

Embedded Software Design Techniques

Real time operating systems and
application development

Tsuyoshi Isshiki

Dept. Communications and Integrated Systems

Tokyo Institute of Technology

TAIST ICTES

Conduct by Asst. Prof. Dr. Kanjanapan Sukvichai

Lecture Outline

- Embedded software overview
 - What are “embedded systems” and “embedded software”?
- C programming 1: C language overview
 - Function, declaration, statement, expression
 - Data types, data structure, pointers and pointer dereferences
- C programming 2: algorithm complexity, program execution model
 - Bubble sort vs quick sort
 - Stack memory and program execution
- C programming 3: programming techniques in image processing
 - Dynamic memory allocation, image array implementation
 - Greyscaling, filtering, binarization, color quantization, dithering
- C programming 4: programming complex applications
 - Program development steps (ex. Huffman coding)
 - Binary tree construction, tree traversal
 - Bitstream handling
- Real time operating systems and application development
 - RTOS services, kernels
 - Context switching, task scheduling
 - Multi-task programming model

Agenda: Embedded Software

- What kind of hardware environment does softwares inside the embedded system operate?
- What kind of key concepts must we understand to write embedded software?

Embedded Systems Requirement

- Low cost
 - Application-specific system optimizations
- Low power
 - Mobile phone, PDA, notebook PC
- High reliability/security
 - Life-critical systems: automotive, transportation, power plants, building control
- Real-time
 - Control systems
- Time-to-market
 - Adapting to market needs
- Operating conditions
 - Temperature, vibration
- Hardware constraints
 - Size, weight
- Maintainability
 - Fault recovery/repair, field upgrades

Embedded Software

- Embedded system

- Definition:

- “A special-purpose computer system designed to perform a set of dedicated functions” (source: Wikipedia)

- Attributes:

- Real time computing constraints
 - “Embedded” as a complete device with hardware and mechanical parts

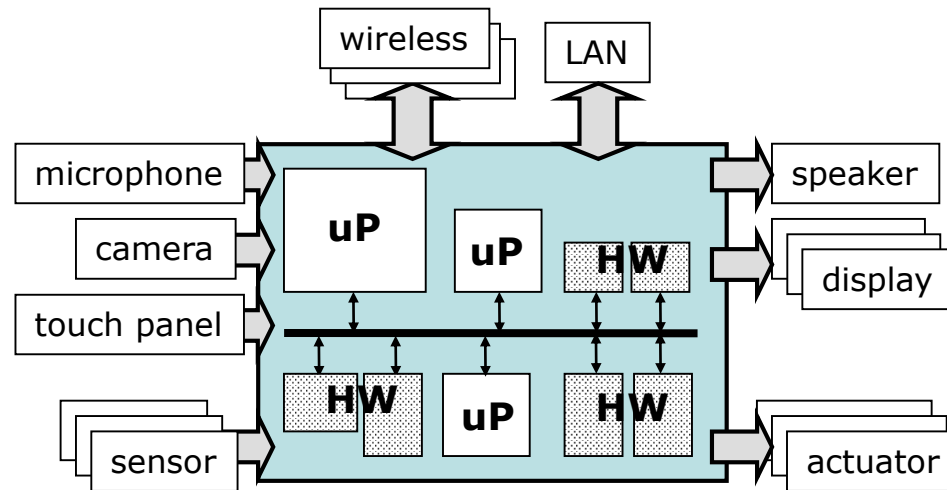
- Components:

- CPUs, memories, dedicated logic blocks (hardware)
 - Software (Software for embedded system → “embedded software”)
 - External parts (IO devices, sensors/actuators, switches, network ports)

Embedded Software

- Key functionalities needed in embedded software
 - Interfacing the hardware components : device drivers
 - Providing “real time” response : real-time operating system (RTOS)
 - Providing a layer that abstracts the hardware-dependent functions and specific software functions: application program interface (API)
 - User application programs that work efficiently in the very resource-restricted environment of embedded systems

