COEN-4720 Embedded Systems

Lecture 12
RTOS

Cris Ababei
Dept. of Electrical and Computer Engineering
MARQUETTE UNIVERSITY
BE THE DIFFERENCE.

Some Preliminary Notes

- We will talk about **processes, tasks**, and **threads**
  - Some say processes are also called tasks; that they mean the same thing
  - Others say treads are tasks (a lot of RTOS terminology uses "task" to refer to thread)
  - Usually, one should be able to tell from the context if "task" is meant to refer to process or thread
- Processes have one relevant characteristic:
  - Memory space of a process is physically insulated from other processes, thanks to features offered by the Memory Management Unit (MMU) inside a general-purpose CPU.
- True embedded architectures (like STM32) do not provide a MMU (only a feature-limited Memory Protection Unit, MPU, is available in some of them).
  - Absence of this unit does not allow to have separated address spaces (not possible to alias physical addresses to logical ones).
  - This means that they can carry out just one single application (one process), which can be eventually split in several tasks as threads sharing the same memory address space.
  - **For this reason, it is preferable to use "task" to mean thread, because in the type of embedded systems we look at, we deal with threads mainly**
- To make things worse/more-confusing, actually, one can talk about embedded systems using Linux like OS running on general-purpose processors (not necessarily MCUs). In that case, one can talk about systems with more than one process. However, in this course we deal with STM32 MCU.

Outline

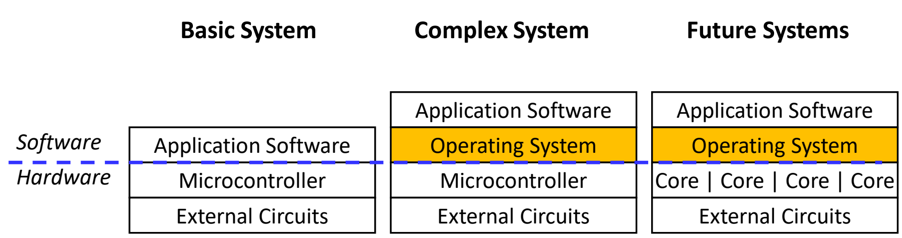
- What is an Operating System?
- Processes or Tasks, Scheduling
- Threads
- RTOS
- Cortex-M

3

3

What is an Operating System (OS)?

- A software layer between the application software and the hardware



4

Embedded Operating System

- Embedded Operating System provides an environment within which firmware pieces, the tasks that make up the embedded application, are executed
- Generally, an OS provides or supports three main control functions:
  1. **Schedule** task execution
  2. **Dispatch** a task to run
  3. Ensure **communication and synchronization** among tasks

5

Real Time Operating System

- Typical embedded system (ES) solves a problem by decomposing it into smaller pieces called **tasks** that work together in an organized way
- System is called **multitasking** system and design aspects include:
  - Exchanging/sharing data between tasks
  - Synchronizing tasks
  - Scheduling tasks
- When the control must ensure that task execution satisfies a set of specified time constraints, the OS is called a **real-time operating system (RTOS)**

6

The Kernel

- The **Kernel** is the smallest portion of the OS that provides these functions

1. Scheduler
   - Determines which task will run and when it will do so
2. Dispatcher
   - Performs the necessary operations to start the task
3. Intertask or interprocess communication
   - Mechanism for exchanging data and information between tasks and processes on the same machines or different ones

7

Services

- The above functions are captured in the following types of services:
  - Process or task management
    - Creation and deletion of user and system processes
  - Memory management
    - Includes tracking and control of which tasks are loaded into memory, monitoring memory, administer dynamic mem
  - I/O System management
    - Interaction with devices done through a special piece of software called a **device driver**
    - The internal side of that software is called a **common calling interface** (an **application programmer's interface, API**)
  - File system management
  - System protection
  - Networking
  - Command interpretation

8

Process (or Task)

- Embedded program (a static entity) = a collection of firmware modules
- When a firmware module is executing, it is called a **process** or **task**
- A task is usually implemented in C by writing a function
- A task or process simply identifies a job that is to be done within an embedded application
- When a process is created, it is allocated a number of resources by the OS, which may include:
  - Process stack
  - Memory address space
  - Registers (through the CPU)
  - A program counter (PC)
  - I/O ports, network connections, file descriptors, etc.
- These resources are generally not shared with other processes

