Lesson FreeRTOS + LPC40xx

- LPC40xx MCU Memory Map
- FreeRTOS & Tasks
- Lab: FreeRTOS Tasks

LPC40xx MCU Memory Map

What is a Memory Map

A **memory map** is a layout of how the memory maps to some set of information. With respect to embedded systems, the memory map we are concerned about maps out where the Flash (ROM), peripherals, interrupt vector table, SRAM, etc are located in address space.

Memory mapped IO

Memory mapped IO is a means of mapping memory address space to devices external (**IO**) to the CPU, that is not memory.

For example (assuming a 32-bit system)

- Flash could be mapped to address **0x00000000** to **0x00100000** (1 Mbyte range)
- GPIO port could be located at address **0x1000000** (1 byte)
- Interrupt vector table could start from **0xFFFFFFFF** and run backwards through the memory space
- SRAM gets the rest of the usable space (provided you have enough SRAM to fill that area)

It all depends on the CPU and the system designed around it.

Port Mapped IO

Port mapped IO uses additional signals from the CPU to qualify which signals are for memory and which are for IO. On Intel products, there is a **(~M/IO)** pin that is **LOW** when selecting **MEMORY** and **HIGH** when it is selecting **IO**.

The neat thing about using port mapped IO, is that you don't need to sacrifice memory space for IO, nor do you need to decode all 32-address lines. You can limit yourself to just using 8-bits of address space, which limits you to 256 device addresses, but that may be more than enough for your purposes.

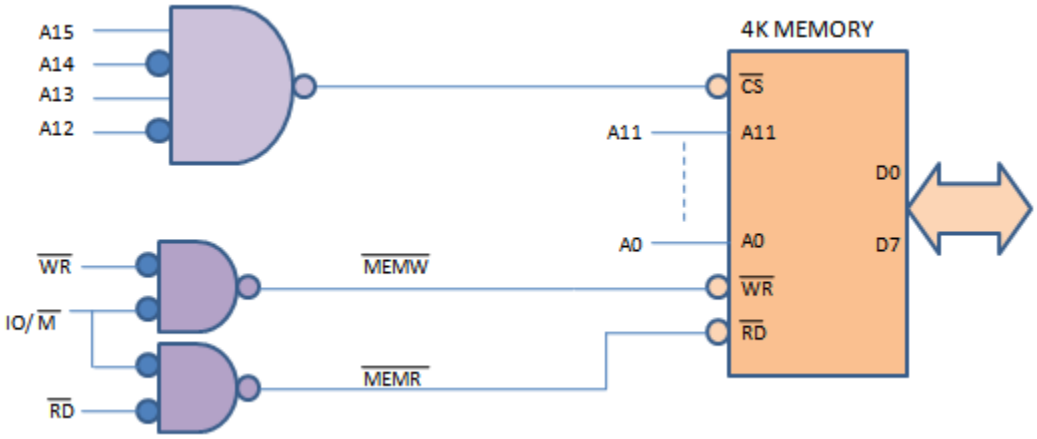


Figure 2. Address Decoding with port map

(http://www.dgtal-sysworld.co.in/2012/04/memory-intercaing-to-8085.html)

LPC40xx memory map

UM10562

Chapter 2: LPC408x/407x Memory map

Rev. 3 — 12 March 2014 User manual

2.1 Memory map and peripheral addressing

The ARM Cortex-M4 processor has a single 4 GB address space. The following table shows how this space is used on the LPC408x/407x.

