# MTE 241 RTOS Lab Specification

Updated 2019-10-24

## Configure uVision5 project:

- Project > New uVision Project
- Select Device NXP LPC1768
- Manage Run-Time Environment dialog box
    - expand Device; check Startup
    - expand CMSIS; click CORE
    - do not choose an RTOS
    - click OK
- double-click Target 1 > Device > startup_LPC17xx.s
    - choose Configuration Wizard tab
    - change Stack size to 0x2000 (= 8 kiB)
    - leave Heap size at 0x0
- right-click Target 1 > Options for Target 'Target 1'
    - in C/C++ tab Define __RTGT_UART
- right-click Target 1 > Source Group 1
    - add existing files Retarget.c, uart.c, main_default.c

## Background

The Cortex-M3 has two processor modes: Handler mode and Thread mode. Handler mode is used automatically for exception handlers (except the Reset handler) and has privilege level Privileged: that is Handler mode code can execute privileged and unprivileged instructions. Application code executes in Thread mode and can have privilege level Unprivileged or Privileged. There are two stack pointers: the Main Stack Pointer (MSP) and the Processor Stack Pointer (PSP). Handler mode uses the Main stack. Thread mode can use the Main stack or the Process stack depending on the SPSEL bit in the Control register: 0 = MSP, 1 = PSP.

After reset the thread mode defaults to privilege level Privileged and the Main stack. We will leave the privilege level alone but will change the stack to the Process stack. The Main stack will be used for exception handlers and we'll leave it 2 kiB. The tasks will each have their own stack that is 1 kiB in size. Since we configured the stack size to 8 kiB that means that we can have up to 6 tasks. Here is the memory map:

*Table 1 RTOS Memory Map*

| starting address | | size |
|---|---|---|
| 00000000 | Text Section | |
| 10000000 | Data Section | |
| | TCB[6] | |
| | Stack 0 | 1 kiB |
| | Stack 1 | 1 kiB |
| | Stack 2 | 1 kiB |
| | Stack 3 | 1 kiB |
| | Stack 4 | 1 kiB |
| | Stack 5 | 1 kiB |
| | Main stack | 2 kiB |

The Task Control Blocks (TCBs) are actually part of the Data section (you should declare them as global variables). The base address for the Main stack is found in Vector 0 of the Vector Table at address 0x0.

## Setup

You will need to provide a kernel initialization function and start function. The initialization function will initialize each TCB with the base address for its stack. You can work backwards from the base address of the Main stack (found in Vector 0). One TCB should be allocated to the idle task. It is up to you to define the rest of the contents of the TCB.

After initialization, the application should create at least one task, then invoke the kernel start function. The start function should not return but instead transform into the idle task. Here are the start function steps:

1. Reset the MSP to the main stack base address. The main stack will only be used by exception handlers.
2. Switch from using the MSP to the PSP by changing the Stack Pointer Select (SPSEL) bit in the Control register. The Stack Pointer register (R13) is a banked register. Before switching the SPSEL bit it will have the address of the MSP. After switching stacks, the stack pointer will have the address of the PSP which will not contain a valid address. You should set it to the base of the stack that you selected for your idle task.

3. Configure SysTick and invoke the idle task function.

CMSIS provides “intrinsics” (functions that expose processor-specific assembly instructions). Use the Core Register Access functions (https://www.keil.com/pack/doc/CMSIS/Core/html/group__Core__Register__gr.html) to access the MSP, PSP and Control registers. There is a CONTROL_Type union that you can use when setting the SPSEL bit in the Control register (or you can define your own bitmask).

