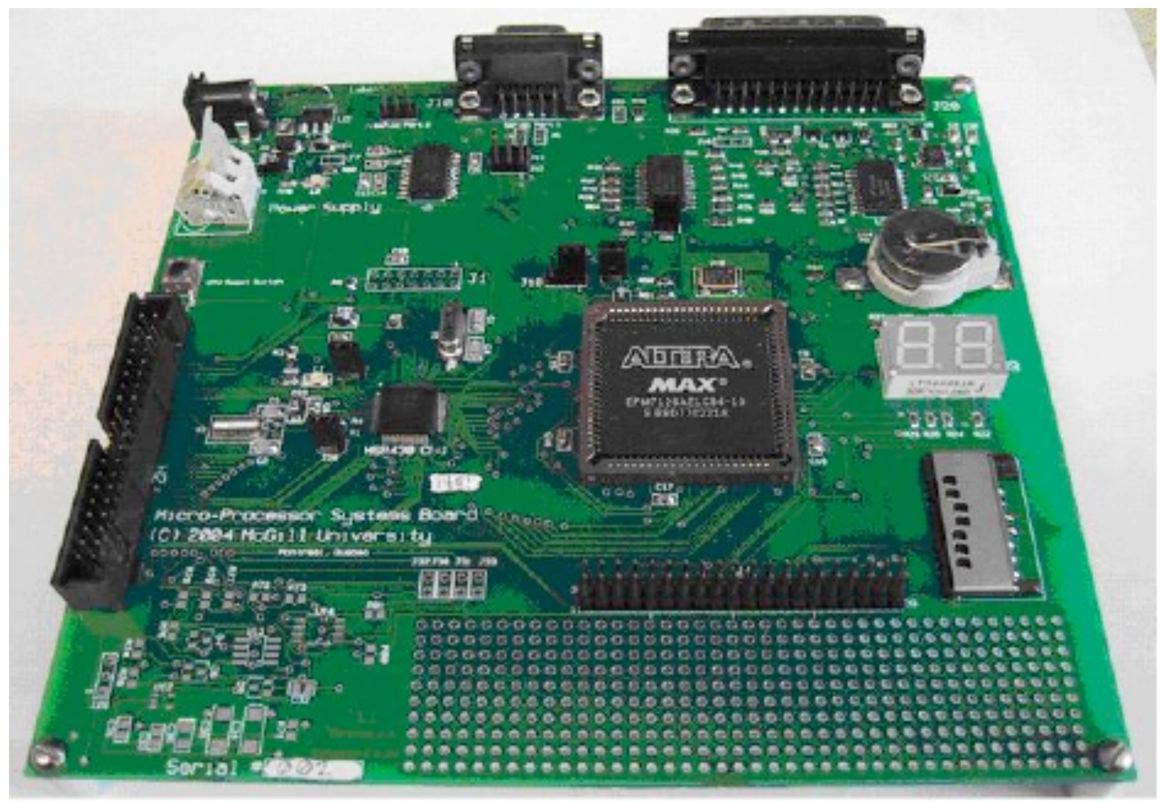
ECSE 444
Embedded Operating Systems

Zeljko Zilic

zeljko.zilic@mcgill.ca

McGill



Acknowledgments: to STMicroelectronics for material on processors and the board

Topics for Today

- Lab 4 background
- Final project preview
- Interrupts+Exceptions Refresh
  - SVC + PendSV
- Operating system services
- CMSIS-RTOS
- FreeRTOS

Lab 4: Putting Things Together

- Integrate Devices
  - Multiple sensors,
  - Periodic tasks,
  - Interrupts
- Simpler ways
  - Further abstract hardware (no need for, e.g., timer)
  - Abstract SW and OS
- Use CMSIS-RTOS/FreeRTOS
- Revisit design decisions
  - Improve, abandon poorer solutions based on evaluation of sw with OS

Lab 4: Strategy

- New tools to learn (and later on harness)
  - OS Services
  - (CMSIS-RTOS abstraction of OS)
- Make high-level solution expressed in RTOS primitives
  - Threads
  - ISRs
  - Inter-Process Communication (IPC)
- Split tasks in team, in your code
  - Code and deploy
  - Debug techniques- observe enough, gain from OS
- UI, optimization and performance gains

Final Project: Venturing Anew, Creative

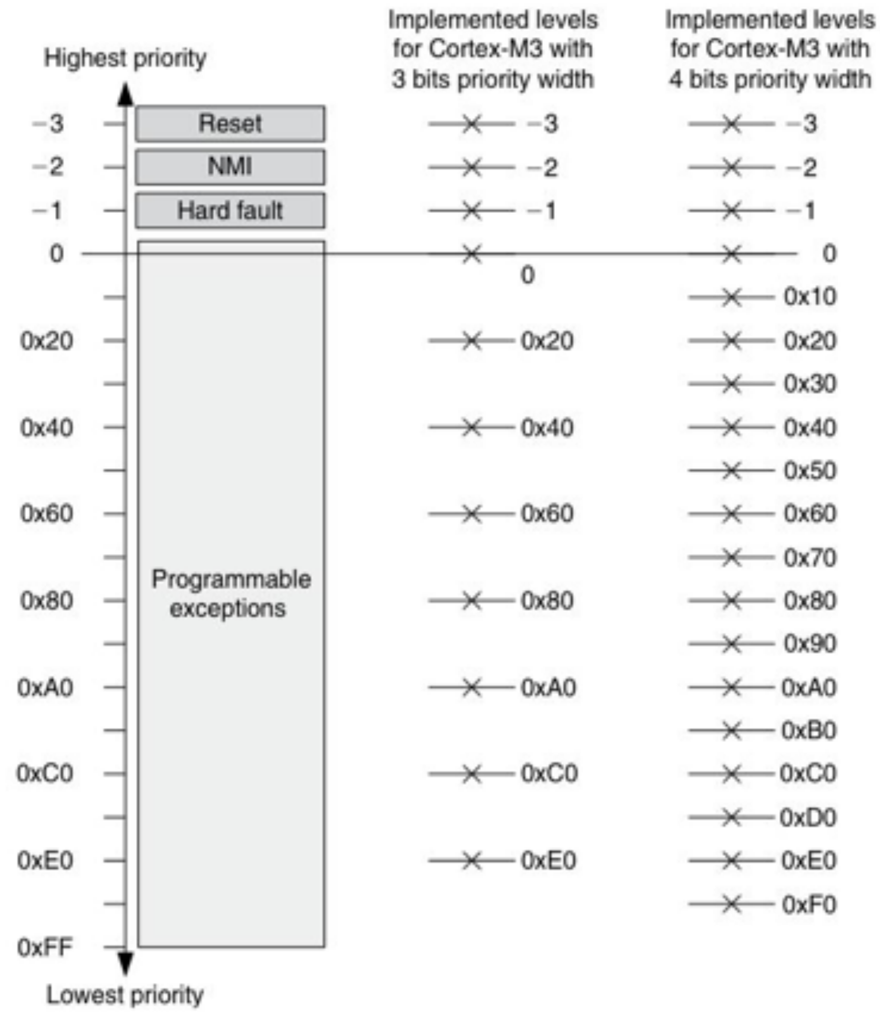
- Start from scratch, using everything learned so far
  - Sensors, threads, debug
- Neat application: to develop
  - Use sound devices, sensors
  - Lots of extra features, options to choose
- Plan for success
- Finish and add to your CV