Table 3. Memory usage and details

Address range	General Use	Address range details and description	
0x0000 0000 to 0x1FFF FFFF	On-chip non-volatile memory	0x0000 0000 - 0x0007 FFFF	For devices with 512 kB of flash memory.
		0x0000 0000 - 0x0003 FFFF	For devices with 256 kB of flash memory.
		0x0000 0000 - 0x0001 FFFF	For devices with 128 kB of flash memory.
	On-chip SRAM	0x1000 0000 - 0x1000 FFFF	For devices with 64 kB of Main SRAM.
		0x1000 0000 - 0x1000 7FFF	For devices with 32 kB of Main SRAM.
	Boot ROM	0x1FFF 0000 - 0x1FFF 7FFF	8 kB Boot ROM with flash services.
	Driver ROM	0x1FFF 8000 - 0x1FFF 1FFF	16 kB Driver ROM
0x2000 0000 to 0x3FFF FFFF	On-chip SRAM (typically used for peripheral data)	0x2000 0000 - 0x2000 1FFF	Peripheral SRAM - bank 0 (first 8 kB)
		0x2000 2000 - 0x2000 3FFF	Peripheral SRAM - bank 0 (second 8 kB)
		0x2000 4000 - 0x2000 7FFF	Peripheral SRAM - bank 1 (16 kB)
	AHB peripherals	0x2008 0000 - 0x200B FFFF	See Section 2.3.1 for details
	SPIFI buffer space	0x2800 0000 - 0x28FF FFFF	SPIFI memory mapped access space
0x4000 0000 to 0x7FFF FFFF	APB Peripherals	0x4000 0000 - 0x4007 FFFF	APB0 Peripherals, up to 32 peripheral blocks of 16 kB each.
		0x4008 0000 - 0x400F FFFF	APB1 Peripherals, up to 32 peripheral blocks of 16 kB each.
0x8000 0000 to 0xDFFF FFFF	Off-chip Memory via the External Memory Controller	Four static memory chip selects:	
		0x8000 0000 - 0x83FF FFFF	Static memory chip select 0 (up to 64 MB)[1]
		0x9000 0000 - 0x93FF FFFF	Static memory chip select 1 (up to 64 MB)[2]
		0x9800 0000 - 0x9BFF FFFF	Static memory chip select 2 (up to 64 MB)
		0x9C00 0000 - 0x9FFF FFFF	Static memory chip select 3 (up to 64 MB)
		Four dynamic memory chip selects:	
		0xA000 0000 - 0xAFFF FFFF	Dynamic memory chip select 0 (up to 256MB)
		0xB000 0000 - 0xBFFF FFFF	Dynamic memory chip select 1 (up to 256MB)
		0xC000 0000 - 0xCFFF FFFF	Dynamic memory chip select 2 (up to 256MB)
		0xD000 0000 - 0xDFFF FFFF	Dynamic memory chip select 3 (up to 256MB)
0xE000 0000 to 0xE00F FFFF	Cortex-M4 Private Peripheral Bus	0xE000 0000 - 0xE00F FFFF	Cortex-M4 related functions, includes the NVIC and System Tick Timer.

Figure 3. LPC40xx Memory Map

From this you can get an idea of which section of memory space is used for what. This can be found in the UM10562 LPC40xx user manual. If you take a closer look you will see that very little of the address space is actually taken up. With up to 4 billion+ address spaces (because 2^32 is a big number) to use you have a lot of free space to spread out your IO and peripherals.

Reducing the number of lines needed to decode IO

The LPC40xx chips, to reduce bus line count, make all the peripherals 32-bit word aligned. Which means you must grab 4-bytes at a time. You cannot grab a single byte (8-bits) or a half-byte (16-bits) from memory. This eliminates the 2 least significant bits of address space.

Accessing IO using Memory Map in C

Please read the following code snippet. This is runnable on your system now. Just copy and paste it into your **main.c** file.

```
/*
    The goal of this software is to set the GPIO pin P1.0 to
    low then high after some time. Pin P1.0 is connected to an LED.
    The address to set the direction for port 1 GPIOs is below:
        DIR1 = 0x20098020
    The address to set a pin in port 1 is below:
        PIN1 = 0x20098034
*/
#include <stdint.h>
volatile uint32_t * const DIR1 = (uint32_t *)(0x20098020);
volatile uint32_t * const PIN1 = (uint32_t *)(0x20098034);
int main(void)
{
    // Set 0th bit, setting Pin 0 of Port 1 to an output pin
    (*DIR1) |= (1 << 0);
    // Set 0th bit, setting Pin 0 of Port 1 to high
    (*PIN1) |= (1 << 0);
    // Loop for a while (volatile is needed!)
    for(volatile uint32_t i = 0; i < 0x01000000; i++);
    // Clear 0th bit, setting Pin 0 of Port 1 to low
    (*PIN1) &= ~(1 << 0);
    // Loop forever
    while(1);
```

```
    return 0;}
```

> ***volatile*** keyword tells the compiler not to optimize this variable out, even if it seems useless

> ***const*** keyword tells the compiler that this variable cannot be modified

> Notice **"const" placement** and how it is placed after the **uint32_t *.** This is because we want to make sure the pointer address never changes and remains constant, but the value that it references should be modifiable.