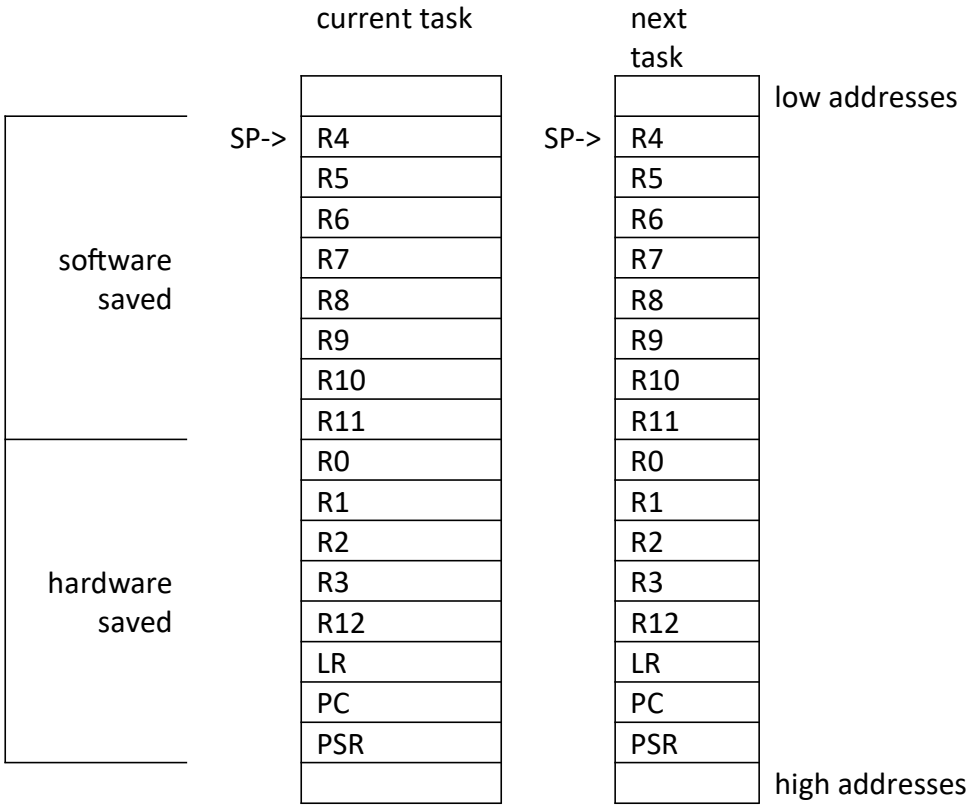
## Context Switch

To perform a context switch you:

1) push register contents onto the current task’s stack
2) record the current stack pointer address in the current task’s TCB
3) load the next task’s top of stack address into the stack pointer
4) pop register contents from the next task’s stack

On ARM Cortex-M3 you implement the context switch in the PendSV_Handler(). When the Cortex-M3 enters an exception handler it pushes these eight registers on the stack R0-R3,R12,LR,PC,PSR (LR =R14 = link register for procedure call return, PC =R15 = program counter, PSR = processor status register). When it exits the exception handler it pops from the stack into those registers. To perform a context switch then, the software just needs to push/pop R4-R11 to/from the stack and manipulate the SP (SP = R13 = stack pointer). This is pictured in Table 2 following.

*Table 2 Context on Task Stack*




PendSV_Handler() should be implemented in assembly language. The reason is that the C compiler starts each function by preserving any registers used by the function and it does this by pushing them onto the stack. Writing the handler is assembly language avoids these extra pushes. An empty PendSV_Handler() is included in `main_default.c` which simply does a return. You should push R4-R11 using the PUSH or STMFD instructions and copy the stack pointer to a global variable. To get the address of global variable x into register R1, the ARM assembly code is:

```
LDR        R1,=__cpp(&x)
```

You could then store the SP using R1 as the address register (see STR instruction). You'll also need to get the address of the next stack pointer from a global variable and pop R4-R11 using the POP or LDMFD instructions.

## Create Task

You will need to provide a function to create new tasks. The parameters should include a pointer to the task function, and a task function argument. The function pointer type can be declared as:

```
typedef void (*rtosTaskFunc_t)(void *args);
```

`rtosTaskFunc_t` is the type name for a pointer to a function that takes a void * parameter and returns void.

When you create a new task, you need to initialize its stack according to Table 2. The only parts that matter are:

- R0 – argument to task function
- PC – address of task fuction
- PSR – default value is 0x01000000

However the other parts (R4-R11, R1-R3,R12,LR) must exist so that the context pop works correctly.

## SysTick Handler

The `main_default.c` that is provided shows how to configure the SysTick to 1ms and declare the `SysTick_Handler()`. You will need to modify the SysTick handler to suit your purposes including triggering the PendSV exception when a context switch is required. To do this you need to write a 1 to the PENDSVSET bit of the Interrupt Control and State Register (ICSR). This register is described in Table 4-15 of the Cortex-M3 Devices Generic User Guide found at http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0552a/index.html. By including the LPC17xx.h header file you can reference the ICSR as `SCB->ICSR` in your code.

## Interrupt Priorities

The SysTick handler should have the highest priority (0x00) so that system ticks are not missed while handling other interrupts. The PendSV handler should have the lowest priority (0xff) so that the context of nested handlers is not present on the stack when doing the context switch. Set these priorities using the `NVIC_SetPriority()` intrinsic function.

## Deliverables

- To declare your intent to create an RTOS for Lab 5 instead of a game, upload a Zip file to the RTOS Prelab dropbox on Learn instead of submitting to the game design dropbox. The Zip file should contain a uVision 5 project that demonstrates a working context switch. Also send the names, student numbers, and email addresses of your team members to Andrew Morton.
- For Lab 5, your RTOS should implement the following: kernel initialization, task creation, kernel start, context switch, fixed-priority preemptive scheduling, blocking semaphores, and blocking mutexes with owner test on release and priority inheritance.
- Create a test suite that demonstrates the following:
    - context switching between at least two tasks 40%
    - FPP scheduling 20%
    - blocking semaphores 20%
    - blocking mutex with owner test on release (10%) and priority inheritance (10%)
- Extra fun stuff to try (not for marks):
    - task termination
    - delay and/or timer functions