Hardware Interface



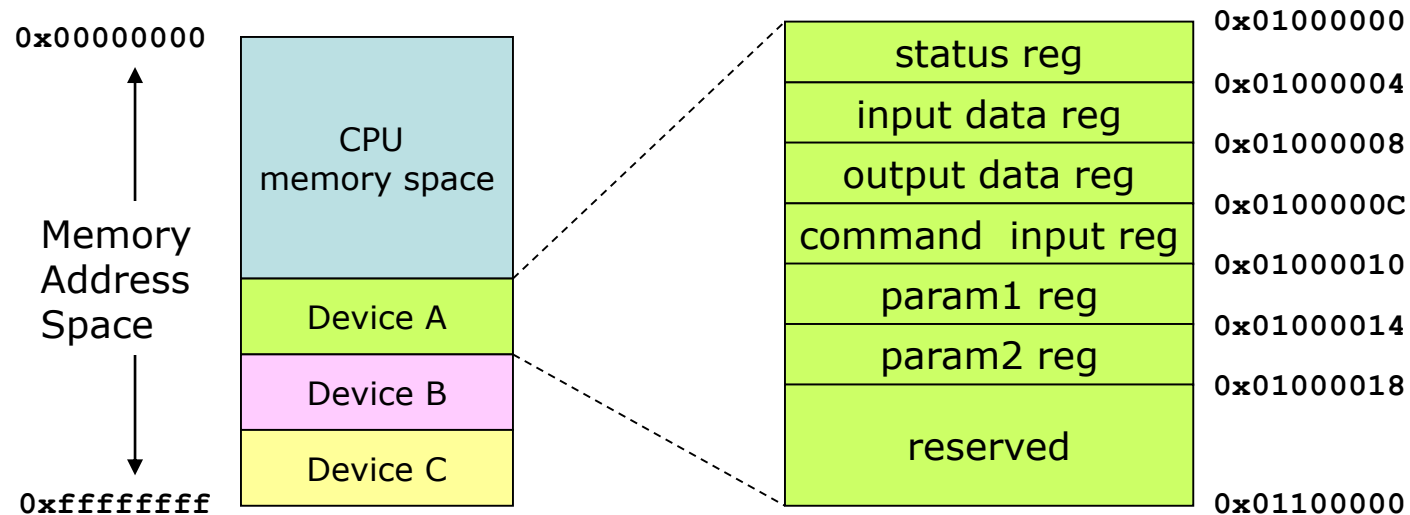
- Two kinds of hardware components
 - Hardware circuit blocks that performs certain computation tasks
 - Hardware circuit blocks that communicates with external devices
- How does the “software” communicate with these hardware components??

Hardware Interface

- The only basic operations to the hardware is “read” and “write”, very SIMPLE!!
 - Write: write some data to the hardware
 - Command: instructs the hardware to do something (set some hardware parameter, do some dedicated task)
 - Data: to a memory device, to a serial bus
 - Read: read some data from the hardware
 - Status: indicates the hardware status (such as “ready” or “busy”)
 - Data: from a memory device, from a serial bus

Hardware Interface

- “WHERE” do we write and read?
 - Memory mapped devices: each device is allocated a dedicated memory space, so we can simply read and write at these addresses
 - Typically, hardware component has several registers (input, output, input/output), where these registers are assigned different addresses



Hardware Interface

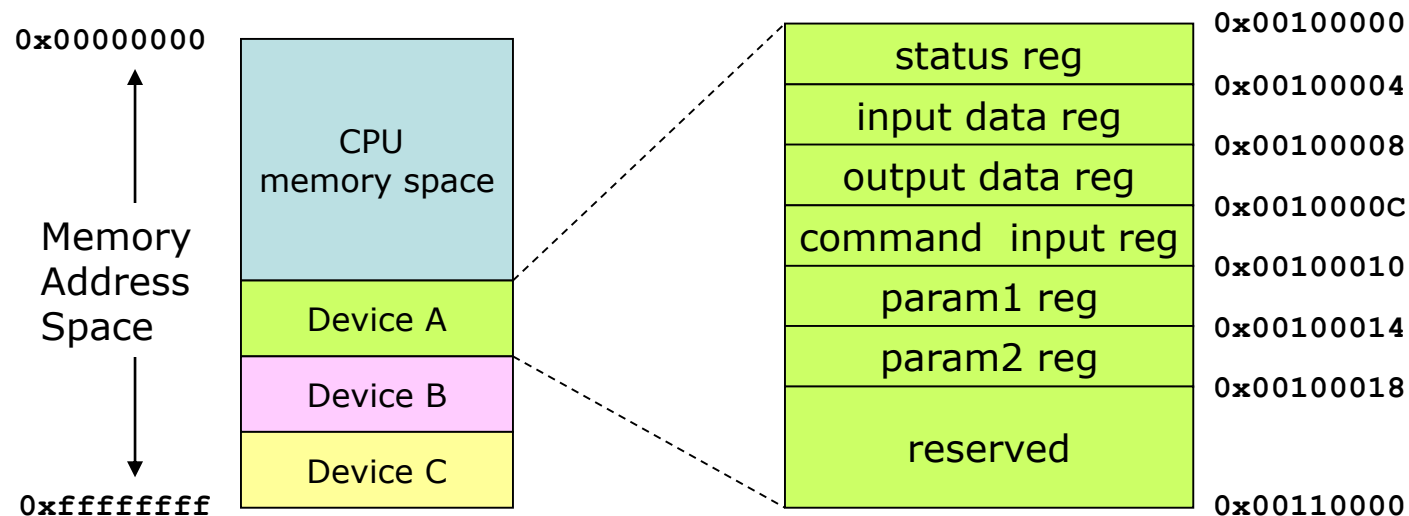
- “How” do we write and read?