9

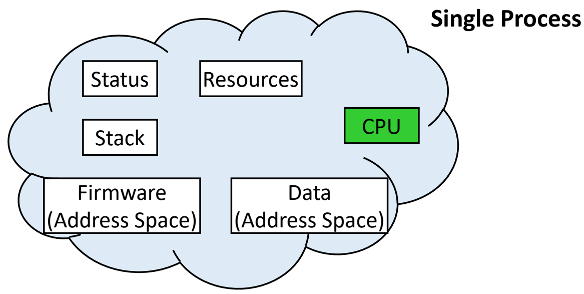
Types of Tasks

- Periodic tasks
  - Found in hard real-time applications
  - Examples: control, every 10 ms; multimedia, every 22.727us;
- Intermittent tasks
  - Found in all types of applications
  - Examples: send email every night at 4am; calibrate a sensor on startup; save all data when power goes down;
- Background tasks
  - A soft real-time or non real-time task
  - Will be accomplished only if CPU time is available
- Complex tasks
  - Found in all types of applications
  - Examples: Microsoft Word; Apache web server;

10

Single Process

- Traditional view of computing: focuses on **program**. One says that the program (or task within the program) runs on the computer
- In embedded applications, we change the p.o.v. to that of microprocessor: CPU is used to execute the firmware. CPU is just another resource
- The time it takes a task to complete is called **execution time**



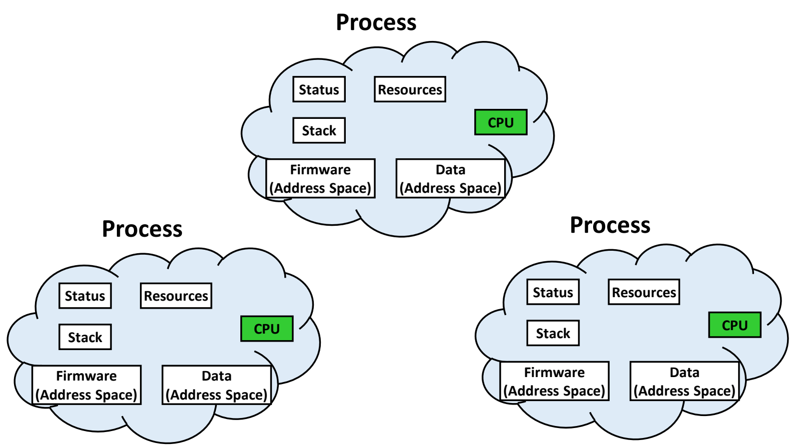
11

Multiple Processes

- If another task is added to the system, potential resource contention problems arise
- This is resolved by carefully managing how the resources are allocated to each task and by controlling how long each can retain the resources
- The main resource, CPU, is given to tasks in a time multiplexed fashion (i.e., time sharing); when done fast enough, it will appear as if both tasks are using it at the same time
- The execution time of the program will be extended, but operation will give the *appearance* of simultaneous execution. Such a scheme is called **multitasking**

12

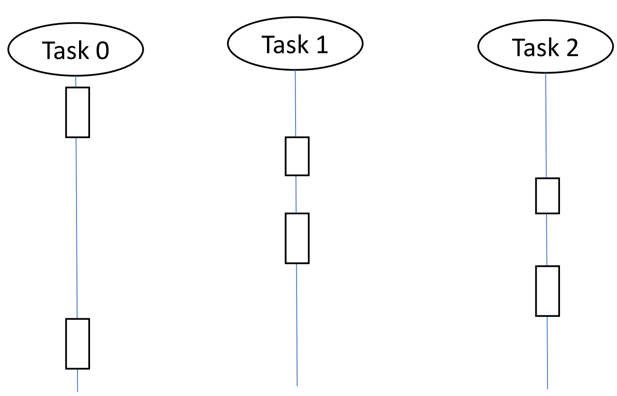
Multiple Processes



13

Sequence Diagram

- At any instant in time, only one process is actively executing; it said to be in **Run** state
- The other processes are in **Ready waiting** state



14

Task Scheduling

- A **schedule** is set up to specify when, under what conditions, and for how long each task will be given the use of the CPU (and other resources)
- The criteria for deciding which task is to run next are collectively called a **scheduling strategy**, which generally falls into three categories:
  - Multiprogramming
    - each task continues until it performs an operation that requires waiting for an external event
  - Real-Time
    - tasks with specified temporal deadlines are guaranteed to complete before those deadlines expire
  - Time sharing
    - running task is required to give up the CPU so that another task may get a turn