Using the LPC40xx.h

The above is nice and it works, but it's a lot of work. You have to go back to the user manual to see which addresses are for what register. There must be some better way!!

Take a look at the **lpc40xx.h** file, which It is located in the `sjtwo-c/projects/lpc40xx_freertos/lpc40xx.h`. Here you will find definitions for each peripheral memory address in the system.

Let's say you wanted to port the above code to something a bit more structured:

- Open up "**lpc40xx.h**"
- Search for "**GPIO**"
  - You will find a struct with the name **LPC_GPIO_TypeDef**.
- Now search for "**LPC_GPIO_TypeDef**" with a **#define** in the same line.
- You will see that **LPC_GPIO_TypeDef** is a pointer of these structs
  - `#define LPC_GPIO0 ((LPC_GPIO_TypeDef *) LPC_GPIO0_BASE )`
  - `#define LPC_GPIO1 ((LPC_GPIO_TypeDef *) LPC_GPIO1_BASE )`
  - `#define LPC_GPIO2 ((LPC_GPIO_TypeDef *) LPC_GPIO2_BASE )`
  - `#define LPC_GPIO3 ((LPC_GPIO_TypeDef *) LPC_GPIO3_BASE )`
  - `#define LPC_GPIO4 ((LPC_GPIO_TypeDef *) LPC_GPIO4_BASE )`
- We want to use **LPC_GPIO1** since that corresponds to the GPIO port 1.

- If you inspect **LPC_GPIO_TypeDef**, you can see the members that represent register DIR and PIN
- You can now access **DIR and PIN registers in the following way:**

```
#include "lpc40xx.h"
int main(void)
{
    // Set 0th bit, setting Pin 0 of Port 1 to an output pin
    LPC_GPIO1->DIR |= (1 << 0);
    //// Set 0th bit, setting Pin 0 of Port 1 to high
    LPC_GPIO1->PIN |= (1 << 0);
    //// Loop for a while (volatile is needed!)
    for(volatile uint32_t i = 0; i < 0x01000000; i++);
    //// Clear 0th bit, setting Pin 1.0 to low
    LPC_GPIO1->PIN &= ~(1 << 0);
    //// Loop forever
    while(1);
    return 0;}
```

At first this may get tedious, but once you get more experience, you won't open the **lpc40xx.h** file very often. This is the preferred way to access registers in this course and in industry.

> On occasions, the names of registers in the user manual are not exactly the same in this file.

?

FreeRTOS & Tasks

Introduction to FreeRTOS

Objective

To introduce what, why, when, and how to use Real Time Operating Systems (RTOS) as well as get you started using it with the sjtwo-c environment.
I would like to note that this page is mostly an aggregation of information from Wikipedia and the FreeRTOS
Website.

What is an OS?

> Operating system (OS) is system software that manages computer hardware and software resources and provides common services for computer programs. - Wikipedia

Operating systems like Linux or Windows

They have services to make communicating with Networking devices and files systems possible without having
to understand how the hardware works. Operating systems may also have a means to multitasking by allow
multiple processes to share the CPU at a time. They may also have means for allowing processes to communicate together.

What is an RTOS?

An RTOS is an operating system that meant for real time applications. They typically have fewer services such
as the following:

- **Parallel Task Scheduler**
- **Task communication** (Queues or Mailboxes)
- **Task synchronization** (Semaphores)

Why use an RTOS?

> “
> You do not need to use an RTOS to write good embedded software. At some point though, as your application grows in size or complexity, the services of an RTOS might
> become beneficial for one or more of the reasons listed below. These are not absolutes,
> but opinion. As with everything else, selecting the right tools for the job in hand is an important first step in any project.
> In brief:
>
> - **Abstract out timing information**
>
> The real time scheduler is effectively a piece of code that allows you to specify the
> timing characteristics of your application - permitting greatly simplified, smaller (and
> therefore easier to understand) application code.
>
> - **Maintainability/Extensibility**
>
> Not having the timing information within your code allows for greater maintainability

and extensibility as there will be fewer interdependencies between your software
modules. Changing one module should not effect the temporal behavior of another
module (depending on the prioritization of your tasks). The software will also be less
susceptible to changes in the hardware. For example, code can be written such that it
is temporally unaffected by a change in the processor frequency (within reasonable
limits).

