Mechatronics Engineering

Introduction to Multitasking and FreeRTOS

GUC German University in Cairo

Prof. Ayman A. El-Badawy
Department of Mechatronics Engineering
Faculty of Engineering and Material Science

Tutorial Contents

- **Introduction to Multitasking**
- Multitasking Concepts
  - Concurrency
  - Context Switching
  - Real-time Systems
- Multitasking
  - Cooperative Multitasking
  - Preemptive Multitasking
- FreeRTOS Real-time Kernel
  - Task Management
  - AVR GCC Example Program

Introduction to Multitasking

- Example of periodic tasks performed in a single program:

```
int main(void)
{
    initialization_function();
    while(1) // Infinite main loop
    {
        task_1();
        if(task_2_should_run())
            task_2();
        task_3();
        wait_for_end_of_period();
        /* Sleep until the start of the next period (For example, wait for timer compare match) */
    }
}
```

•When multiple operations must be performed on a microcontroller at widely varying times, writing a **single large sequential program** to perform these tasks can result in a complex and disorganized program.

•In addition, modifying the program can be difficult and may result in errors if the CPU is heavily loaded.

•As the complexity of embedded real-time applications increases, a more efficient strategy for managing processor time is required.

Introduction to Multitasking

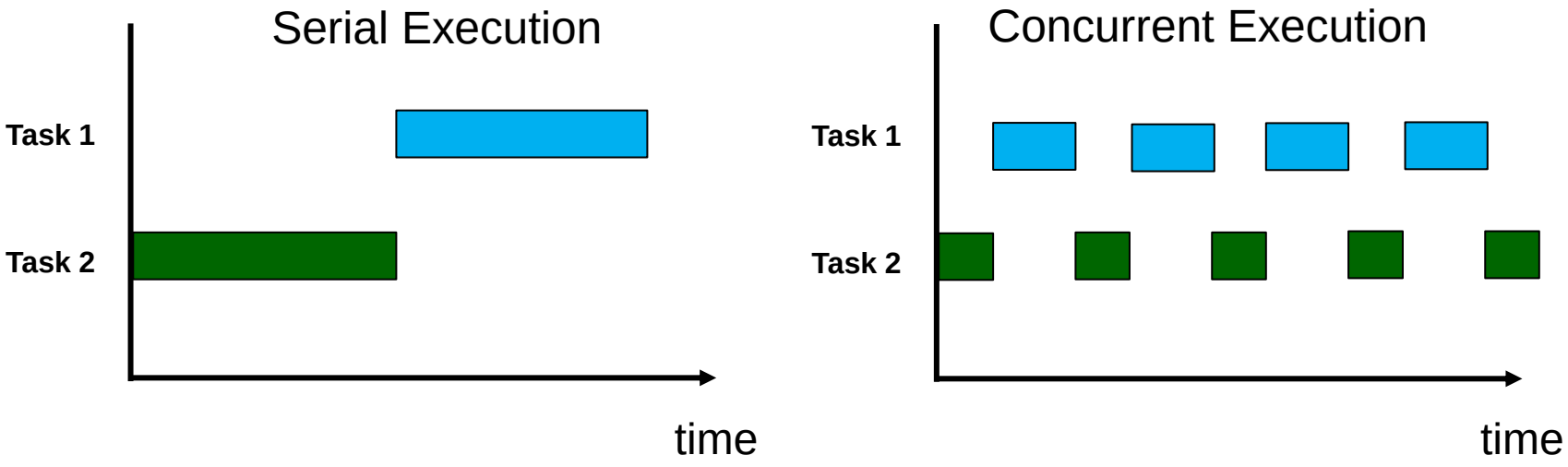
- Accordingly, the concept of **multitasking** should be introduced.
- **Multitasking**: It is the simultaneous execution of multiple tasks.
- We can break the program into multiple tasks in order to manage the program's timing efficiently and to reduce the system's complexity.
- Multitasking is based on two main concepts:
  - **The process:** It is a single execution of a program. E.g., two different runs of the same program are considered two different processes.
  - **The OS/kernel:** It provides mechanisms for switching execution between different processes.
- The terms **process** and **task** are used interchangeably in this course.