15

Task States

- Primarily 4 states
  1. **Running** or Executing
  2. **Ready to Run** (but not running)
  3. **Waiting** (for something other than the CPU)
  4. **Inactive**
- Transition between states is referred to as **context switch**
- Only one task can be Running at a time, unless we use a multicore CPU
- Task waiting for CPU is Ready to Run
- When a task has requested I/O or put itself to sleep, it is Waiting
- An Inactive task is waiting to be allowed into the schedule

16

Address Space of a Process

- When a process is created by the OS, it is given a portion of the physical memory in which to work
- The set of addresses delimiting that code and the data memory, proprietary to each process, is called its address space
- Processes are segregated
  - Supervisor mode
  - User mode – limited to a subset of instructions
- A process may create or spawn **child processes** (each with its own **data** address space, data, status, and stack)
- A process may create multiple **Threads** (each with its own stack and status information)

17

Outline

- What is an Operating System?
- Processes or Tasks, Scheduling
- Threads
- RTOS
- Cortex-M

18

18

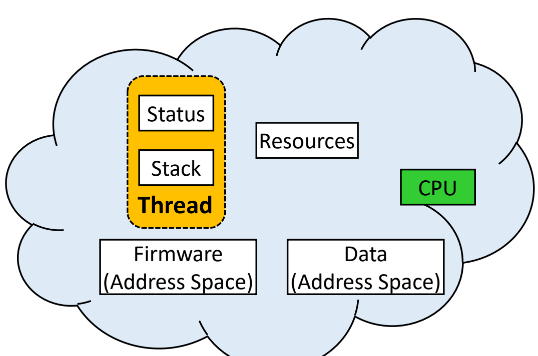
Threads

- A process is characterized by a collection of resources that are utilized to execute a program
- The smallest subset of these resources (a copy of the CPU registers including the PC and a stack) that is necessary for the execution of the program is called a **Thread**
- A thread is a unit of computation with code and context, but no private data
- A thread can be in only one process; a process without a thread can do nothing!

19

Single-process single-thread

- The sequential execution of a set of instructions through a task or process in an embedded application is called a **thread of execution** or **thread of control**
- This model is referred as single-process single-thread



20

Multiple Threads

- During partitioning and functional decomposition of the function intended to be performed by an ES → identify which actions would benefit from parallel execution
  - For example, allocate a sub-job for each type of I/O
- Each of the sub-jobs has its own thread of execution
  - Such a system is called a **single-process multithread** design
- Threads are not independent of each other (unlike processes or tasks)
  - Threads can access any address within the process, including other threads' stacks
- An OS that supports tasks with multiple threads is called a **multithreaded operating system**

21

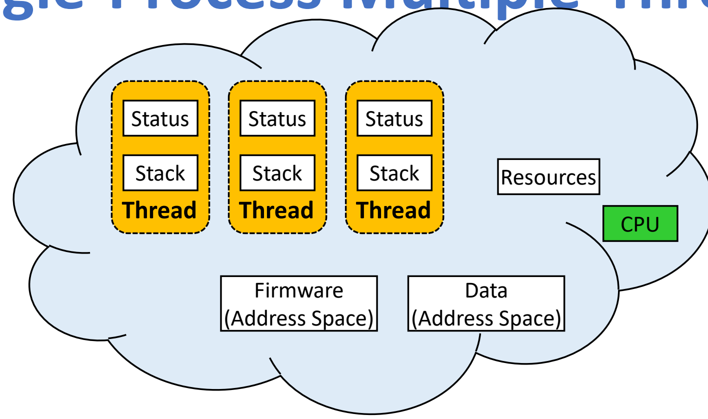
Multithreading

- Multithreading extends the idea of multitasking into single processes, so that you can subdivide specific operations within a single application into individual threads.
- Threads can run in parallel.
- The important trait of threads it that they share the same memory address space.

22

22

Single-Process Multiple-Threads



- All four categories of multitasking operating system:
  - Single process single thread
  - Multiprocess single thread
  - Single process multiple threads
  - Multiprocess multiple threads

23

Processes vs. Threads

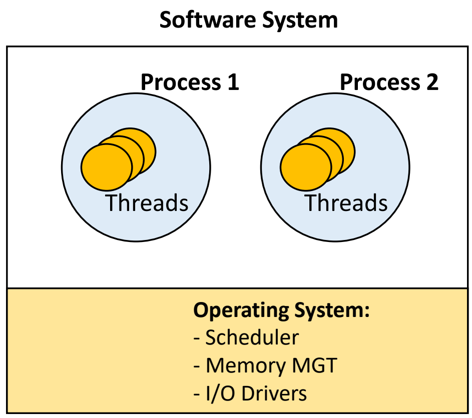
- At the minimum, a process or task needs the following:
  1. The code or firmware, the instructions
     - These are in the memory and have addresses
  2. The data that the code is manipulating
     - The data starts in the memory and may be moved to registers. The data has addresses
  3. CPU and associated physical registers
  4. A stack
  5. Status information