- **Modularity**

  Organizing your application as a set of autonomous tasks permits more effective
  modularity. Tasks should be loosely coupled and functionally cohesive units that within
  themselves execute in a sequential manner. For example, there will be no need to
  break functions up into mini state machines to prevent them taking too long to execute
  to completion.

- **Cleaner interfaces**

  Well defined inter task communication interfaces facilitates design and team development.

- **Easier testing (in some cases)**

  Task interfaces can be exercised without the need to add instrumentation that may
  have changed the behavior of the module under test.

- **Code reuse**

Greater modularity and less module interdependencies facilitates code reuse across
projects. The tasks themselves facilitate code reuse within a project. For an example
of the latter, consider an application that receives connections from a TCP/IP stack -
the same task code can be spawned to handle each connection - one task per connection.

- **Improved efficiency?**

  Using FreeRTOS permits a task to block on events - be they temporal or external to
  the system. This means that no time is wasted polling or checking timers when there
  are actually no events that require processing. This can result in huge savings in
  processor utilization. Code only executes when it needs to. Counter to that however is
  the need to run the RTOS tick and the time taken to switch between tasks. Whether
  the saving outweighs the overhead or vice versa is dependent of the application. Most
  applications will run some form of tick anyway, so making use of this with a tick hook
  function removes any additional overhead.

- **Idle time**

It is easy to measure the processor loading when using FreeRTOS.org. Whenever the
idle task is running you know that the processor has nothing else to do. The idle task
also provides a very simple and automatic method of placing the processor into a low
power mode.

- **Flexible interrupt handling**

  Deferring the processing triggered by an interrupt to the task level permits the interrupt
  handler itself to be very short - and for interrupts to remain enabled while the task level
  processing completes. Also, processing at the task level permits flexible prioritization -
  more so than might be achievable by using the priority assigned to each peripheral by
  the hardware itself (depending on the architecture being used).

- **Mixed processing requirements**

  Simple design patterns can be used to achieve a mix of periodic, continuous and
  event driven processing within your application. In addition, hard and soft real time
  requirements can be met though the use of interrupt and task prioritisation.

- **Easier control over peripherals**

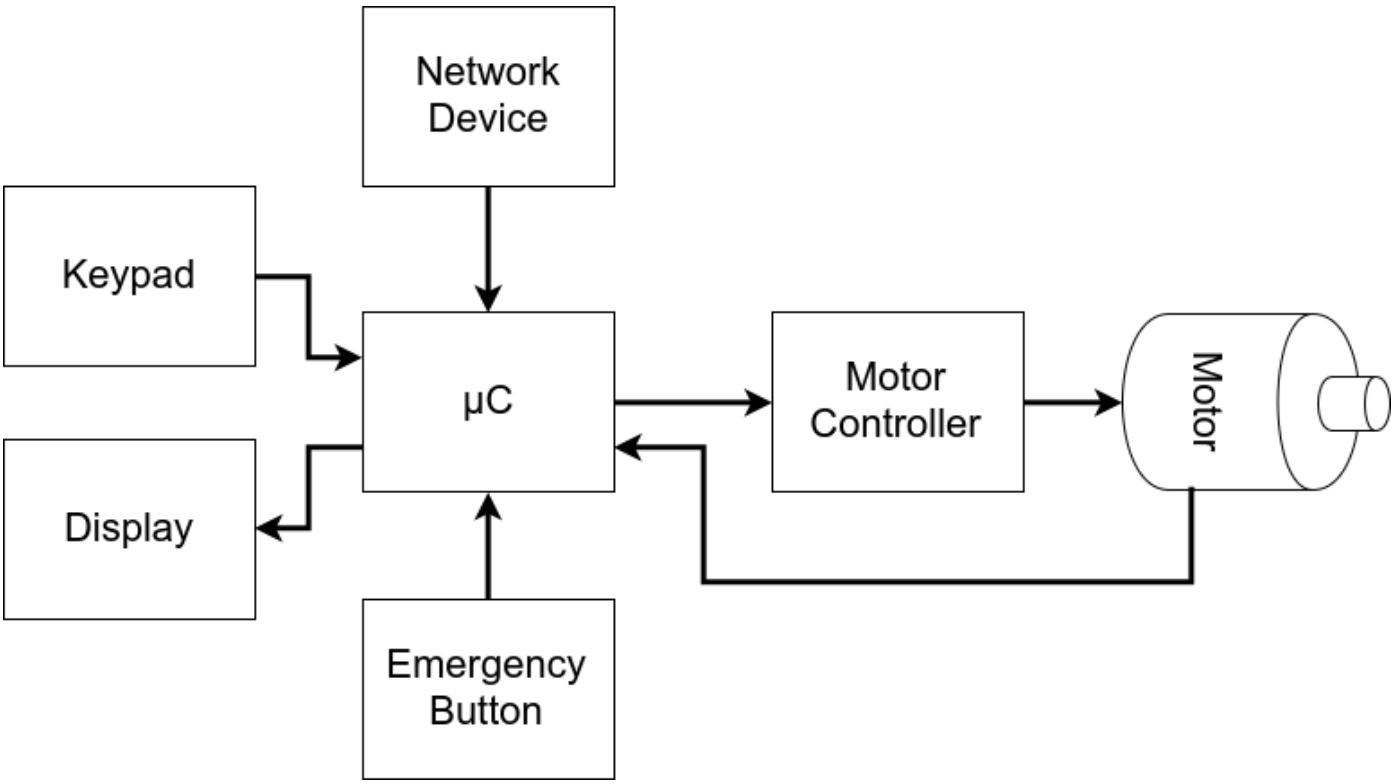
  Gatekeeper tasks facilitate serialization of access to peripherals - and provide a good
  mutual exclusion mechanism.