- Simply access that memory location

- `int status = *(int *) (0x100000);`

- `*(int *) (0x100004) = input_data;`

- It is safer to say `*(volatile int *)` instead of `*(int *)` to indicate that the address location is in the memory instead of a CPU register



Hardware Interface

- But hardware is usually very unflexible
 - Need to follow a certain sequence of reads and writes to hardware registers so that the hardware can react properly → “PROTOCOL”
 - These protocols are different for each hardware
 - Software that handles these hardware dependent protocols is called the “device driver”
 - Protocols can have layers (“protocol stack”) so that low-level protocols can be reused for different devices

Hardware Interface

- Handshaking protocol for “slave” device
 1. Send a “request” to the device indicating the type of transaction to be performed on the device (read data, write data) → write to device command register
 2. Execute the transaction
 3. CPU monitors the device status register to know when the transaction finished on the device (continuously read the device status register until it becomes “READY”)
 - For a slow device which takes a very long time, CPU time is wasted while waiting for the device to perform the transaction
- How can interface the hardware more efficiently?
 - OS provides a way of calling the different device driver functions in the same way
 - For interfacing to slow devices, OS can assign other tasks to the CPU to avoid wasting the CPU time
 - OS has other important features that allows the embedded software to be programmed easily

Operating System (OS)

- Operating system is the software in a computer system that provide a number of services to the operators (the users) and application programs while maintaining the resources in the system
 - OS is composed of a library of functions called “kernels” (below describes the functions of these kernels)
- Program execution:
 - When an application program is invoked to be executed, the OS kernel creates a “process” by allocating some memory space, loading the program to that allocated memory, then starts running the program
 - Console terminals (where you type commands) is not part of the OS, but is an application program that interprets the typed command and calling the OS kernels according to the interpreted commands

Operating System (OS)

- **Disk access and file system:**
 - Programs and data are stored as “files” in storage devices (hard-drive, floppy disks, CD, DVD, FLASH memory) where OS kernel maintains the file system (file names, file attributes, directories) and calls the “device driver” for that storage device
- **Device driver:**
 - Softwares for interacting with hardware devices
 - Each hardware device requires its own device driver
 - OS provides provides the interface for the application programs to these device drivers by registering these device drivers in the system whenever a device is attached to the system

Operating System (OS)

- Interrupts:

- Interrupts are signals to notify the CPU that an event has occurred (in hardware devices or in some program) which requires immediate handling
- Upon detecting the interrupt signal, CPU immediately jumps to a specific routine (interrupt handler), and after the interrupt handling finishes, returns to the current program

- Multitasking:

- Multiple application programs and subroutines running concurrently by time-sharing the CPU resource
- “Scheduler” kernel controls how long each each program runs and in what order
- Multitask programming is very common especially in embedded systems where an application is composed of many concurrently running tasks

Operating System (OS)

- **Memory management:**

- Memory management kernel maintains the system memory which are used by a number of simultaneously running programs
- Memory protection mechanism must be provided to avoid a system crash caused by buggy (and malicious) programs that access the memory location beyond its allocated space
- Virtual memory system enables the usage of the same physical memory space by multiple programs in order to increase the memory capacity well beyond its physical memory → physical memory is divided into “pages” where old pages (not accessed for a long time) are swapped with new pages between the disk drives