Tutorial Contents

- Introduction to Multitasking
- **Multitasking Concepts**
  - **Concurrency**
  - Context Switching
  - Real-time Systems
- Multitasking
  - Cooperative Multitasking
  - Preemptive Multitasking
- FreeRTOS Real-time Kernel
  - Task Management
  - AVR GCC Example Program

Concurrency

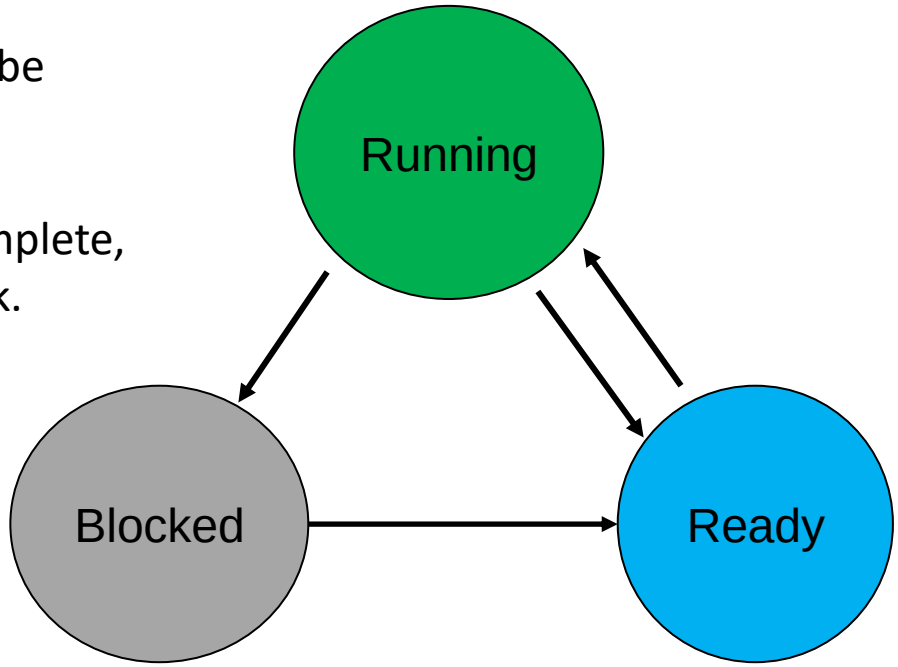
- In a single-processor system, **only one process** can be in a running state at a time, while other processes are in a non-running state (e.g., waiting).
- Processor can switch between multiple processes, such that they appear to be running simultaneously. Processes are said to be running **concurrently**.
- Different priorities can be assigned to each process. Accordingly, the **scheduler** determines which process should be running.



Concurrency

- Simplified state diagram of a task:

  - **Running State**: the task is currently being executed (only one task can be in this state).
  - **Ready State**: Task ready and waiting to be executed.
  - **Blocked**: Waiting for an event to be complete, e.g., waiting for input from another task.