- Etcetera

> - FreeRTOS Website (https://www.freertos.org/FAQWhat.html)

Design Scenario

Building a controllable assembly conveyor belt



Think about the following system. Reasonable complex, right?

Without a scheduler

? Small code size.
? No reliance on third party source code.
? No RTOS RAM, ROM or processing overhead.
? Difficult to cater for complex timing requirements.
? Does not scale well without a large increase in complexity.
? Timing hard to evaluate or maintain due to the inter-dependencies between the different functions.

With a scheduler

? Simple, segmented, flexible, maintainable design with few inter-dependencies.
? Processor utilization is automatically switched from task to task on a most urgent need basis with no
explicit action required within the application source code.
? The event driven structure ensures that no CPU time is wasted polling for events that have not occurred.
Processing is only performed when there is work needing to be done.
* Power consumption can be reduced if the idle task places the processor into power save (sleep) mode,
but may also be wasted as the tick interrupt will sometimes wake the processor unnecessarily.
* The kernel functionality will use processing resources. The extent of this will depend on the chosen
kernel tick frequency.
? This solution requires a lot of tasks, each of which require their own stack, and many of which require a
queue on which events can be received. This solution therefore uses a lot of RAM.
? Frequent context switching between tasks of the same priority will waste processor cycles.

FreeRTOS Tasks

What is an FreeRTOS Task?

A FreeRTOS task is a function that is added to the FreeRTOS scheduler using the `xTaskCreate()` API call.

A task will have the following:

1. **A Priority level**
2. **Memory allocation**
3. Singular input parameter (optional)
4. A Task name
5. A Task handler (optional): A data structure that can be used to reference the task later.

A FreeRTOS task declaration and definition looks like the following:

```
void vTaskCode( void * pvParameters )
{
    /* Grab Parameter */
    uint32_t c = (uint32_t)(pvParameters);
    /* Define Constants Here */
    const uint32_t COUNT_INCREMENT = 20;
    /* Define Local Variables */
    uint32_t counter = 0;
    /* Initialization Code */
    initTIMER();
    /* Code Loop */
    while(1)
    {
        /* Insert Loop Code */
    }
    /* Only necessary if above loop has a condition */
    xTaskDelete(NULL);}
```

Rules for an RTOS Task

- The highest priority ready tasks **ALWAYS** runs
  - If two or more have equal priority, then they are **time sliced**
- Low priority tasks only get CPU allocation when:
  - All higher priority tasks are **sleeping, blocked, or suspended.**
- Tasks can sleep in various ways, a few are the following:
  - Explicit "task sleep" using API call **vTaskDelay();**

- Sleeping on a semaphore
- Sleeping on an empty queue (reading)
- Sleeping on a full queue (writing)