Operating System (OS) Support

- Double (shadowed) stack pointer
  - MSP vs. PSP – main vs. per-process stack pointer
- SysTick timer
  - Initiating context switches
  - Could be substituted by regular timers
- SVC and PendSV
  - Service calls, surrendering control to OS
- Exclusive access instructions
  - Used to implement semaphores, mutual exclusion, …

Exceptions

- Asynchronous processing of various types:
  - Reset
  - Non-maskable interrupt
  - Various faults (mem, bus, usage)
  - Supervisor call
  - Debug and monitoring
  - External interrupts
- Each type can have multiple subtypes/priorities



Cortex-M Exception Types

No.	Exception Type	Priority	Type of Priority	Descriptions
1	Reset	-3 (Highest)	fixed	Reset
2	NMI	-2	fixed	Non-Maskable Interrupt
3	Hard Fault	-1	fixed	Default fault if other hander not implemented
4	MemManage Fault	0	settable	MPU violation or access to illegal locations
5	Bus Fault	1	settable	Fault if AHB interface receives error
6	Usage Fault	2	settable	Exceptions due to program errors
7-10	Reserved	N.A.	N.A.	
11	SVCall	3	settable	System Service call
12	Debug Monitor	4	settable	Break points, watch points, external debug
13	Reserved	N.A.	N.A.	
14	PendSV	5	settable	Pendable request for System Device
15	SYSTICK	6	settable	System Tick Timer
16	Interrupt #0	7	settable	External Interrupt #0
……	…………………..	…………………..	settable	…………………..
256	Interrupt#240	247	settable	External Interrupt #240

Supervisor Calls: SVC & PendSV

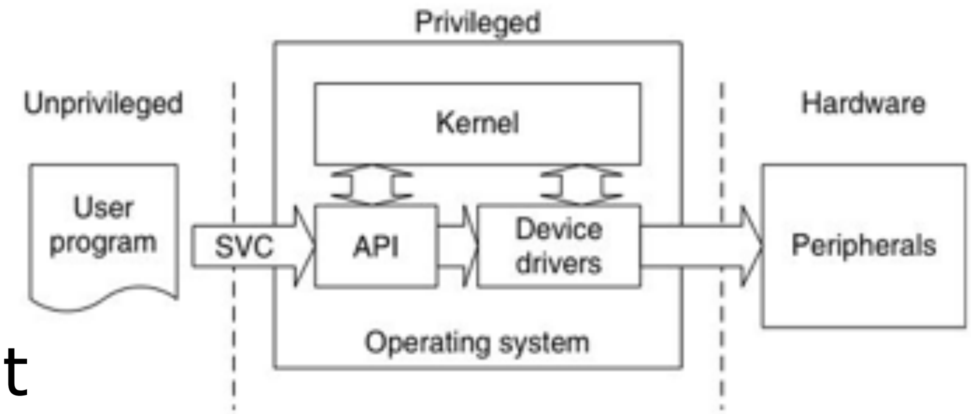
- SVC (Supervisor Call) and PendSV (Pendable Service Call)
  - Important for OS designs

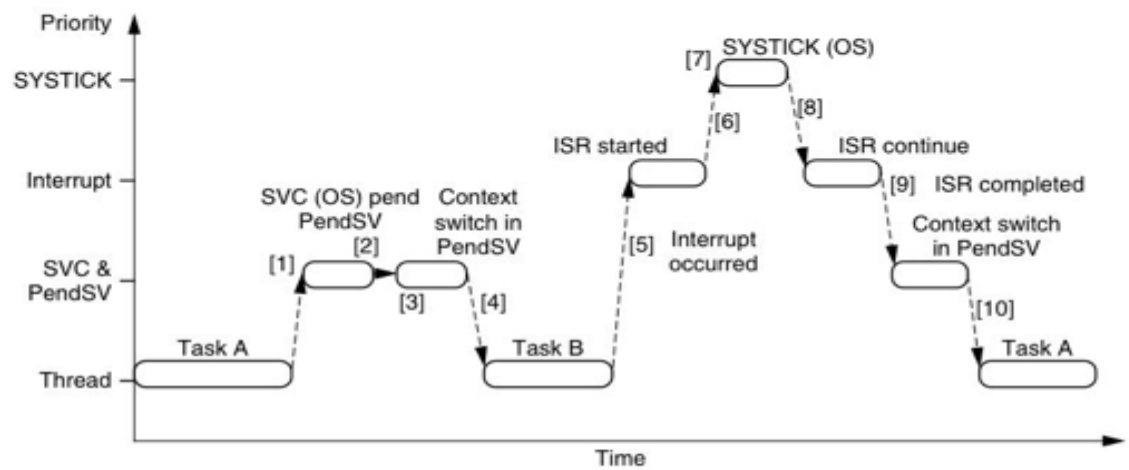
- SVC Instruction
  - Keil/ARM: `__svc`
  - Portable; hardware abstraction
  - Can write to NVIC using a software trigger interrupt register, but several instructions might execute while the interrupt is pending
  - With __svc, the SVC handler executes immediately (except when another higher priority exception arrives)