Interrupts

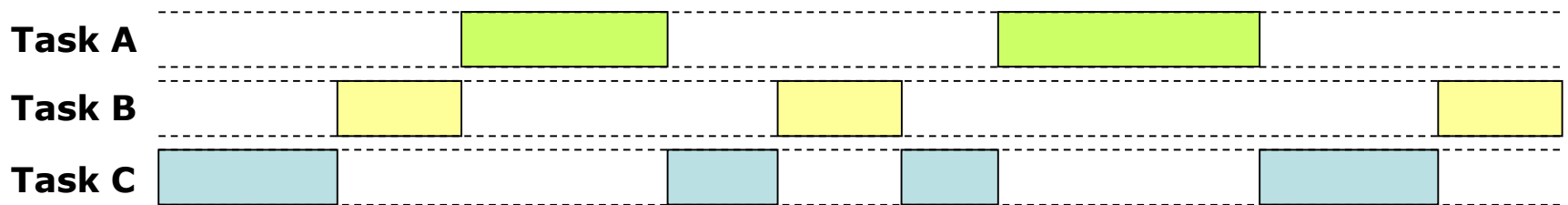
- Interrupts notify the external events to the CPU which needs to be responded immediately
 - Interrupts can be used to notify the CPU that the device transaction has finished (instead of making the CPU monitor the device status register)
 - Interrupts can be used to notify the CPU that the input device has received some data to be processed

Multitasking

- Why do we need “multitasking” in embedded systems?
 - There can be MANY input ports to monitor and respond accordingly
 - There can be MANY output ports to drive in a timely manner (such as on a given interval time)
 - It will be very hard to process all the above tasks using a single program
 - You may not know when those inputs come
 - Output interval times can be different
 - While processing some events, other events can happen which may be undetected
 - Writing separate programs for each task is much easier
 - let the OS handle the scheduling of these programs
 - These individual programs are sometimes called *tasks*

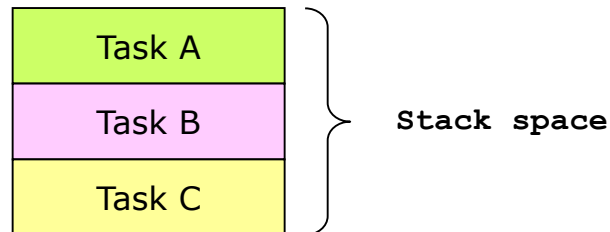
Task

- In embedded software programming, a “task” can be thought of as an individual program
 - Instead of starting from “main” function, each task starts from separate function
- Tasks can execute concurrently, even though there is only 1 CPU resource
 - OS scheduler kernel switches the executing tasks occasionally so that it gives the illusion that these tasks are running simultaneously



Task

- Each task has its own stack memory space
 - The programmer needs to specify the FIXED stack size for each task
 - The programmer must make sure that each task does not exceed its stack space → understanding how much stack space is required for each function call becomes VERY important
 - Common practice in embedded software programming is that **recursive calls should not be used!!!**

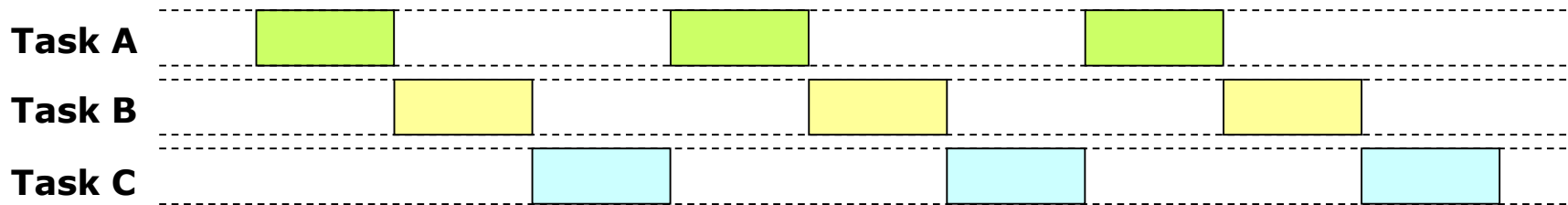


Task Scheduling

- There are several scheduling scheme for the OS scheduler kernel
 - Time sharing: each task is given an equal amount of CPU time → used in non-real-time systems (like PCs) but this cannot guarantee a fast response if there are many running tasks
 - Priority-based preemptive scheduling: task with the highest priority is given the CPU time → used in most real-time OS which can guarantee fast response for those “important” (high priority) tasks