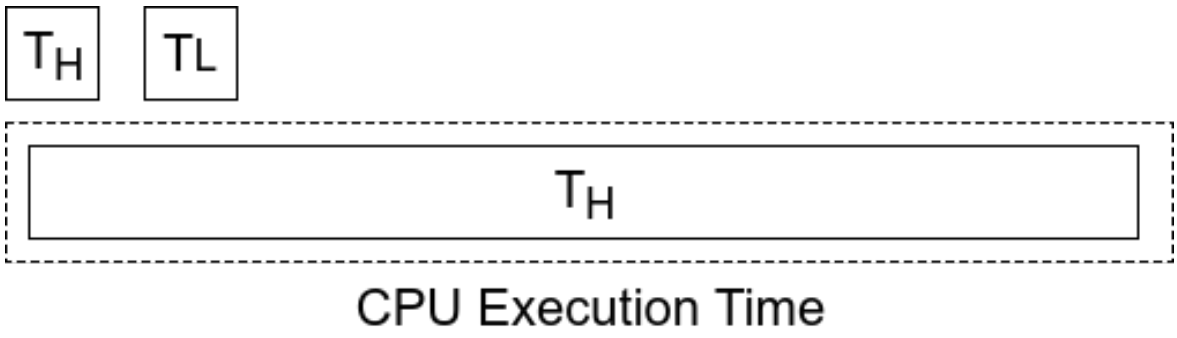
Adding a Task to the Scheduler and Starting the Scheduler

The following code example shows how to use **`xTaskCreate()`** and how to start the scheduler using **`vTaskStartScheduler()`**

```
int main(int argc, char const *argv[])
{
    //// You may need to change this value.
    const uint32_t STACK_SIZE = 128;
    xReturned = xTaskCreate(
                    vTaskCode,       /* Function that implements the task. */
                    "NAME",          /* Text name for the task. */
                    STACK_SIZE,      /* Stack size in words, not bytes. */
                    ( void * ) 1,    /* Parameter passed into the task. */
                    tskIDLE_PRIORITY,/* Priority at which the task is created. */
                    &xHandle );      /* Used to pass out the created task's handle. */
    /* Start Scheduler */
    vTaskStartScheduler();
    return 0;}
```

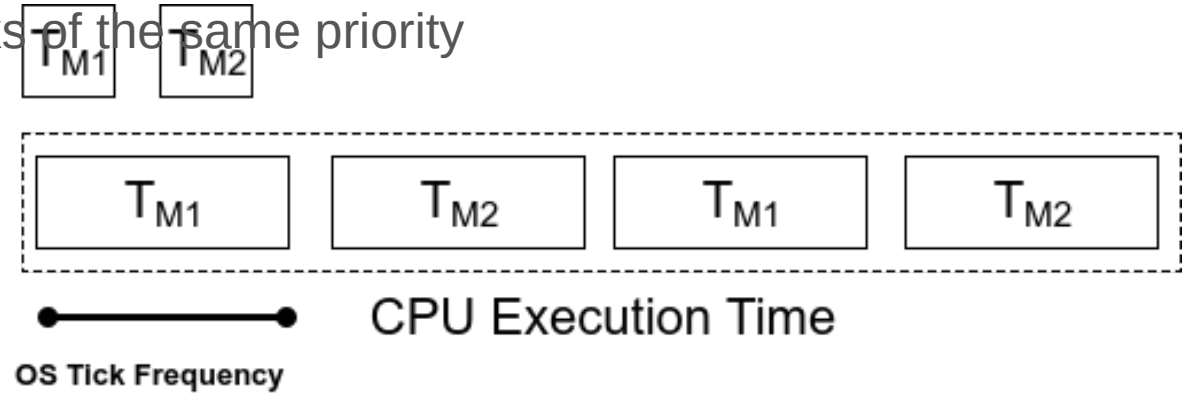
Task Priorities

High Priority and Low Priority tasks



In the above situation, the high priority task never sleeps, so it is always running. In this situation where the low priority task never gets CPU time, we consider that task to be **starved**.

Tasks of the same priority



In the above situation, the two tasks have the same priority, thus they share the CPU. The time each task is allowed to run depends on the **OS tick frequency**. The **OS Tick Frequency** is the frequency that the FreeRTOS scheduler is called in order to decide which task should run next. The **OS Tick is a hardware interrupt** that calls the RTOS scheduler. Such a call to the scheduler is called a **preemptive context switch**.

Context Switching

When the RTOS scheduler switches from one task to another task, this is called a **Context Switch**.