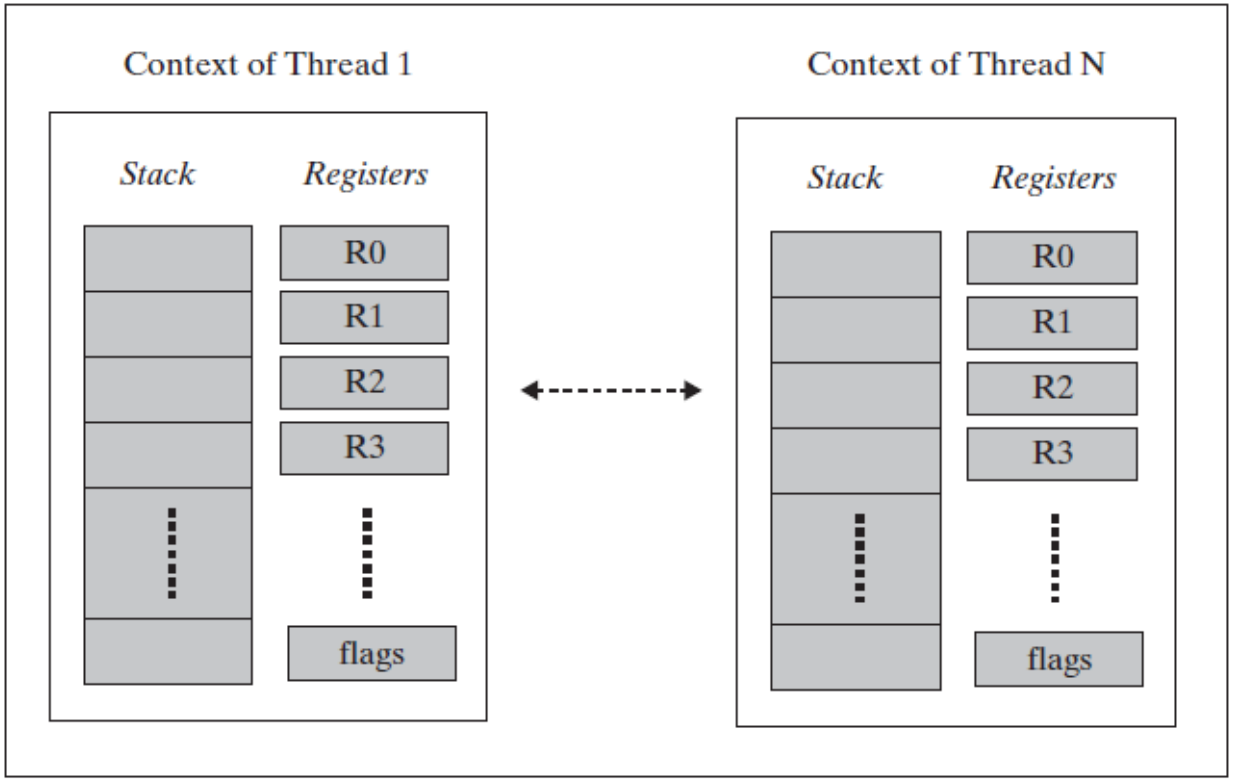
Tutorial Contents

- Introduction to Multitasking
- **Multitasking Concepts**
  - Concurrency
  - **Context Switching**
  - Real-time Systems
- Multitasking
  - Cooperative Multitasking
  - Preemptive Multitasking
- FreeRTOS Real-time Kernel
  - Task Management
  - AVR GCC Example Program

Context Switching

- **Task execution context:** During task execution, the microcontroller utilizes some registers, RAM and ROM like any program. These resources, including all CPU registers, the stack, etc., define the **context** of a process

- Each process maintains its own stack and register contents (context):

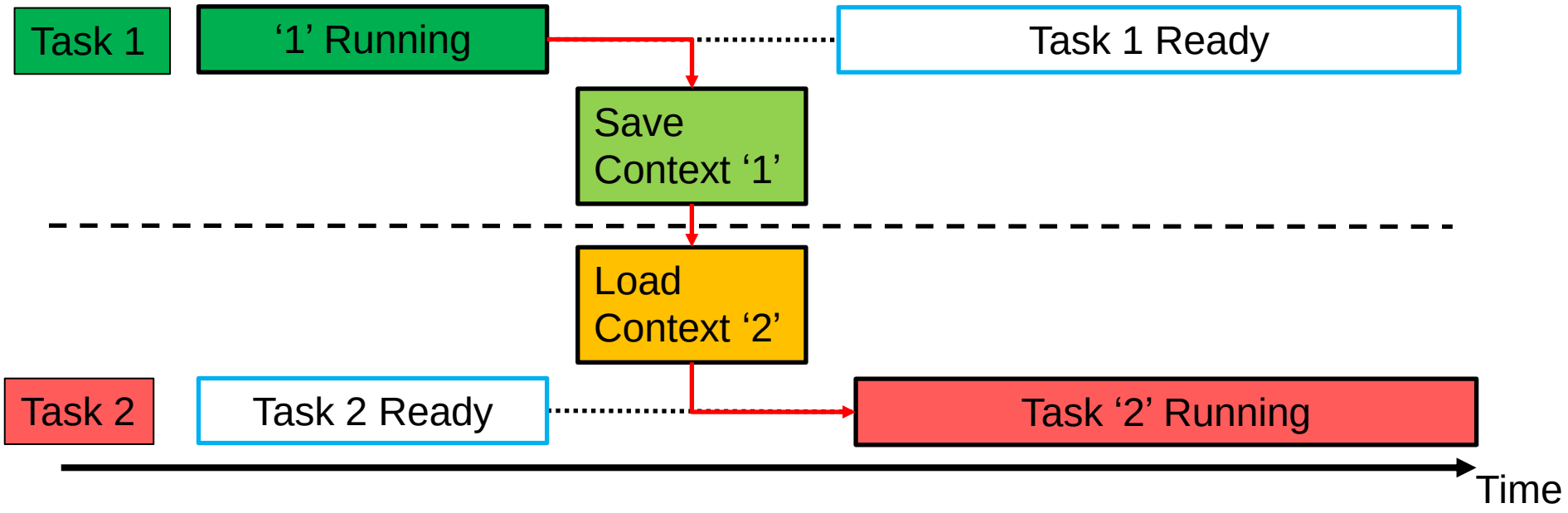


Context Switching

- Before switching between two tasks, for example, from task 1 to task 2, a **context switch** from ‘1’ to ‘2’ first saves the context of ‘1’ and loads the context ‘2’ (This process is usually handled by the kernel/OS).