Shared among member Threads

Proprietary to each Thread

24

Example: complete software system with two processes



25

Reentrant Code

- Child processes (and their threads) share the same firmware memory area → two different threads can execute the same function
- Functions using only local variables are inherently reentrant
- Functions using global variables, variables local to the process, variables passed by reference, or shared resources are not reentrant
- Any shared functions must be designed to be reentrant

26

Outline

- What is an Operating System?
- Processes or Tasks, Scheduling
- Threads
- RTOS
- Cortex-M

27

27

Real-Time Operating System (RTOS)

- An OS able to offer/support multitasking (or better, multithreading) while ensuring response within specified (rigid) time constraints, often referred to as deadlines.
- Commonly found in embedded applications
- Key characteristic of an RTOS is that it has **deterministic behavior** = given the same state and the same state of inputs, the next state (and associated outputs) will be the same each time the control algorithm utilized by the system is executed

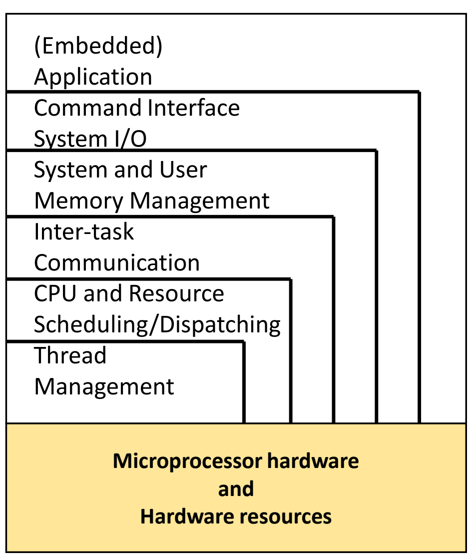
28

Hard vs. Soft Real Time

- Real time
  - A software system with specific speed or response time requirements
- Soft real time
  - Critical tasks have priority over other tasks and retain that priority until complete
  - If performance is not met, performance is considered low
- Hard real time
  - System delays are known or at least bound
  - If deadlines are not met, the system has failed
- Super hard real time
  - Mostly periodic tasks: OS system tick, task compute times, and deadlines are very short

29

Architecture of Operating System



30

Architecture of Operating System

- Organized like the onion model
  - The hierarchy is designed such that each layer uses functions/operations and services of lower layers → increased modularity
- In some architectures, upper layers have access to lower layers through system calls and hardware instructions

31

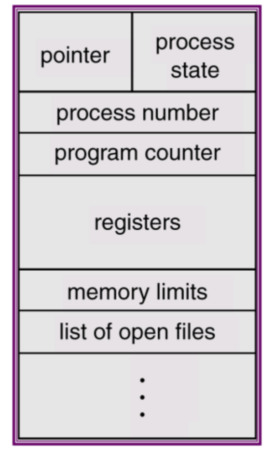
Process or Task Control Block (PCB or TCB)

- An RTOS "orchestrates" the behavior of an application by executing each of the tasks that comprise the design according to a specified schedule
- Each task or process is represented by a **Process or Task Control Block (TCB)**
- A TCB is a **data structure** in the operating system kernel containing the information needed to manage a particular process
- The TCB is "the manifestation of a process in an operating system"

32

Task Control Block (TCB)

- TCB allocation
  - Static: used typically in ES's
  - Dynamic
- A fixed number of TCBs is allocated at system generation time and placed in dormant (unused) state
- When a task is initiated, a TCB is created and the appropriate information is entered
- TCB is placed in Ready state by scheduler
- TCB will be moved to Execute state by dispatcher
- When task terminates, associated TCB is returned to a dormant state
- With fixed number of TCBs, no runtime memory management is necessary



33

Queue or Job Queue

- When a task enters the system, it will be placed into a queue called the Entry Queue or Job Queue
- May be implemented as a linked list or as an array

34

A Simple Kernel

- For the description of several versions of a primitive operating system kernel, read Ch. 11 of:

[BOOK] James K. Peckol, Embedded Systems, A Contemporary Design Tool, John Wiley & Sons, Inc., 2007.