What needs to be stored for a Context switch to happen

In order for a task, or really any executable, to run, the following need to exist and be accessible and storable:

- **Program Counter (PC)**

    - This holds the position for which line in your executable the CPU is currently executing.
    - Adding to it moves you one more instruction.
    - Changing it jumps you to another section of code.
- **Stack Pointer (SP)**
    - This register holds the current position of the call stack, with regard to the currently executing program. The stack holds information such as local variables for functions, return addresses and [sometimes] function return values.
- **General Purpose Registers**
    - These registers are to do computation.
        - **In ARM**:
            - R0 - R15
        - **In MIPS**:
            - $v0, $v1
            - $a0 - $a3
            - $t0 - $t7
            - $s0 - $s7
            - $t8 - $t9
        - **Intel 8086**
            - AX
            - BX
            - CX
            - DX
            - SI
            - DI
            - BP

How does Preemptive Context Switch work?

1. A hardware timer interrupt or repetitive interrupt is required for this preemptive context switch.
    1. This is independent of an RTOS.
    2. Typically, 1ms or 10ms.

2. The OS needs hardware capability to have a chance to STOP synchronous software flow and enter the OS “tick” interrupt.
    1. This is called the "Operating System Kernel Interrupt"
    2. We will refer to this as the OS Tick ISR (interrupt service routine)

3. Timer interrupt calls RTOS Scheduler
    1. RTOS will store the previous **PC**, **SP**, and **registers** for that task.
    2. The scheduler picks the highest priority task that is ready to run.
    3. Scheduler places that task's **PC**, **SP**, and **registers** into the CPU.
    4. Scheduler interrupt returns, and the newly chosen task runs as if it never stopped.

Tick Rate

Most industrial applications use an RTOS tick rate of 1ms or 10ms (1000Hz, or 100Hz). In the 2000s, probably 100Hz was more common, but as processors got faster, 1000Hz became the norm. One could choose any tick rate, such as 1.5ms per tick, but using such non-standard rates makes API timing non-intuitive, as `vTaskDelay(10)` would result in sleep time of approximately `15ms`. This is yet another reason why 1000Hz is a good tick rate as `vTaskDelay(10)` would sleep for approximately `10ms`, which is intuitive to the developer because the tick times adopt the **units of milliseconds**.

With a far assumed that the RTOS tick ISR (preemptive scheduling) consumes 200 clock cycles, then on a 20Mhz processor, it would only consume 10uS of overhead per scheduling event. When cooperative scheduling triggers a context switch, it would result in a similar overhead as a "software interrupt" is issued to the CPU to perform the context switch. So this means that each scheduling event has an overhead of 20uS on a 20Mhz processor (assuming 200 clocks for RTOS interrupt). Based on these numbers, here is the overhead ratio of using different tick rates.

	100Hz	1000Hz	10,000Hz (100uS per tick)
Scheduling Overhead per second	2,000uS	20,000uS	200,000uS

CPU consumption for RTOS scheduling	0.2%	2%	20%

> Why OS Ticks are 1ms or 1KHz?
> 1. **Shorter Ticks** means that there is less time for your code to run. Ex: if OS ticks = 150us instead of 1ms and the context switching takes 100us then you are only left with 50us to complete your task. Hence, RTOS will be only busy with context switching nothing else.
> 2. **Big Ticks** such as 4ms, the CPU will remain in the wait state. For example, the context switching takes 100us then you have 3900us to complete your task. However, the task will only take 900us to complete. Then 3000us will be unnecessary overhead on CPU wait time.

Based on the numbers above, 1000Hz is a great balance, while the 10,000Hz tick rate would provide more frequent time slices at the expense of more frequent scheduling overhead.

only and nothing more More (Definitions, Synonyms, Translation)

?

Lab: FreeRTOS Tasks

Objective

1. Load firmware onto the SJ board
2. Observe the RTOS round-robin scheduler in effect
3. Provide hands-on experience with the UART character output timing

---

Part 0a. Change UART speed

We will be working with an assumption for this lab, so we will need to change the UART speed. In Visual Studio Code IDE, hit `Ctrl+P` and open `peripherals_init.c`. Then modify the UART speed to 38400. After doing so, make sure you open your serial terminal or Telemetry web terminal and change the port speed to also 38400.

```
static void peripherals_init__uart0_init(void) {
  // Do not do any bufferring for standard input otherwise getchar(), scanf() may not work
  setvbuf(stdin, 0, _IONBF, 0);
  // Note: PIN functions are initialized by board_io__initialize() for P0.2(Tx) and P0.3(Rx)
  uart__init(UART__0, clock__get_peripheral_clock_hz(), 38400); // CHANGE FROM 115200 to 38400

  // ...}
```

The `peripherals_init__uart0_init()` is executed before your `main()` function. When you are finished with this lab, you can choose to change this back to 115200bps for faster UART speed.