- Can be used as API to allow application tasks to access system resources

Supervisor Calls: SVC & PendSV

- Supervisor Call (SVC): for system function calls
  - Portable; HW abstraction
  - Can't nest! (no SVC in SVC)
- SVC Instruction
  - Keil/ARM: `__svc`
- Pendable SV: can wait/nest
- SysTick: OS clock
  - Good for RTOS
  - 24-bit down counter
  - 2 clock sources
  - Only priviledged mode

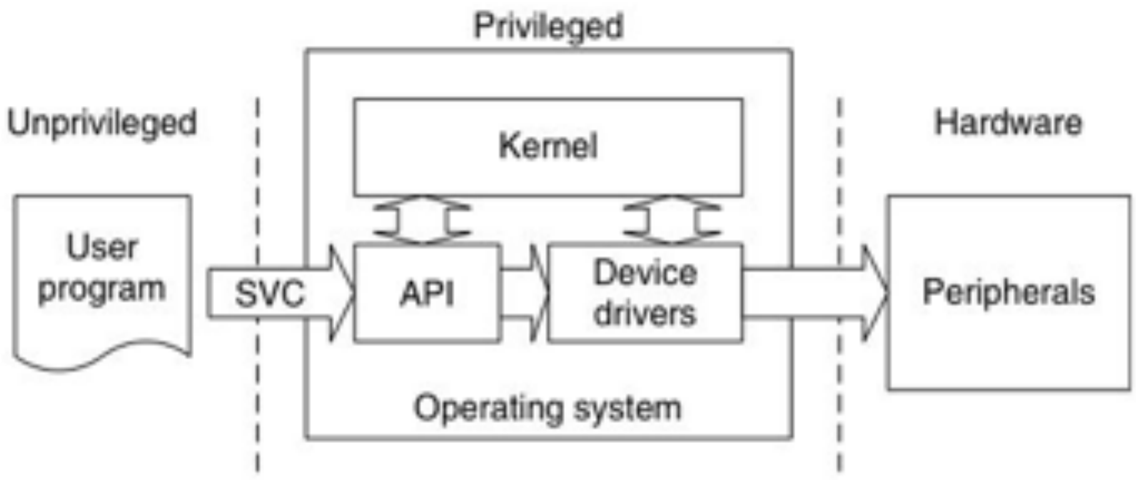




McGill

Supervisor Calls: SVC

- Application tasks
  - Run in unprivileged access level
  - Can only access protected resources via services from OS
  - Cannot gain unauthorized access to critical hardware
  - No need to know programming details of underlying hardware
  - Can be developed independently of OS (don't need to know exact address of OS service functions – only the SVC service number and parameters)
  - Hardware level programming handled by device drivers



SVC Handling

- SVC instruction (`__svc`)
  - Needs immediate value (`SVC #0x3; Call SVC function 3`)
  - SVC exception handler extracts parameter and knows which action to perform
- Procedure
  - Determine if the calling program was using main stack or process stack?
  - Read the program counter value from appropriate stack
  - Read instruction from that address (masking out unnecessary bits)

```
SVC_Handler       ; NOTE: Code written in ARM Assembler!
  TST lr, #4           ; Test bit 2 of EXC_RETURN
  ITE EQ
  MRSNE R0, MSP    ; if 0, stacking using MSP, copy to R0
  MRSEQ R0, PSP    ; if 1, stacking using PSP, copy to R0
  LDR R0, [R0, #24] ; get stacked PC from stack frame
         ; stacked PC = address of instruction after SVC)
  LDRB R0, [R0, #-2] ; get first byte of the SVC instruction
         ; now SVC number is in R0
```

Source : J. Yiu, “The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors”

McGill

SVC Handling

- In C, we break it into two parts
- Can’t check the value of LR (EXC_RETURN) in C
- Use assembly inline (`__asm`)

```
__asm void SVC_Handler(void)
{
   TST LR, #4          ; Test bit 2 of EXC_RETURN
   ITE EQ
   MRSNE R0, MSP    ; if 0, stacking using MSP, copy to R0
   MRSEQ R0, PSP    ; if 1, stacking using PSP, copy to R0
   B _cpp(SVC_Handler_C)
   ALIGN 4
}
```

Source : J. Yiu, “The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors”

SVC Handling

```
void SVC_Handler_C(unsigned int * svc_args)
{
  uint8_t svc_number;
  unint32_t stacked_r0, stacked_r1, stacked_r2, stacked_r3;

  svc_number = ((char *) svc_args[6])[-2];
  // Memory[(Stacked PC)-2]
  stacked_r0 = svc_args[0];
  stacked_r1 = svc_args[1];
  stacked_r2 = svc_args[2];
  stacked_r3 = svc_args[3];

  // other processing
  …
  // Return result (e.g. sum of first two elements)
  svc_args[0] = stacked_r0 + stacked_r1;
  return;
}
```

Source : J. Yiu, “The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors”

SVC Handling

- Passing the address of the stack frame allows the C handler to extract any information it needs
- Essential if you want to pass parameters to an SVC service and get a return value

- A higher priority interrupt could be executed first and change the values of R0, R1, etc.
- Using the stack frame ensures your SVC handler gets the correct input parameters

PendSV

- Pended Service Call (exception type 14)
  - Programmable priority level
  - Triggered by writing to Interrupt Control and Status Register (ICSR)
  - Unlike SVC, it is not precise. Pending status can be set in a higher priority exception handler
- Execution of OS kernel (and context switching) triggered by
  - SVC call from an application task (e.g. task is stalled because it is waiting for data or an event)
  - Periodic SysTick exception
- PendSV is lowest priority exception
  - Context switching delayed until all other IRQ handlers have finished
  - OS can set pending status of PendSV & carry out context switching in PendSV exception handler