35

Outline

- What is an Operating System?
- Processes or Tasks, Scheduling
- Threads
- RTOS
- Cortex-M

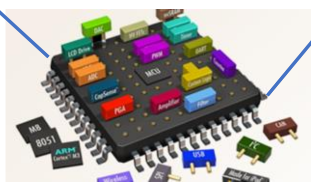
36

36

Recall: Hardware Abstraction Layer (HAL)

User Program

HAL (Defined by ST)
CMSIS (Defined by ARM)



```
while (1)
{
  /* USER CODE END WHILE */
  /* USER CODE BEGIN 3 */
  HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
  HAL_Delay(500);
}
```

```
/* Reset of all peripherals, Initializes the Flash interface and the Systick. */
HAL_Init();

/* Configure the system clock */
SystemClock_Config();
```

37

37

Cortex Microcontroller Software Interface Standard (CMSIS)

- ARM - actively working on a way to standardize the software infrastructure among MCUs vendors
- This is an evolving effort
- Cortex Microcontroller Software Interface Standard (CMSIS) is a vendor-independent hardware abstraction layer for Cortex-M processors
- CMSIS consists of many components, including:
  - CMSIS-CORE: API for the Cortex-M processor core and peripherals
  - CMSIS-Driver: Defines generic peripheral driver interfaces for middleware
  - **CMSIS-RTOS API: Common API for Real-Time Operating Systems**
  - ... many more

38

38

CMSIS-RTOS API

- Common API for Real-Time Operating Systems
- It provides a standardized programming interface
  - Portable to many RTOS and therefore enables software templates, middleware, libraries, and other components which can work across supported RTOS systems.

39

39

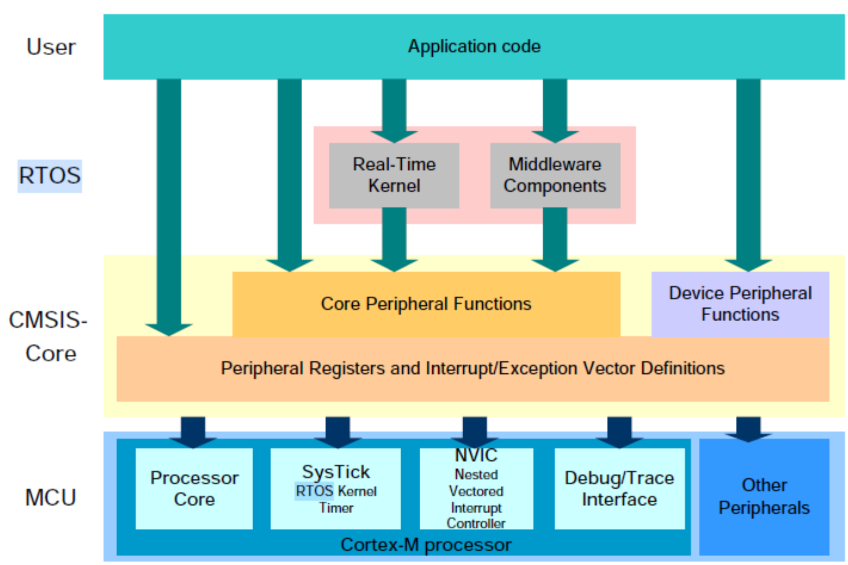


Figure 3.19
CMSIS structure.

Source: [2] Joseph Yiu, The Definitive Guide to ARM Cortex-M0 and Cortex-M0+ Processors, 2nd Ed., 2015. (Book 2). 40

40

Preemptive Multitasking Operating System

- A preemptive multitasking Operating System is a coordinator of physical resources that allows the execution of multiple computing tasks (threads), by assigning a limited quantum time (also called slice time) to each task.
- Every task has a well-defined temporal window, usually about 1ms in embedded systems, during which it performs its activities before it is preempted.
- RTOS kernel decides the execution order of the tasks ready to be executed using a scheduling policy: a scheduler is an algorithm that characterizes the way the OS plans the execution of tasks.

41

41

Context Switch

- A task is "moved" in/out from CPU by a context switch operation.
- A context switch is performed by the OS, with hardware support.
- Cortex-M core takes advantage of a dedicated hardware timer, usually the SysTick: the RTOS uses the periodic interrupt generated on the overflow event to perform the context switch.
- This timer is configured to overflow (or underflow in case of the SysTick, which is a down-counter timer) every 1ms.