- Example switching from task ‘1’ to task ‘2’:

Execution marked by: →



Tutorial Contents

- Introduction to Multitasking
- **Multitasking Concepts**
  - Concurrency
  - Context Switching
  - **Real-time Systems**
- Multitasking
  - Cooperative Multitasking
  - Preemptive Multitasking
- FreeRTOS Real-time Kernel
  - Task Management
  - AVR GCC Example Program

Real-time Systems

- In embedded systems, one of the major design requirement is to minimize the **response time** of the program. In other words, the processes should meet their **deadlines**.

- **Definitions**:

  - **Soft-deadline requirement:** It is a requirement that defines a deadline, however, failing to meet this deadline will not result in failure of the system. For example, delayed display screen update.

  - **Hard-deadline requirement:** A deadline which if missed, system failure may occur. For example, car airbag fails to respond in time.

  - **Real-time kernel/scheduler:** It is the part of the **Real-time Operating System (RTOS)** which determines what process should be running. It provides mechanisms to switch between processes. In addition, it handles communication, synchronization and priorities.

- The **RTOS** provide facilities required to satisfy real-time requirements.

Tutorial Contents

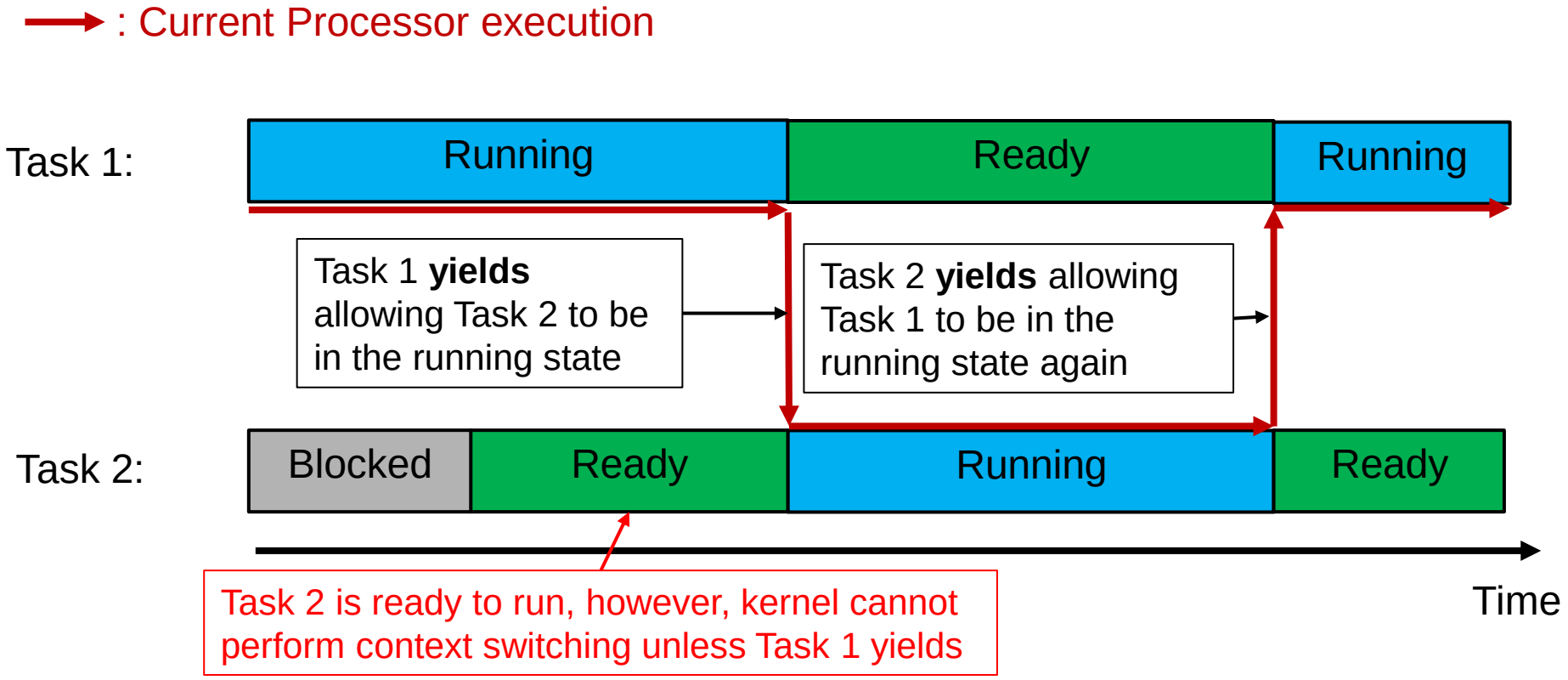
- Introduction to Multitasking
- Multitasking Concepts
  - Concurrency
  - Context Switching
  - Real-time Systems
- **Multitasking**
  - **Cooperative Multitasking**
  - Preemptive Multitasking
- FreeRTOS Real-time Kernel
  - Task Management
  - AVR GCC Example Program