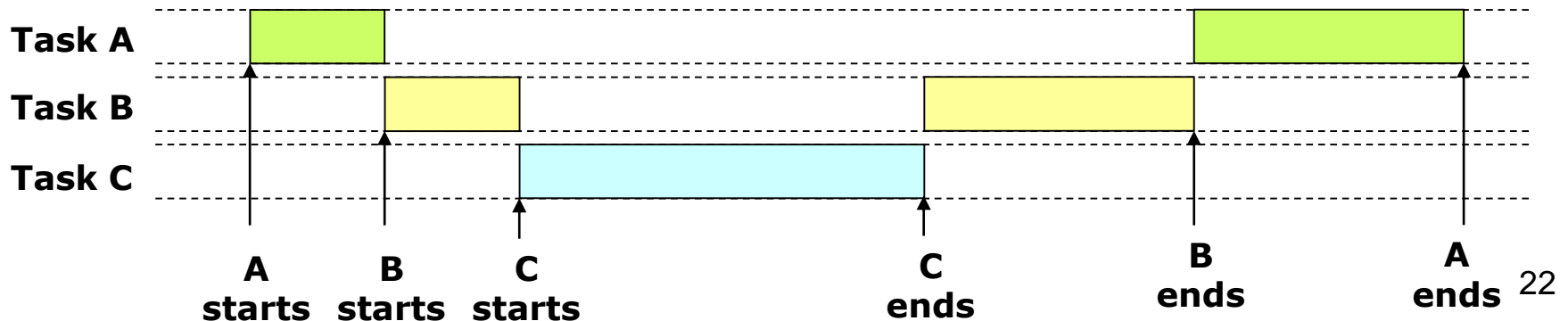
Task Scheduling

- Time sharing: each task is given an equal amount of CPU time



- Priority-based preemptive scheduling: task with the highest priority is given the CPU time

- $\text{priority}(C) > \text{priority}(B) > \text{priority}(A)$

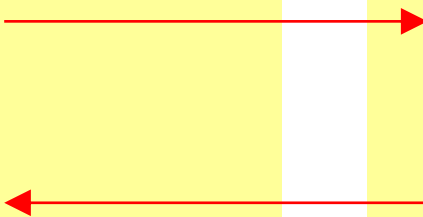


Multitask Programming

- When we program tasks that executes concurrently, there are many things to consider
 - **Resource management**: accesses to system resource such as devices and specific memory locations must be limited to only a single task → mutual exclusion
 - **Synchronization**: exchanging information between tasks requires that these tasks be “synchronized”

```
Task_A:
{
    ...
    data1 = ...; // modify data1
    SignalEvent(1);
    ...
    ...
    ...
    WaitEvent(2);
    ... = data2; // use data2
    ...
}
```

```
Task_B:
{
    ...
    ...
    WaitEvent(1);
    ... = data1; // use data1
    ...
    data2 = ...; // modify data2
    SignalEvent(2);
    ...
}
```