42

42

SysTick

- Mainly used as time-base generator for CubeHAL and the RTOS (if used).
- Every RTOS needs a timer to periodically interrupt the execution of current code and to switch to another task.
- STM32 microcontrollers provide SysTick timer, internal to Cortex-M core.
- Even if every other timer may be used to schedule system activities, the presence of a dedicated timer ensures portability among all STM32 families
- NOTE: Even if we do not use an RTOS in our firmware, it is important to keep in mind that ST CubeHAL uses the SysTick timer to perform internal time-related activities (and it assumes that the SysTick timer is configured to generate an interrupt every 1ms).

43

43

Context Switch Impact

- Context switches are usually computationally intensive (so, much of the design of operating systems is to optimize the use of context switches).
- Take special care when you decide to change the underflow frequency of SysTick timer (e.g., by increasing it) - affects the slice time of each individual task, and hence the number of context switches per second.

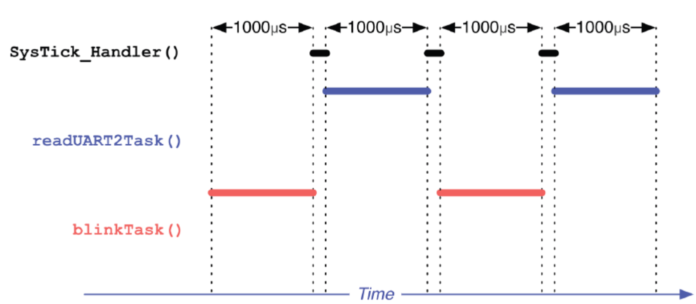


Figure 23.3: The impact of the *context switch* on tasks scheduling

Source: [1] Carmine Noviello, Mastering STM32, Second Edition, 2022. *(Book 1).*

44

44

FreeRTOS

- ST Microelectronics has developed full support to one of the most popular and free and Open Source RTOS: FreeRTOS
- FreeRTOS is possibly the most widespread RTOS for embedded systems on the market today
- CubeMX versions provide full integration of FreeRTOS 10.x for all STM32 microcontrollers
- FreeRTOS was acquired in 2017 by Amazon, and it is now part of the AWS ecosystems

45

45

FreeRTOS

- ST built complete CMSIS-RTOS wrappers around FreeRTOS
  - One for CMSIS-RTOS v1 and one for CMSIS-RTOS v2
- This allows to develop CMSIS-RTOS compliant applications
- Textbook introduces FreeRTOS functionalities using as much as possible the CMSIS-RTOS v2 API

46

46

The FreeRTOS Source Tree

- **CMSIS-RTOS_V2/** folder contains the CMSIS-RTOS v2 compliant layer developed by ST on the top of FreeRTOS

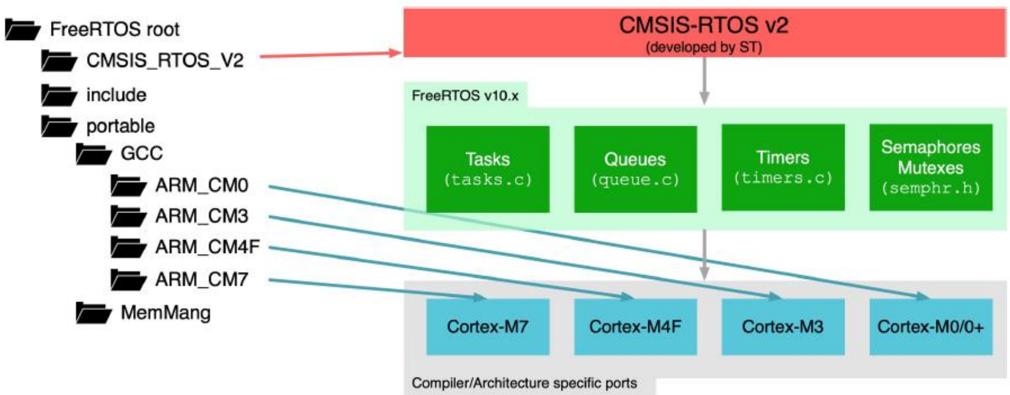


Figure 23.4: The FreeRTOS source tree organization in the CubeHAL

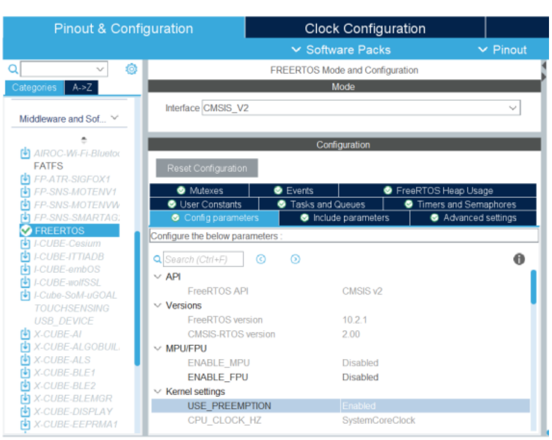
Source: [1] Carmine Noviello, Mastering STM32, Second Edition, 2022. *(Book 1).*