Cooperative Multitasking

- In **cooperative multitasking**, tasks are responsible for returning control to the kernel/scheduler.
- A task must call a specific kernel function, in order to allow context switching to other tasks. Calling this special kernel function is known as a "**yield**".
- Problems with Cooperative Multitasking:
  1. If a currently running process does not call the yield routine frequently other processes may not be allowed to execute for long time.
  2. It is difficult to predict the worst-case response time of cooperative multitasking system.

Cooperative Multitasking

- **Cooperative context switch example:** Two tasks ‘1’ and ‘2’ are being processed by a single processor. The context switching is performed by a cooperative kernel.



Prof. Ayman A. El-Badawy
Department of Mechatronics Engineering
Faculty of Engineering and Material Science

Tutorial Contents

- Introduction to Multitasking
- Multitasking Concepts
  - Concurrency
  - Context Switching
  - Real-time Systems
- **Multitasking**
  - Cooperative Multitasking
  - **Preemptive Multitasking**
- FreeRTOS Real-time Kernel
  - Task Management
  - AVR GCC Example Program

Preemptive Multitasking

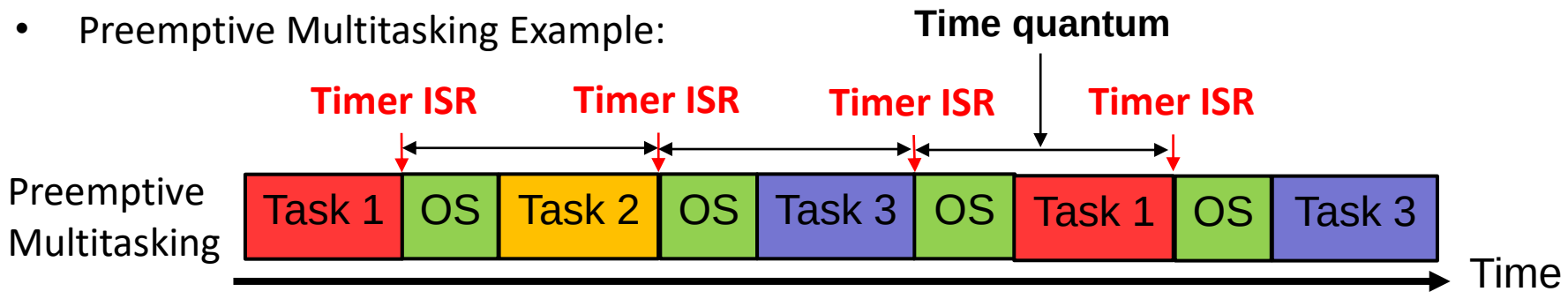
- Usually kernel/scheduler is **activated periodically** by a timer ISR. E.g., on every timer overflow, the kernel/OS regains control over the CPU.

- The length of the timer ISR period is known as the **time quantum**, which is the smallest time period in which the kernel can regain control over the CPU. The kernel should maintain a variable **clock ticks** that increments by 1 whenever the timer ISR is called.

- When the kernel regains control, it determines whether a context switch to another task is required or not. Therefore, it is not required to call the yield routine by the running task.

- A reliable method to achieve real-time performance is to use a **preemptive OS** and to assign **priorities** to each task, in order to determine the order of tasks execution.

- Preemptive Multitasking Example:



Tutorial Contents

- Introduction to Multitasking
- Multitasking Concepts
  - Concurrency
  - Context Switching
  - Real-time Systems
- Multitasking
  - Cooperative Multitasking
  - Preemptive Multitasking
- **FreeRTOS Real-time Kernel**
  - **Task Management**
  - AVR GCC Example Program

FreeRTOS Real-time Kernel

- **FreeRTOS Overview:**
  - Open-source light weight real-time kernel for microcontrollers and small microprocessors.
  - Provides mechanisms for both preemptive and cooperative multitasking.
  - Provides mechanisms for **inter-process communication** (communication between different tasks and ISRs).
  - Flexible task priority assignment and task notification mechanism.
  - Supports various processor architectures and compilers, including the **AVR-GCC** compiler and some MCUs from the **AVR ATMega** family and **Arduino**.
  - FreeRTOS: https://www.freertos.org/