SVC and PendSV for OS Calls

- Starting OS, context switch, …
- Examples from J. Yiu’s “Definitive Guide to ARM Cortex”

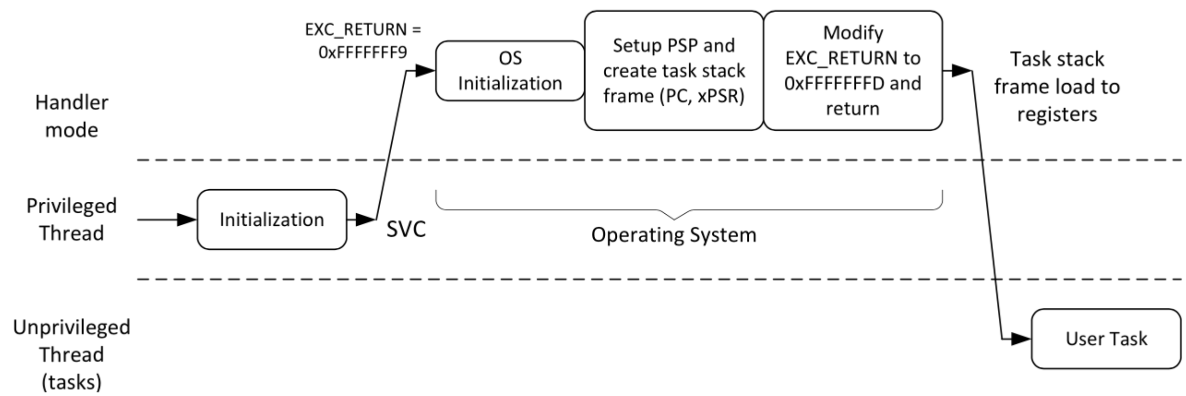


FIGURE 10.2

Initialization of a task in a simple OS

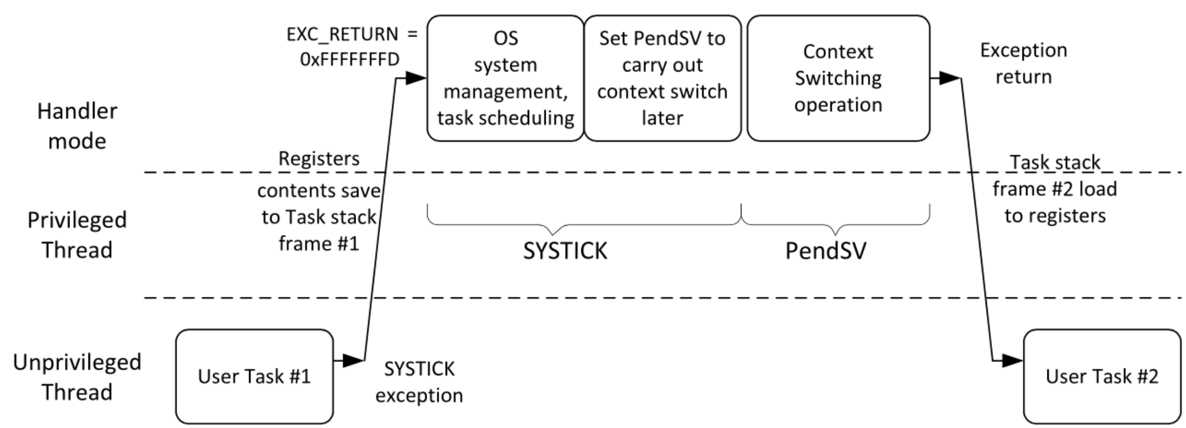


FIGURE 10.3

Concept of context switching

Operating System Services

- Making embedded programming similar to desktop
  - Large number of pre-made services
- Providing real-time operation
- Lots of embedded RTOS choices
  - Free RTOS, RTX (ARM), Azure RTOS, uCos
- Link to ARM Cortex HW
  - SVC, PendSV, SysTick

OS Services

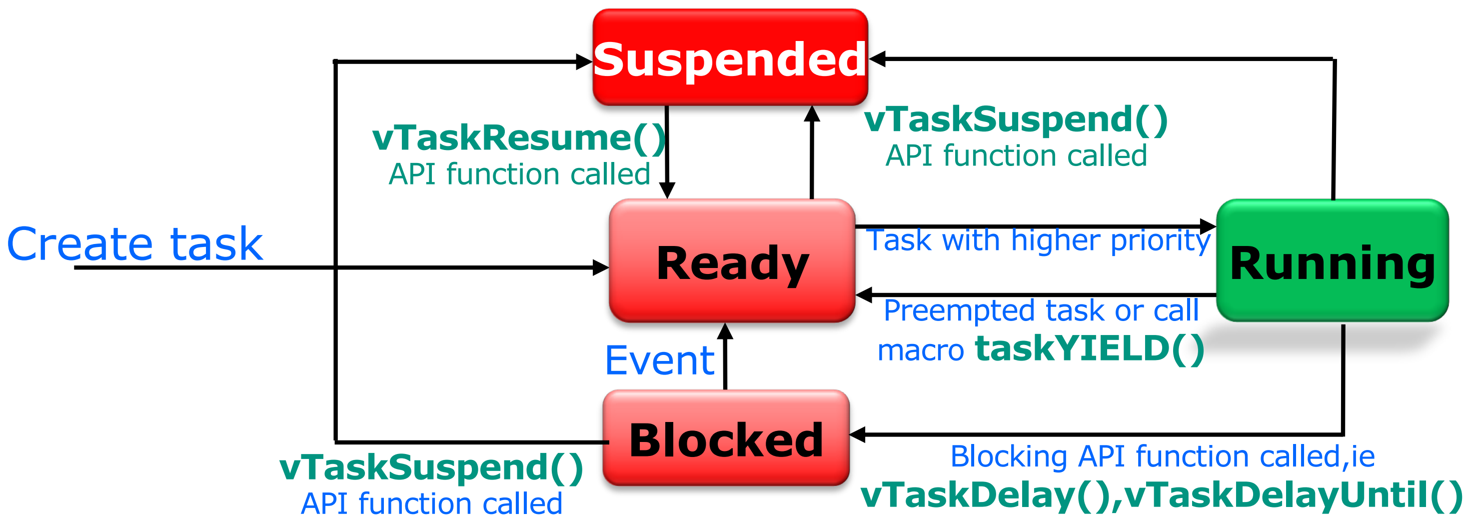
- Process management
  - Create, terminate, signal
- File management
  - Open, read, write, close, lock
- Memory management
  - Virtual memory, sharing, protection
- Date and time
- User management
- Networking