Part 0b. Create Task Skeleton

A task in an RTOS or FreeRTOS is nothing but a forever loop, however unless you sleep the task, it will consume 100% of the CPU. For this part, study existing `main.c` and create two additional tasks for yourself.

```
#include "FreeRTOS.h"
#include "task.h"
static void task_one(void * task_parameter);
static void task_two(void * task_parameter);
int main(void) {
  // ...
}
static void task_one(void * task_parameter) {
  while (true) {
    // Read existing main.c regarding when we should use fprintf(stderr...) in place of printf()
    // For this lab, we will use fprintf(stderr, ...)
    fprintf(stderr, "AAAAAAAAAAA");

    // Sleep for 100ms
    vTaskDelay(100);
  }
}
static void task_two(void * task_parameter) {
  while (true) {
    fprintf(stderr, "bbbbbbbbbbb");
    vTaskDelay(100);
  }}
```

Part 1: Create RTOS tasks

1. Fill out the `xTaskCreate()` method parameters.
   - See the FreeRTOS+Tasks document or checkout the FreeRTOS xTaskCreate API website
   - Recommended stack size is: **`4096 / sizeof(void*)`**
2. Note that you want to make sure you use `fprintf(stderr, ...)` in place of `printf(...)`
   - `fprintf(stderr, ...)` is slower and eats up CPU, but it is useful during debugging

- `printf(...)` is faster (and efficient), but it queues the data to be "sent later"

3. Observe the output
   - After you flash your program, check the output of the serial console

```
#include "FreeRTOS.h"
#include "task.h"
static void task_one(void * task_parameter);
static void task_two(void * task_parameter);
int main(void) {
    /**
     * Observe and explain the following scenarios:
     *
     * 1) Same Priority:      task_one = 1, task_two = 1
     * 2) Different Priority: task_one = 2, task_two = 1
     * 3) Different Priority: task_one = 1, task_two = 2
     *
     * Note: Priority levels are defined at FreeRTOSConfig.h
     * Higher number = higher priority
     *
     * Turn in screen shots of what you observed
     * as well as an explanation of what you observed
     */
    xTaskCreate(task_one, /* Fill in the rest parameters for this task */ );
    xTaskCreate(task_two, /* Fill in the rest parameters for this task */ );
    /* Start Scheduler - This will not return, and your tasks will start to run their while(1) loop */
    vTaskStartScheduler();
    return 0;
}
// ...
```

Part 2: Further Observations

Fundamentals to keep in mind:

- FreeRTOS tick rate is configured at 1Khz
  - This means that the RTOS preemptive scheduling can occur every 1ms repetitively

- Standout output (`printf`) is integrated in software to send data to your UART0
  - This is the same serial bus that is used to load a new program (or hex file)
  - The speed is defaulted to 38400bps, and since there is 10 bits of data used to send 1 byte, we can send as many as 3840 characters per second

Critical thinking questions:

- How come 4(or 3 sometimes) characters are printed from each task? Why not 2 or 5, or 6?
- Alter the priority of one of the tasks, and note down the observations. Note down WHAT you see and WHY.

> **Hint:** You have to relate the speed of the RTOS round-robin scheduler with the speed of the UART to answer the questions above

---

Part 3. Change the priority levels

Now that you have the code running with identical priority levels, try the following:

1. Change the priority of the two tasks

   * Same Priority: `task_one` = 1, `task_two` = 1

   * Different Priority: `task_one` = 2, `task_two` = 1

   * Different Priority: `task_one` = 1, `task_two` = 2
2. Take a screenshot of what you see from the console
3. Write an explanation of why you think the output came out the way it did using your knowledge about RTOS

> Optional: If you have TraceAlyzer program installed, we encourage you to load this file and inspect the trace.

What to turn in:

1. Relevant code
2. Your observation and explanation
3. Snapshot of the output for all scenarios

If your class requires you to turn in the assignment as a Gitlab link, you should:

- Use this article to get started
- Submit a link to Gitlab "Merge Request"
- Be sure to ensure that your Merge Request is only the new code, and not a very large diff