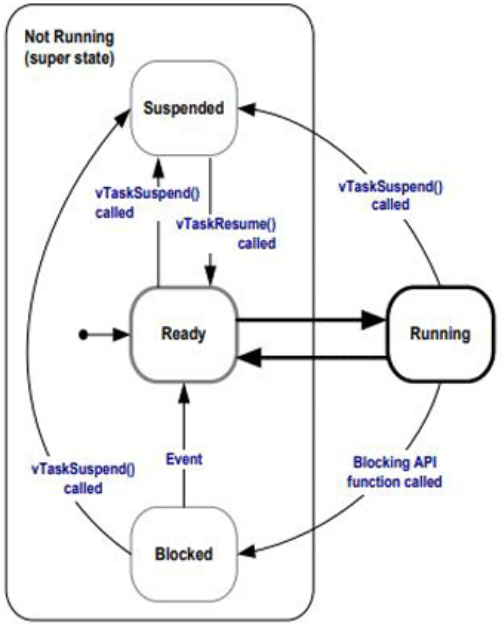
FreeRTOS Task Management

- In FreeRTOS, a task definition is a C function with a special prototype. To define a task, a C function that includes its execution code should be written as follows:

```
void vTaskCode( void * pvParameters )
{
    // Code that define the task initialization here
    while(1)
    {
        // Code that repeats forever defining the task's operation
    }
}
```

- A task definition must have return type `void` and parameter of type `void *`

- A task definition can be used to create multiple instances that run the same function `vTaskCode`. Each instance is considered separate with its own stack and its own copy of automatic variables.



FreeRTOS Task Management

- The Application Programming Interface (API) function `xTaskCreate()` can be used to create a new instance of a task.

- This function returns **pdPASS** if the task instance was created successfully.

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode, const char * pcName, uint16_t
usStackDepth, void *pvParameters, UBaseType_t uxPriority, TaskHandle_t
*pxCreatedTask );
```

- `pvTaskCode`: pointer to task definition function. For example, `vTaskCode`
- `pcName`: a name for the task instance (used for debugging).
- `usStackDepth`: The number of words (RAM memory locations) allocated in the stack for this task instance.
- `pvParameters`: parameters passed to the created task instance as the task parameters.
- `uxPriority`: The priority of the task instance. Kernel will choose the highest priority task that is ready to run to execute.
- `pxCreatedTask`: Used to pass a handle to the created task. It is optional and can be set to `NULL`

FreeRTOS Task Management

- Example to create two task instances from a task definition vTaskCode():

```
xTaskCreate(vTaskCode, "Test Task 1", 128, NULL,1, NULL );
xTaskCreate(vTaskCode, "Test Task 2", 128, NULL,1, NULL );
```

- Previously used delay functions such as delay_ms(duration) generated the delay by letting the CPU executing dummy operations until the duration ends. This is a waste of CPU time.

`vTaskDelay:` This function allows the task calling it to block for the specified period, with a resolution of one kernel tick.

➢ `void vTaskDelay( TickType_t xTicksToDelay );`

- ❖ `xTicksToDelay:` The number of **kernel ticks** for which the task will be blocked. This number of ticks is counted starting from the function call time.
- ❖ Every time a timer ISR is invoked, the **kernel ticks** variable is incremented by one.

- Macro `pdMS_TO_TICKS( duration_ms )` can be used to convert duration_ms to clock ticks.

- Example call to block a task for 30 ms:

```
vTaskDelay( pdMS_TO_TICKS(30) ); // blocks this task for 30 ms
```

FreeRTOS Task Management

- Example debouncing with API `vTaskDelay` function:

```
volatile char buttonPressed;
void vTaskReadButton(void * pvParameters )
{
    buttonPressed=0; // state of button connected to PD2
    DDRD &= 0b11111011;
    for(;;)
    {

        if(!buttonPressed && (PIND & 0b00000100))//button is pressed now
        {
            vTaskDelay( pdMS_TO_TICKS(30) ); // blocks this task for 30 ms (debouncing)
            if(PIND & 0b00000100)
                buttonPressed = 1;
        }else if(buttonPressed && !(PIND & 0b00000100)) //button is released now
        {
            vTaskDelay( pdMS_TO_TICKS(30) ); // blocks this task for 30 ms (debouncing)
            if(!(PIND & 0b00000100))
                buttonPressed = 0;
        }
    }
}
```

FreeRTOS Task Management

- `vTaskDelay()` specifies a time (relative to function call time), at which the task will be unblocked starting from the function call, therefore, it does not control the frequency of execution of the calling task.

- A more accurate way of controlling periodic execution of tasks is to use `vTaskDelayUntil()`.

- `vTaskDelayUntil(TickType_t *pxPreviousWakeTime, const TickType_t xTimeIncrement );`
  This API function specifies the time at which the task instance will unblock. The unblock time is specified relative to the previous unblock time **`pxPreviousWakeTime`**. Therefore, periodic execution of the calling task instance can be achieved.

  - `pxPreviousWakeTime`: Pointer to variable that holds the time in ticks at which this task was last unblocked.

  - `xTimeIncrement`: The cycle time period. The number of ticks after which the task will unblock relative to *pxPreviousWakeTime. The task instance will unblock at time (*pxPreviousWakeTime + pxPreviousWakeTime).

FreeRTOS Task Management

- Example code of `vTaskDelayUntil()` for periodic execution of a task.