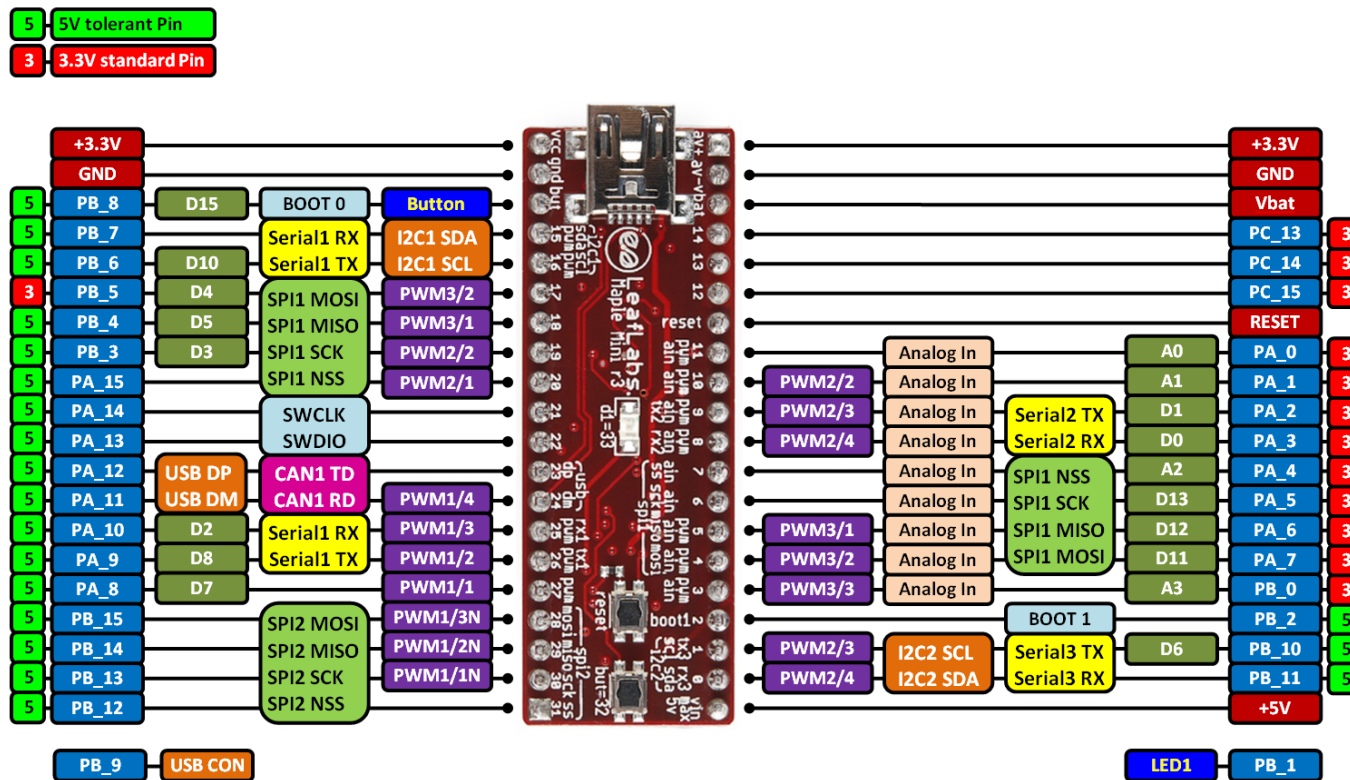


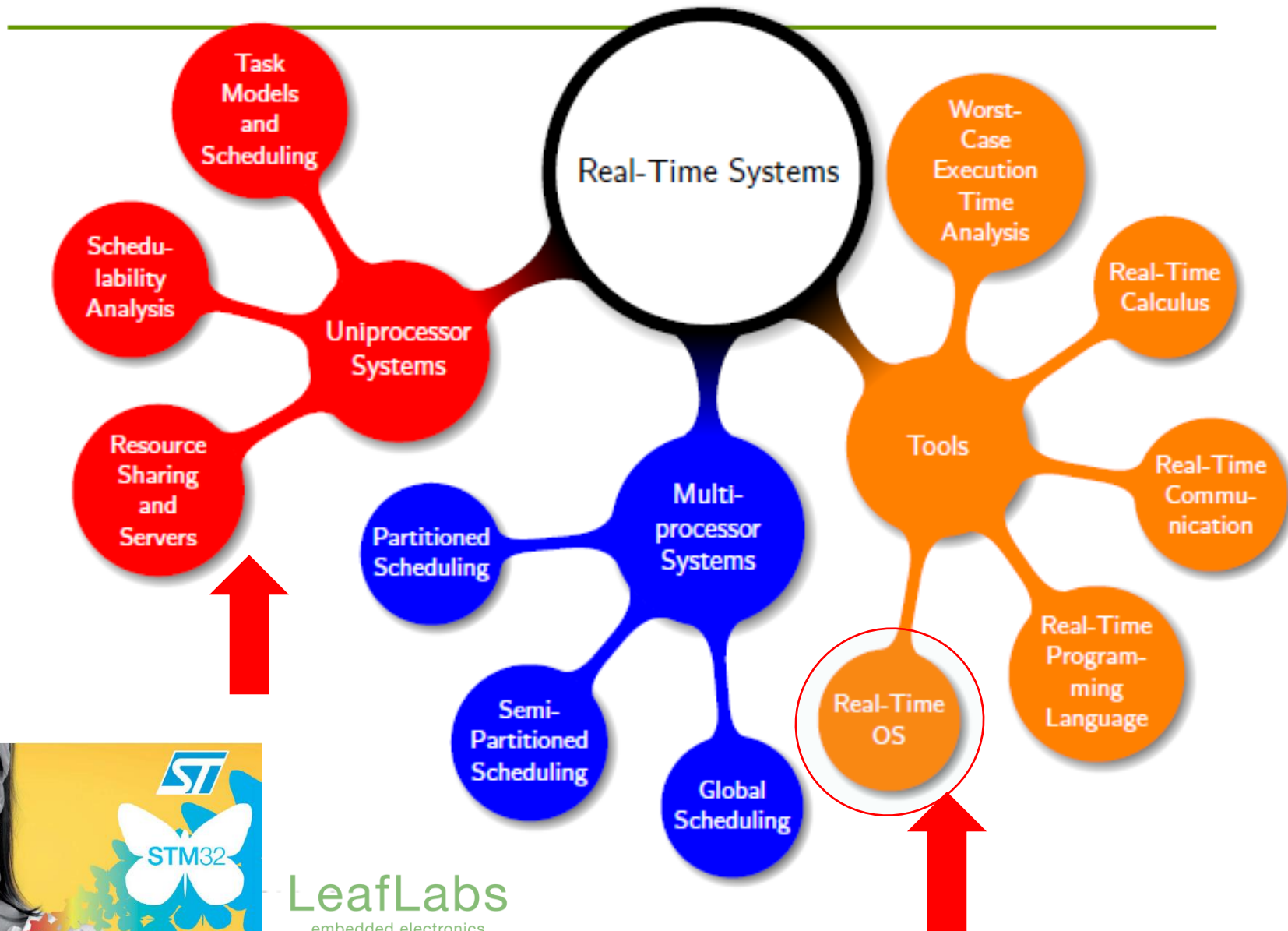
Multitask Programming

- **Timers**: in real-time systems, timers are very useful tool provided in all real-time OS
 - Timers can trigger tasks to activate on strict time interval → Ex: reading sound data from AD (analog-to-digital) converter that has specific sampling frequency (48kHz, 44.1kHz)
 - Timers can deactivate (sleep) tasks for a certain amount of time
- **Interrupts**: tasks can help implement interrupt handling more efficiently
 - Instead of having an interrupt handler routine that can take a long time to process (during interrupt handler routine, other interrupts are usually ignored), tasks can be activated for the heavy processing that can be controlled by the scheduler kernel