Real-Time OS – FreeRTOS

- Compact kernel, suitable for smallest (8- and 16-bit) processors
- Low overhead, simple use scenarios
- Application built by
  - *Tasks*: code run periodically
  - *Semaphores*: synchronization between tasks
  - *Queues*: data transfer management

FreeRTOS –Tasks

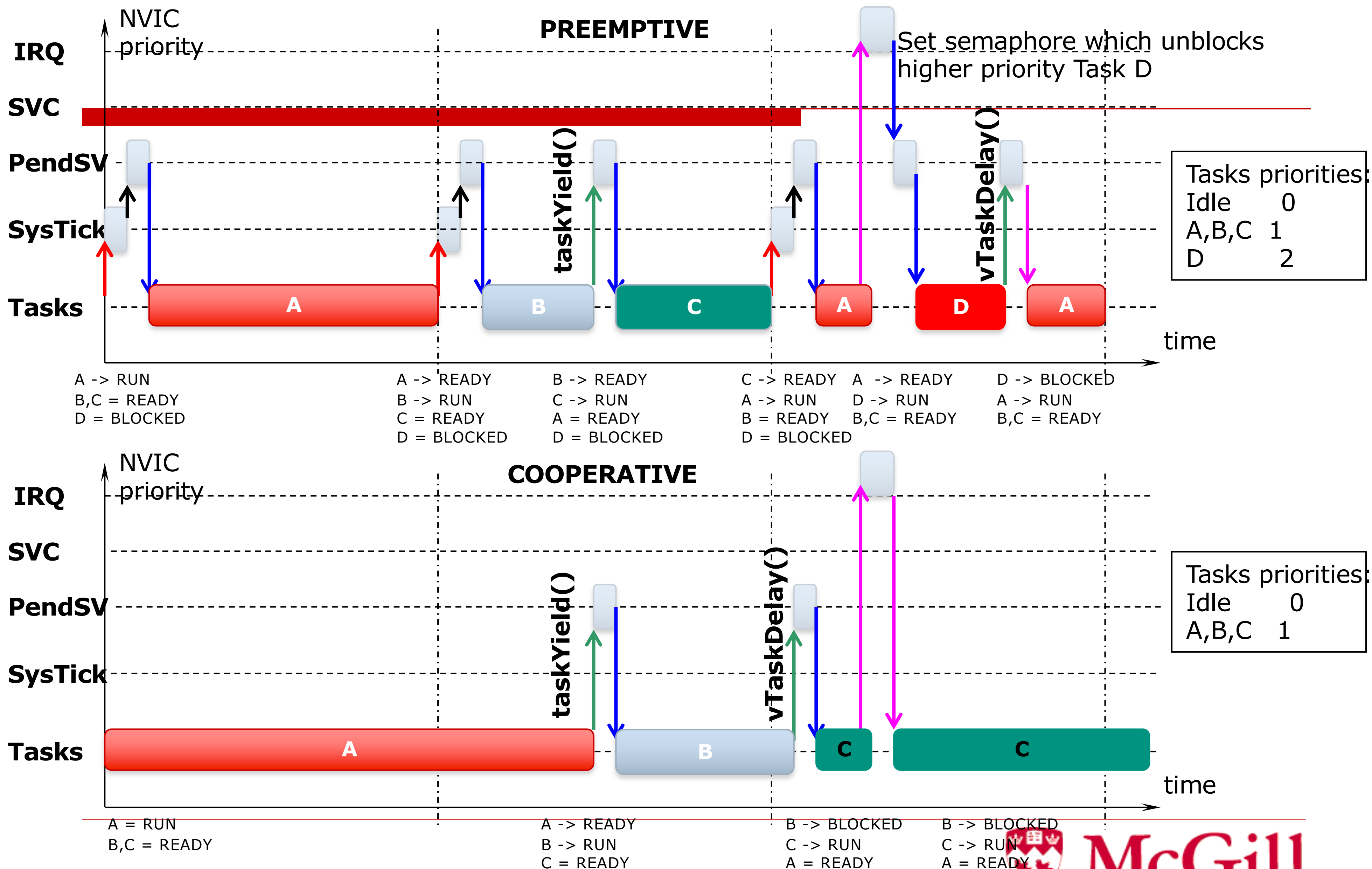
- A C function **void vTask(void *pvParameters)** used to run any number of separate instances by the function **xTaskCreate()**.
- Task can be in one of the 4 states:
  Running : it is currently using the processor. Only **one task can be in RUN** mode at the moment
  Ready : able to execute (not blocked or suspended) but a different task of equal or higher priority is already in the Running state
  Blocked : waiting for either a temporal (indicated by TiTick when) or external event (indicated by semaphore or queue).
  Suspended : not available for scheduling. Tasks will only enter or exit the suspended state when explicitly commanded to do so through the API calls of the RTOS (**Task_Suspend()** and **Task_Resume()** respectively).