```
void vTaskPeriodicLEDToggle(void * pvParameters )
{

    TickType_t xLastWakeTime ;// variable that stores last time task unblocked
    const TickType_t xPeriod = 100;// task should execute every 10 kernel clock ticks

    DDRD|=0b00000001; // initialize PD0 as output to toggle LED connected to it.

    //Initialize the last wake time with the current time.
    xLastWakeTime = xTaskGetTickCount();// function returns current kernel tick count

    for(;;)
    {
        PORTD^=0b000000001; //toggle LED
        vTaskDelayUntil(&xLastWakeTime, xPeriod);/* block the task until the period
                                                 ,specified in ticks, passes*/
    }
}
```

Tutorial Contents

- Introduction to Multitasking
- Multitasking Concepts
  - Concurrency
  - Context Switching
  - Real-time Systems
- Multitasking
  - Cooperative Multitasking
  - Preemptive Multitasking
- **FreeRTOS Real-time Kernel**
  - Task Management
  - **AVR GCC Example Program**

FreeRTOS AVR GCC Example

- FreeRTOS has many ports. A port is a version of the FreeRTOS that is compatible with a specific **hardware device** and a specific **compiler**.

- To start development using FreeRTOS, it is recommended to use the demo project for the port of interest, remove the unnecessary demo functions, and add the application specific functions.

- It is important to note that the memory of the microcontroller used is limited in size, therefore, the application designer should be careful in the choice of stack sizes for the tasks.

FreeRTOS Example

- The file **FreeRTOSConfig.h** contains important configuration flags that configure the kernel.

- Some of the important flags:

  configCPU_CLOCK_HZ: define the MCU clock speed in Hz

  configUSE_PREEMPTION: set this flag to 1 for preemptive multitasking

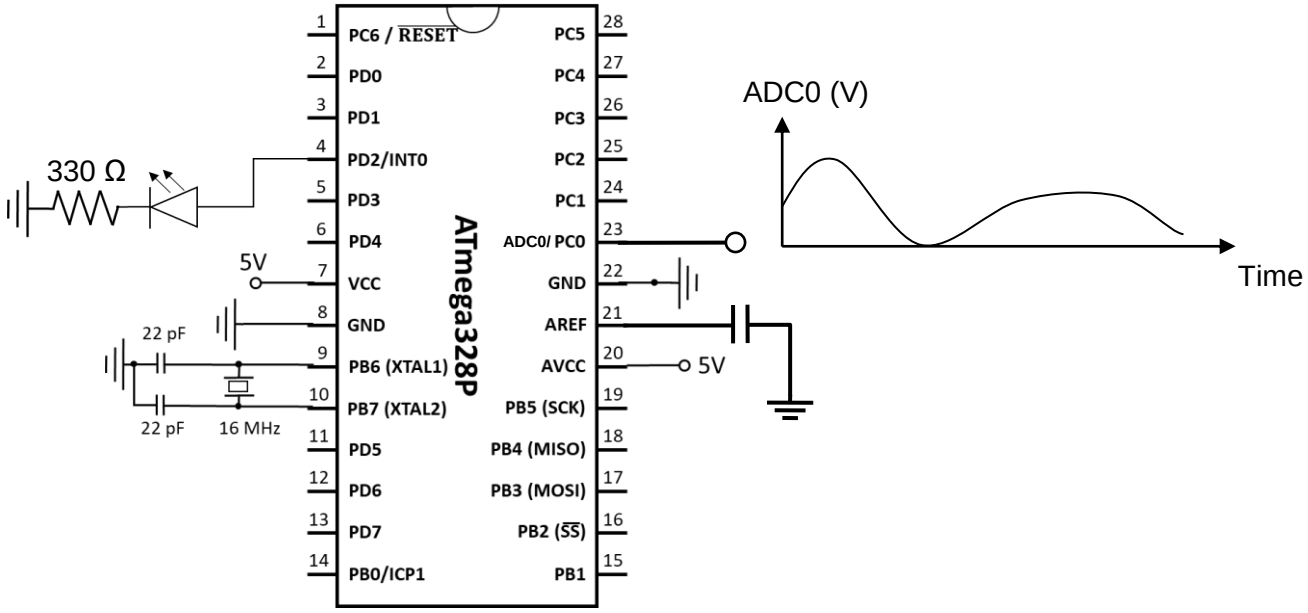
  configMINIMAL_STACK_SIZE: minimum stack size used by a task

  configTOTAL_HEAP_SIZE: total heap size for dynamic memory allocation

  configCHECK_FOR_STACK_OVERFLOW: detection of stack overflow in any of the running tasks

FreeRTOS Example

- **Problem 1:**
  - ❖ Develop a C program on the ATmega328P **to toggle an LED** connected to **PD2** every **0.5 seconds** while **reading the ADC value** on the pin **ADC0** every **60 milliseconds**.
  - ❖ Use FreeRTOS to define two tasks, one to read the analog voltage on ADC0, while the other task should blink the LED connected to PD2.



FreeRTOS Code

- Define the task to toggle the LED:

```
#include <atmel_start.h> // or just include <avr/io.h> and configure the peripherals in code
/* Scheduler include files. */
#include "FreeRTOS.h"
#include "task.h"
/* Tests. These test files are just used by the demo and are not necessary to run FreeRTOS */
#include "regtest.h“
#include "integer.h"
#include "PollQ.h"
#include "partest.h"
/* Priority definitions for most of the tasks in the demo application. */
#define mainCHECK_TASK_PRIORITY( tskIDLE_PRIORITY + 3 ) // checks FreeRTOS validity (not necessary to run freeRTOS)
#define LED_TASK_PRIORITY( tskIDLE_PRIORITY + 2 )
#define ADC_TASK_PRIORITY( tskIDLE_PRIORITY + 2 )
volatile int ADC_Result;

void LEDTask( void* pvParameters){

    TickType_t xLastWakeTime;
    const TickType_t xPeriod = pdMS_TO_TICKS(500) ;// task should execute every 500 ms
    DDRD|=0b00000100; // initialize PD2 as output to toggle LED connected to it
    //Initialize the last wake time with the current time.
    xLastWakeTime = xTaskGetTickCount();// function returns current kernel tick count

    for(;;)
    {
        PORTD^=0b000000100; //toggle LED
        vTaskDelayUntil(&xLastWakeTime, xPeriod);/* block the task until the period
                                                    ,specified in ticks, passes*/
    }
}
```

FreeRTOS Code

- Define the task to read the ADC channel:

```
void ADCTask( void* pvParameters){
    TickType_t xLastWakeTime;
    const TickType_t xPeriod = pdMS_TO_TICKS(60) ;// task should execute every 60 ms

    // Initialize ADC
    DDRC&=0b11111110;
    ADMUX&=0b00100000;
    ADCSRA&=0b10000111;
    DIDR0=0b11111111;

    //Initialize the last wake time with the current time.
    xLastWakeTime = xTaskGetTickCount();// function returns current kernel tick count
    for(;;)
    {
        ADCSRA|=0b010000000; //Start ADC conversion
        vTaskDelay(1); //Allow any other task to execute during the ADC conversion
        while(ADCSRA & 0b01000000); // wait for end of conversion
        ADC_Result = ADC; // read ADC conversion result
        vTaskDelayUntil(&xLastWakeTime, xPeriod);/* block the task until the period
                                                 ,specified in ticks, passes*/
    }
}
```

FreeRTOS Example

- Button press triggered LED toggle task:

```
int main(void){

    /* Initializes MCU, drivers and middleware.
    This is generated from Atmel START project. (not necessary if peripherals initialized in code) */
    atmel_start_init();

    // a standard register test to check the validity of context switching (not necessary to run FreeRTOS)
    vStartRegTestTasks();
 /* Create the tasks defined. */
    xTaskCreate(LEDTask, "LED", 128, NULL, LED_TASK_PRIORITY, NULL );
    xTaskCreate(ADCTask, "ADC", 128, NULL, ADC_TASK_PRIORITY, NULL);
    /* Create the error checking task of the demo. */
    xTaskCreate( vErrorChecks, "Check", configMINIMAL_STACK_SIZE, NULL, mainCHECK_TASK_PRIORITY, NULL );

    vTaskStartScheduler(); // start scheduler
}
```

- Additional functions required by the demo include:

  `vApplicationStackOverflowHook, ErrorChecks,`
  `vApplicationGetIdleTaskMemory, vApplicationGetTimerTaskMemory`