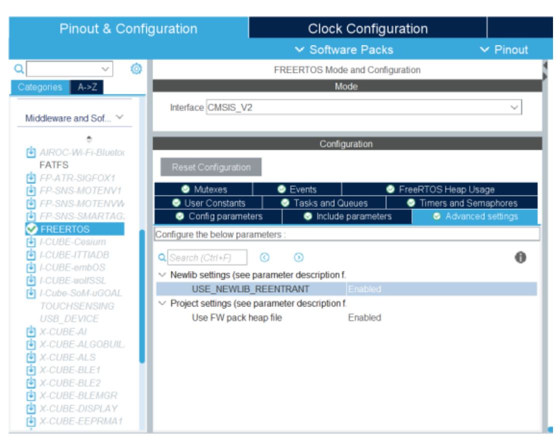
47

47

How to Configure FreeRTOS Using CubeMX

- Enable the FreeRTOS middleware by selecting the wanted CMSIS-RTOS wrapper (V1 or V2) in the Middleware section of the Categories pane.





48

48

Thread Management

- Coding-wise, a thread is nothing more than a C function, which FreeRTOS requires to be defined as follows:

```c
void ThreadFunc(void const *argument) {
  while(1) {
    ...
  }
  osThreadTerminate(osThreadGetId());
}
```

- To start a new thread with the CMSIS-RTOS v2 API (or simply CMSIS-RTOS2) use:

```c
osThreadId_t osThreadNew (osThreadFunc_t func, void *argument, const osThreadAttr_t *attr);
```


49

Example 1 (see demo in class)

```c
#include "nucleo_hal_bsp.h"
#include "cmsis_os.h"

/* Private function prototypes -----------------------------------------------*/
void blinkThread(void *argument);

/* Definitions for blinkThread */
osThreadId_t blinkThreadID;
const osThreadAttr_t blinkThread_attr = {
  .name = "blinkThread",
  .stack_size = 128 * 4, /* In bytes */
  .priority = (osPriority_t) osPriorityNormal,
};

int main(void) {
  HAL_Init();

  Nucleo_BSP_Init();

  /* Init scheduler */
  osKernelInitialize();

  /* Creation of blinkThread */
  blinkThreadID = osThreadNew(blinkThread, NULL, &blinkThread_attr);

  /* Start scheduler */
  osKernelStart();

  /* We should never get here as control is now taken by the scheduler */
  while (1);
}

void blinkThread(void *argument) {
  while(1) {
    HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
    osDelay(500);
  }
}
```


50

Thread States

- A thread can have two major execution states: Running and Not Running.
- On a singlecore architecture, only one thread can be in running state at any time

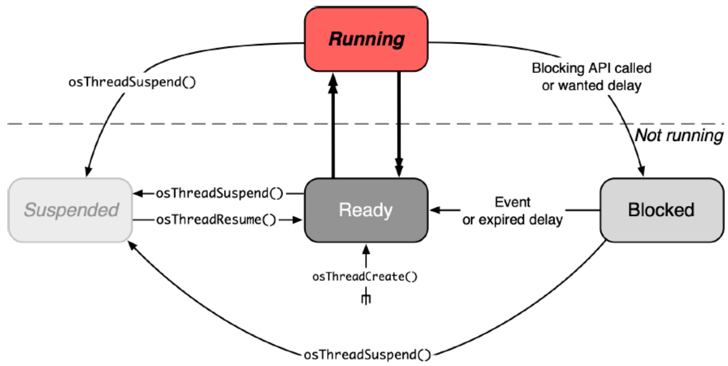


Figure 23.7: The possible states of a thread in FreeRTOS

Source: [1] Carmine Noviello, Mastering STM32, Second Edition, 2022. *(Book 1).*

51

51

Thread Priorities and Scheduling Policies

- Priorities impact the scheduling algorithm - allowing to alter the execution order in case a thread with a higher priority turns in ready state
- Priorities are a fundamental aspect of RTOSes and provide the foundation blocks to achieve short responses to deadlines.
- Important to underline: <u>thread priority is not related to priority of IRQs</u>.
- FreeRTOS has a user-defined priority system, which gives a great degree of flexibility in defining priorities.