- Each task has its own stack area and its priority (which can be changed)
- To free RAM, recommended to delete the task when no longer in use (function: **xTaskDelete**(task handler or NULL when we would like to delete current task)

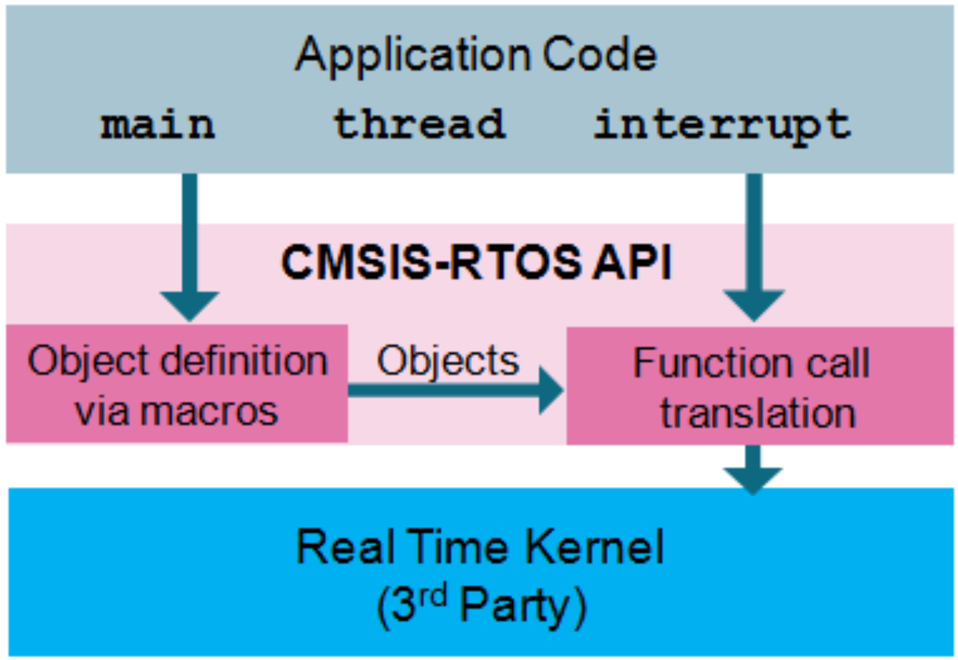
McGill

Context Switching



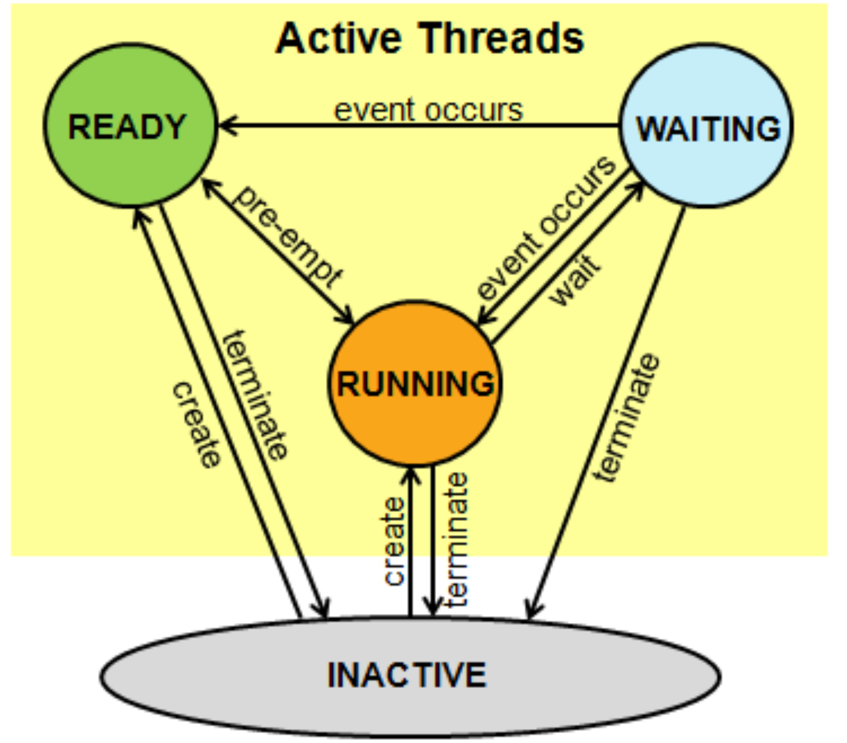
Abstracting Away OS – CMSIS-RTOS

- Makes RTOS-es look the same
- Execution under 3 stubs
  - Main
  - Thread
  - Interrupt
- Example: mapping multiple sensor processing, interrupts



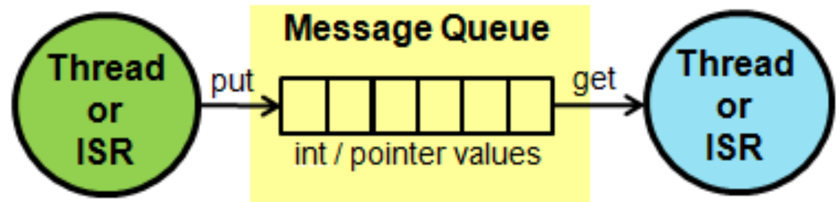
Thread Management

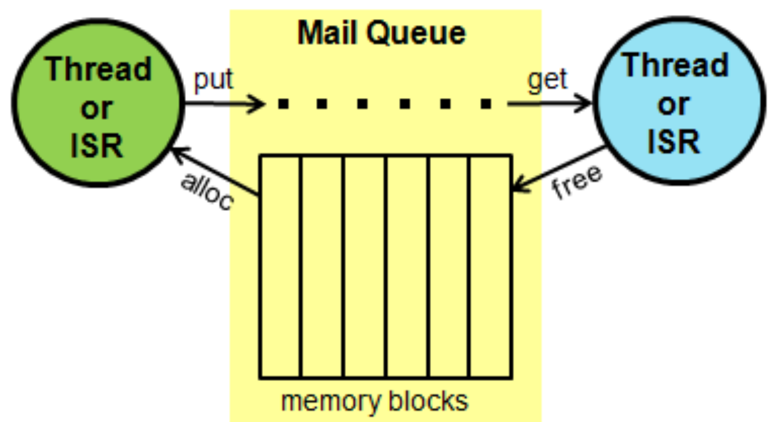
- Define, create, manage threads
- 6 `osThread` Functions
- Special `MainThread` at system initialization
- Thread states allow for controlled multithreading
- Context switching: round robin or deterministic



