allows us to set up interrupts for memory access. This leads us to:
3. **Virtual Memory:** Modern desktop operating systems use virtual memory. Virtual memory is a way to map real memory ("physical memory") to a virtual address space. This lets us pretend like each program is the only program that is currently running. It simplifies the problem of compiling code, accessing RAM, and it protects processes from each other. Whenever a program tries to access memory outside of what is already mapped to it, an interrupt is triggered (this is why we needed the IDT). When this happens, the OS decides if this memory access is valid (so it gives the process more memory) or invalid (so it causes a segmentation fault).

Wow! Once we have these things up and running, we are ready to run programs, which, when running, are often called "processes"[49].

Processes in a desktop OS are quite different from threads in our RTOS. A process is an entirely separately running program. It believes that it has full access to all of the computer's memory via virtual memory. It may even run multiple threads or sub-processes. Our threads were mainly independent of each other and definitely did not have any memory protection! In fact, a fun exercise is to try to access the memory of one thread from another. It is totally possible in our RTOS. It is very, very much not possible in a desktop OS unless you are doing something stunningly wrong.

Hardware Interfacing and the Runtime Environment

A desktop OS has a kernel that is usually, but not always, expanded via programs known either as "drivers" or "modules". The kernel is the absolute core of the system, developed by whoever owns the OS. The drivers and modules, though, are not always written by that same group. In fact, for very large operating systems like Linux this is not possible, and it is generally assumed that hardware manufacturers will write their own drivers. For hobby operating systems these modules may be written by a dedicated user group or by a sole developer.

What is interesting is that a modern desktop OS uses many of the same concepts that we used in microcontroller interfacing. All hardware is memory mapped[50], and you need to learn how to access the various registers by finding their memory locations. The hardware is much more complicated than on a

[49] A "program" is the code and data that can be loaded and run. A "process" is an instance of a program that is running on the CPU. You are forgiven if you mix the terms up.

[50] There are other ways to access hardware, but they are legacy and are only really used at the very start of development to get the basics going. It is possible to use memory mapped interfacing exclusively.

microcontroller but there isn't really much difference otherwise – set and clear bits to set up the hardware, write and read from known memory locations to transfer data, use interrupts if you want to free the CPU to do other tasks.

Other than hardware control, most operating systems also impose what is known as a "runtime environment". This is a group of settings, data, programs, and memory locations that a user program can rely on to exist. For example, in Linux, a user program can rely on the existence of the syscall function and can rely on it to correctly perform system calls. The user program doesn't need to know exactly how it works, but it needs to know that it exists and has defined behaviour.

Runtime environments differ from OS to OS, and therefore you can't just take a program compiled on Windows and run it natively on Linux. The various function calls built into the program simply don't exist on a different operating system. There has been some progress to create emulated runtime environments (like the WINE environment to run Windows code on Linux) but generally speaking these are incomplete and not really a substitute for running the program on the OS for which it was compiled.

The Runtime environment also includes, at least, the C standard library. The C standard library is a large collection of functions, like printf, that must all be re-written for the specific OS. For example, printf requires access to the stdin input stream and the ability to read and write to FILE objects (which are also defined in the C standard library). How the OS handles stdin and FILE objects, among many other things, depends on how the OS is set up. Therefore, every aspiring OS developer will at some point write their own cstdlib! The fun part of this is to write your own cstdlib, then use it to compile some basic programs (for example an assembler). These basic programs can be used to compile more functional programs (a C compiler, perhaps), which can then be used to start compiling open-source programs. At this point your OS is largely ready for beta testing and expansion.

Can you Write your own Desktop OS?

Yes…sort of. The subtitle of this book includes the words "definitely not for profit", and that extends to desktop OS development. Your OS will be kludgy and slow, it is unlikely someone else will want to use it, but it is possible to make it work. Feel free to check out mine, which is both poorly documented and barely functional[51]. You'll find it here: https://github.com/mstachowsky/schismx86

One of the best resources for hobby OS development is the osdev wiki, which contains the collected knowledge of hobby operating systems developers. It is mainly correct, sometimes out of date, but you will learn so much about computer organization even if you just get a simple "Hello, world" to run. You can find it here: https://wiki.osdev.org/Main_Page

osDev also maintains a forum that you need to answer a skill-testing question for, and a reddit page, which you do not. You will learn a lot from these resources as long as you don't ask something like "please give me the code for an OS" – they expect you to do a lot of the work on your own!

If you do choose to write your own desktop OS be warned that you are in for a long term project. It took me three or four months to get my hard drive to respond to me, and then another month to write a

[51] I can write to the hard drive and run programs, sort of, but not both at the same time!

simple file system layer that will eventually (as of July 2022) form the backbone of a more capable virtual file system. I spent about three weeks getting my first executable to run. There is a lot of learning, a lot of code to write, but also I find it to be a lot of fun.