Implementation

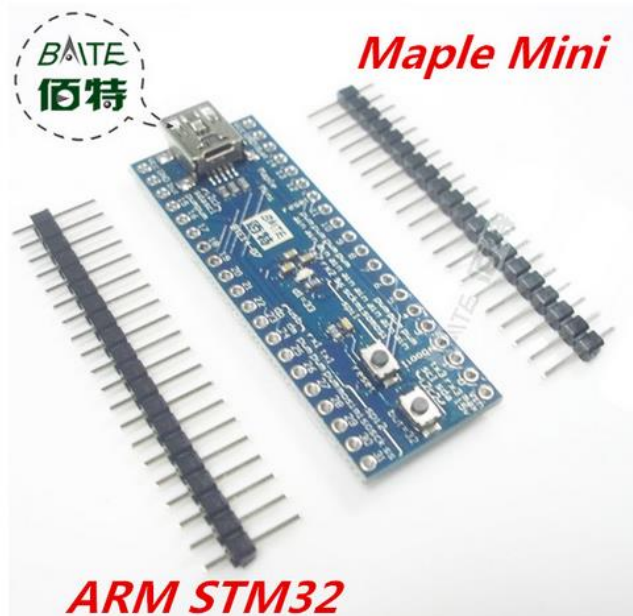
- Using Stm32f103 as the controller
- Based on FreeRTOS and Maple lib





- STM32 F103RCBT6: a 32-bit ARM Cortex M3 microprocessor
- Clock Speed: 72 MHz
- Input Voltage (recommended): 3.0V-12V
- 34 Digital I/O Pins (GPIO)
- 9 Analog Input Pins
- 12-bit ADC resolution
- 12 PWM pins at 16-bit resolution
- Dedicated USB port for programming and communications
- 120 KB Flash and 20 KB SRAM
- Integrated SPI (2) and I2C (2)
- 7 Channels of Direct Memory Access
- 3 USART devices
- Four 4-channel Timers
- Support for low power and sleep modes (<500u