Services: Signals, Messages, Mail

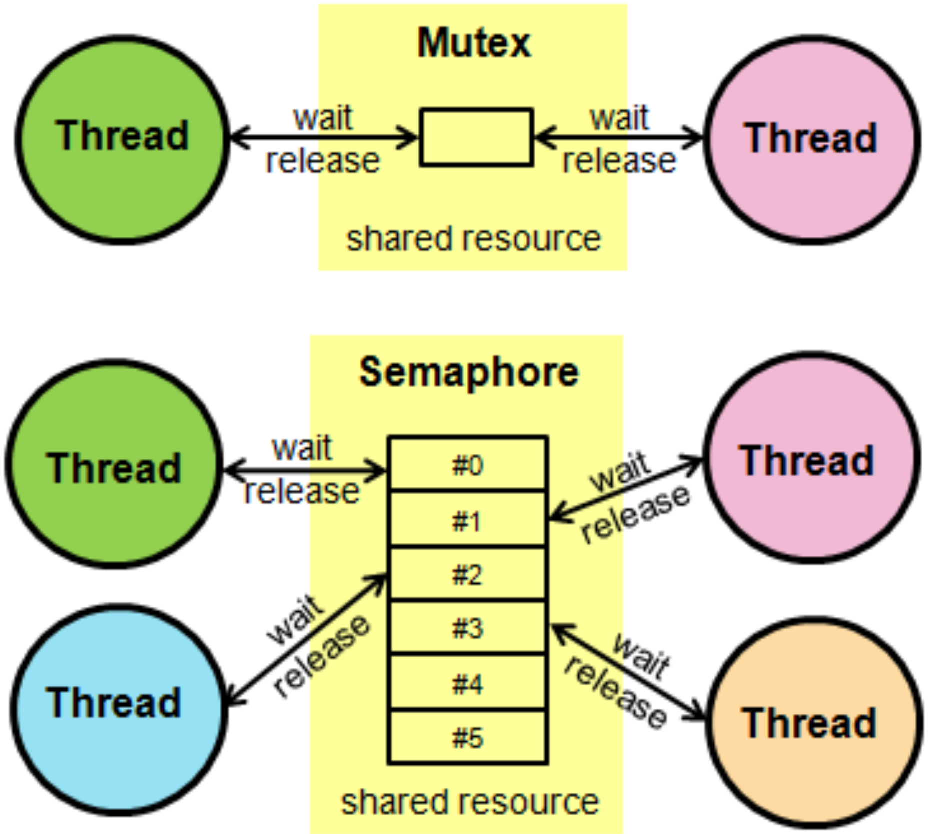
- Signals: simplest thread communication
  - Synchronization between threads: waiting until flags set by other thread
- Messages: queuing
  - Integer or pointer
- Mail: larger blocks





Mutex, Semaphore

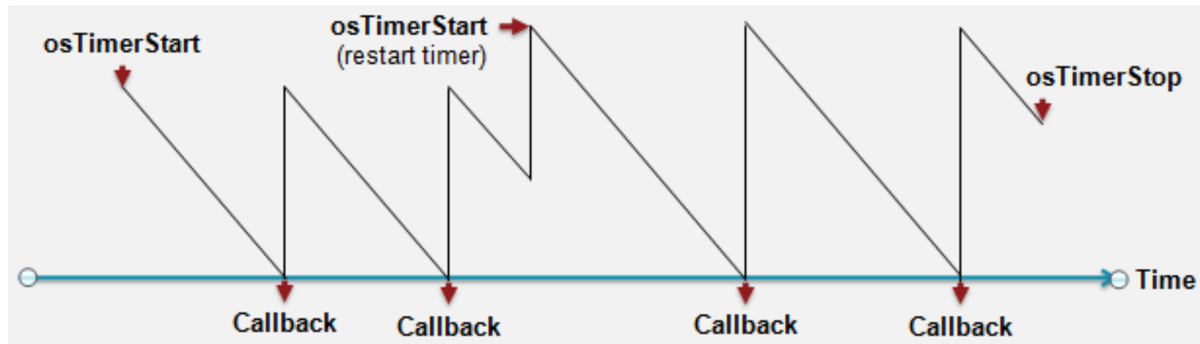
- Mutex: thread synchronization for access protection
- Semaphore:
  - Counting
  - Binary



McGill

Timer, Wait Function

- Timer causes callbacks on the period expiration
  - OS Timer thread



- Wait provides OS event or delay in ms
  - Signal, message, mailbox or timeout

McGill

Applications with CMSIS-RTOS

- Define threads, ISRs, main, timers
- Initialize thread management
- Start Execution
- Use underlying RTOS functions

```
int main (void) {                          // program execution starts here
    :                                      // setup and initialize
  osKernelStart (osThread(job2), NULL);        // start kernel with job2 execution
  while (1);                                  // program will never reach this point
}
```

STM32 + FreeRTOS

Core resources used:

- System timer (SysTick) – generate system time (time slice)
- Two stack pointers: MSP, PSP

Interrupt vectors used (those 3 vectors should be removed from ***stm32f4xx_it.c*** file):

- **SVC** – system service call (like SWI in ARM7)
- **PendSV** – pended system call (switching context)
- **SysTick** – System Timer

File system -> please refer to next slide. Most important one are ***port.x*** and ***portmacro.x*** files which are strictly dedicated to the core and software toolchain

Configuration of the system is done via ***FreeRTOSConfig.h*** file.