52

52

Table 23.1: The fixed priorities defined in the CMSIS-RTOS2 specification

Priority level	Value	Description
osPriorityNone	0	No priority (not initialized).
osPriorityIdle	1	Priority: *idle* - Reserved for Idle thread.
osPriorityLow	8	Priority: *low*
osPriorityLow[1..7]	8+[1..7]	Priority: low + 1..7
osPriorityBelowNormal	16	Priority: *below normal*
osPriorityBelowNormal[1..7]	16+1	Priority: below normal + 1..7
osPriorityNormal	24	Priority: *normal*
osPriorityNormal[1..7]	24+1	Priority: normal + 1..7
osPriorityAboveNormal	32	Priority: *above normal*
osPriorityAboveNormal[1..7]	32+1	Priority: above normal + 1
osPriorityHigh	40	Priority: *high*
osPriorityHigh[1..7]	40+1	Priority: high + 1
osPriorityRealtime	48	Priority: *realtime*
osPriorityRealtime[1..7]	48+1	Priority: realtime + 1
osPriorityISR	56	Priority: *ISR* - Reserved for ISR deferred thread
osPriorityError	-1	System cannot determine priority or illegal priority
osPriorityReserved	0x7FFFFFFF	Prevents enum down-size compiler optimization

Source: [1] Carmine Noviello, Mastering STM32, Second Edition, 2022. *(Book 1).*

53

53

Scheduling Algorithms

- FreeRTOS provides three different scheduling algorithms
  - Selected by the right combination of the symbolic constants configUSE_PREEMPTION and configUSE_TIME_SLICING, both defined in Core/Inc/FreeRTOSConfig.h file.

Table 23.2: How to select the wanted scheduling policy in FreeRTOS

configUSE_PREEMPTION	configUSE_TIME_SLICING	Scheduling algorithm
1	1 or undefined	*Prioritized preemptive scheduling with time slicing*
1	0	*Prioritized preemptive scheduling without time slicing*
0	any value	*Cooperative scheduling*

Source: [1] Carmine Noviello, Mastering STM32, Second Edition, 2022. *(Book 1).*

54

54

1.Prioritized preemptive scheduling with time slicing

- Every thread has a fixed priority, which is assigned during its creation
  - But, programmer is free to reassign a different priority
  - Scheduler will immediately preempt a running thread if one with a higher priority becomes ready to be executed
  - Being preempted means being involuntarily (without explicitly yielding or blocking) moved out of the running state into the ready state
- Time slicing (aka quantum time) is used to share CPU processing time between threads with the same priority
  - When a thread "consumes" its time slice, the scheduler will select the next running thread in the scheduling list (if available)

55

55

2.Prioritized preemptive scheduling without time slicing

- Similar to the previous algo
  - Except that once a thread enters in running state, it will leave the CPU only on a voluntary basis (by blocking, stopping or yielding) or if a higher priority thread enters in ready state.
- This algorithm minimizes a lot the impact of the context switch on the overall performance
  - Because number of switches is dramatically reduced.
- However, a bad designed thread may monopolize the CPU, causing unpredictable behavior

56

56

3.Cooperative Scheduling

- A thread will leave the CPU only on a voluntary basis (by blocking, stopping or yielding).
- Even if a higher priority thread becomes ready, the OS will never preempt the current thread, and it will reschedule it again in case of an external interrupt.
- Gives all responsibility to the programmer – who must carefully design the threads as if designing a firmware without using an RTOS!

57

57

Example 2 (see demo in class)

- Two threads
  - One that blinks the LD2 LED
  - One that constantly prints on the UART2 a message.
  - UARTThread() is created with a priority higher than the blinkThread()
- Running this example, you can see that the LD2 LED never blinks.
- This happens because UARTThread() is designed to continuously do something and when its slice time expires, it is still in ready state and, having a higher priority, it is rescheduled for execution.
- This proves that priorities must be used carefully to prevent other processes from starving.

58

58

Credits, References

- Ch.23 of: Carmine Noviello, Mastering STM32, Second Edition, 2022.
- Ch.3,10,20 of: Joseph Yiu, The Definitive Guide to ARM Cortex-M0 and Cortex-M0+ Processors, 2nd Ed., 2015. (Book 2)
- Ch.11,12 of: James K. Peckol, Embedded Systems, A Contemporary Design Tool, John Wiley & Sons, Inc., 2007.
- Ch.3,4 of: Jonathan W. Valvano, Real-Time Operating Systems for ARM Cortex-M Microcontrollers, 2012.

59