FreeRTOS Source File Structure

File / header Directory	role
croutine.c / croutine.h .\Source .\Source\include	Co-routines functions definitions. Efficient in 8 and 16bit architecture. In 32bit architecture usage of tasks is suggested
heap_x.c .\Source\portable\MemMang	Memory management functions (allocate and free memory segment, three different approaches in heap_1, heap_2 and heap_3 files).**Heap_2.c** is the most efficient one
list.c / list.h .\Source .\Source\include	List implementation used by the scheduler.
port.c / portmacro.h .\Source\portable\gcc\ARM_CM3	Low level functions supporting SysTick timer, context switch, interrupt management on low hw level – strongly depends on the platform (core and sw toolset). Mostly written in assembly. In portmacro.h file there are definitions of portTickType and portBASE_TYPE
queue.c / queue.h .\Source .\Source\include	Queues, semaphores, mutexes function definition
tasks.c / task.h .\Source .\Source\include	Task functions and utilities definition
FreeRTOS.h .\Source\include	Configuration file which collect whole FreeRTOS sources
FreeRTOSConfig.h	Configuration of FreeRTOS system, system clock and irq parameters configuration

FreeRTOS – structure of the code

- Include files:
  - ***FreeRTOS.h*** – main header file for basic functionality of FreeRTOS
  - ***Task.h*** – if tasks will be used in the application (99% cases)
  - ***Semphr.h*** – if semaphores/queues will be used in the application
- At beginning of the main() all hardware configuration should be done (clock, GPIO configuration). In our example it is **prvSetupHardware()** function
- Next step is creation of the basic components of the application:
  - **tasks** – piece of the code executed periodically (ie. LCD_control, GPIO_control)
  - **semaphores** – used to synchronization among the tasks and between the task and interrupt
  - **queues** – used to transfer data between the tasks
- Last point in main() function is starting the scheduler:
  vTaskStartScheduler()
- If the MCU will pass start scheduler function it means that stack is overflowed
- Code of the **task** should be put in never ending loop, like:

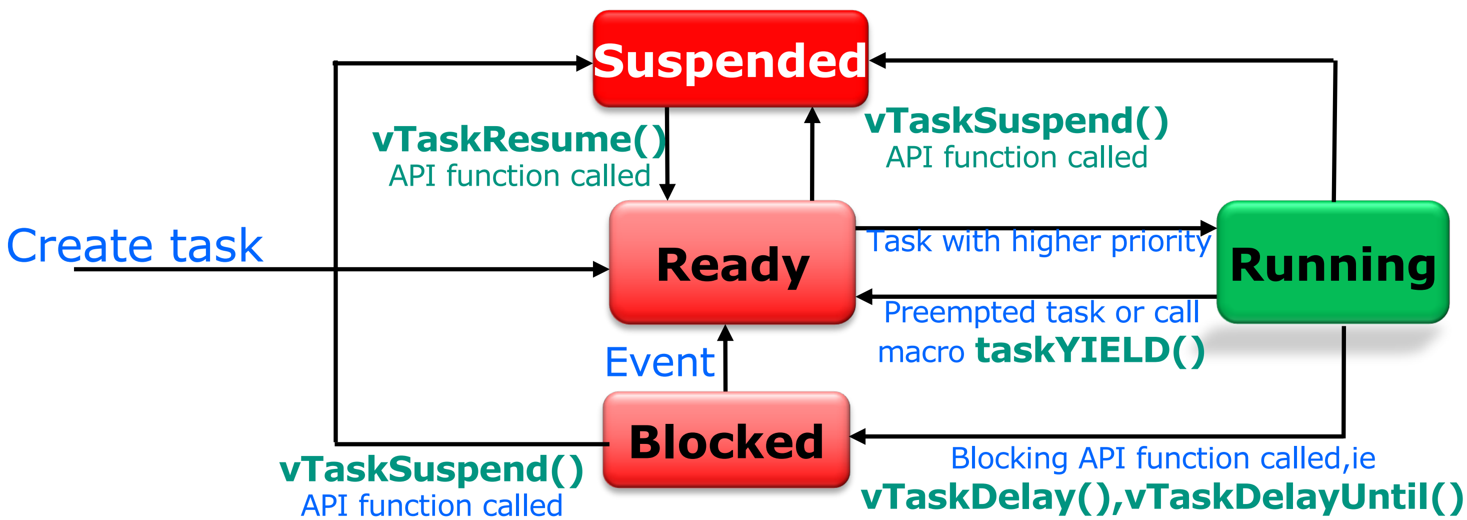
```
for(;;)
{
    // task code
}
```

- Between specified tasks IDLE task is run (if it is possible according to task priority scheme)

McGill

FreeRTOS –Tasks

- Task is a C function **void vTask(void *pvParameters)** which can be used to run any number of separate instances by the function **xTaskCreate()**. In our example we have created 4 different tasks for LED control passing as its argument number of the LED to be controlled and delay parameter.
- Once created task can be in one of the 4 states:
  - **Running** : it is currently using the processor. Only **one task can be in RUN** mode at the moment.
  - **Ready :** able to execute (not blocked or suspended) but a different task of equal or higher priority is already in the Running state.
  - **Blocked** : currently waiting for a temporal (indicated by SysTick when) or external event (indicated by semaphore or queue).
  - **Suspended** : not available for scheduling. Tasks will only enter or exit the suspended state when explicitly commanded to do so through the API calls of the RTOS (**Task_Suspend()** and **Task_Resume()** respectively).



- Each task has its own stack area and its priority (which can be changed)
- To free the RAM memory, recommended to delete the task when no longer in use (function: **xTaskDelete**(task handler or NULL when we would like to delete current task)

McGill

FreeRTOS – semaphores

- In FreeRTOS implementation, semaphores are based on queue mechanism
- There are three types of semaphores in FreeRTOS:
  - Binary – simple on/off mechanism
  - Counting – counts multiple give and multiple take
  - Recursive
- Semaphores are used to synchronize tasks with other events in the system (especially IRQs)
- Waiting for semaphore is equal to wait() procedure, task is in blocked state not taking RTOS time
- Semaphore should be created before usage: i.e **vSemaphoreCreateBinary()**
- Turn on semaphore = give a semaphore can be done from other task or from interrupt subroutine

  xSemaphoreGive() or **xSemaphoreFiveFromISR()**
- Turn off semaphore = take a semaphore can be done from the task

  xSemaphoreTake()