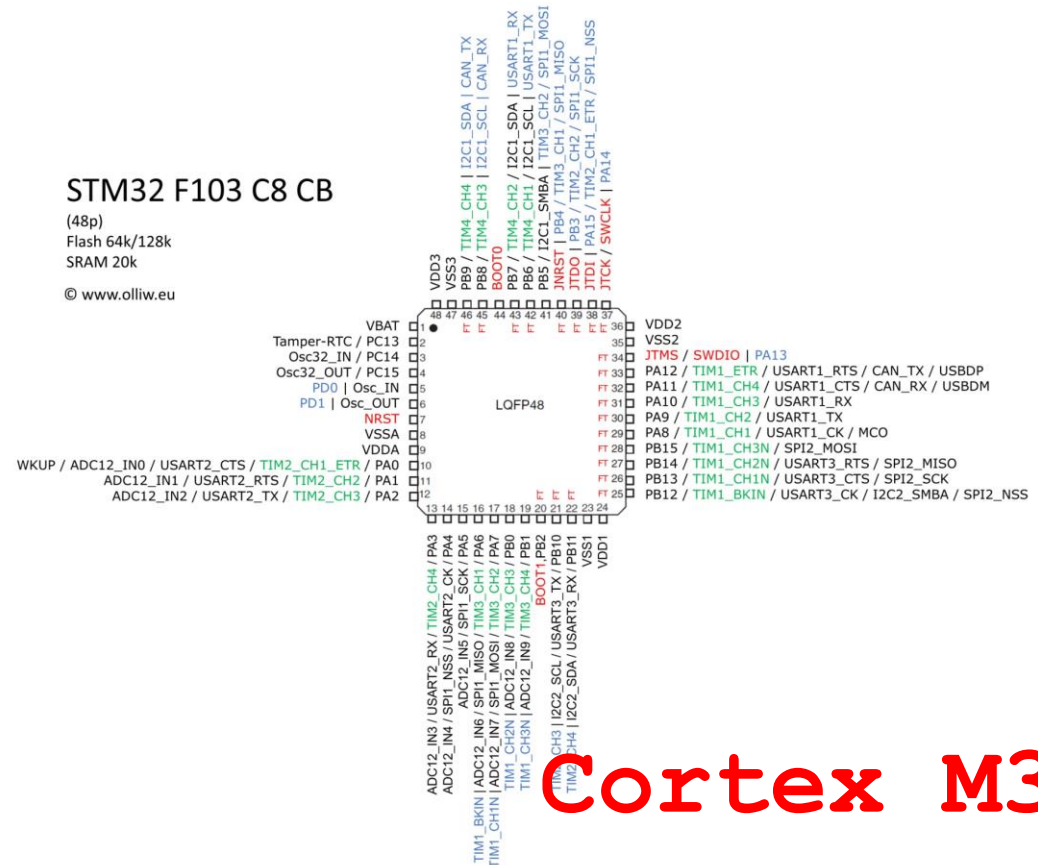
Skills



STM32 F103 C8 CB

(48p)
Flash 64k/128k
SRAM 20k

© www.olliw.eu



Cortex M3

RCC configuration

```

/*****
* Function      : RCC_Configuration
* Description: Configures the different system clocks.
* Input         : None
* Output        : None
* Return        : None
*****/

void RCC_Configuration(void)
{
    RCC_DeInit(); // RCC system reset (for debug purpose)
    RCC_HSEConfig(RCC_HSE_ON); // Enable HSE
    HSEStartUpStatus = RCC_WaitForHSEStartUp(); // Wait till HSE is ready

    if(HSEStartUpStatus == SUCCESS)
    {
        FLASH_PrefetchBufferCmd(FLASH_PrefetchBuffer_Enable); // Enable Prefetch Buffer
        FLASH_SetLatency(FLASH_Latency_2); // Flash 2 wait state
        RCC_HCLKConfig(RCC_SYSCLK_Div1); // HCLK = SYSCLK
        RCC_PCLK2Config(RCC_HCLK_Div1); // PCLK2 = HCLK/1
        RCC_PCLK1Config(RCC_HCLK_Div2); // PCLK1 = HCLK/2
        RCC_PLLConfig(RCC_PLLSource_HSE_Div1, RCC_PLLMul_9); // PLLCLK = 8MHz * 9 = 72 MHz
        RCC_PLLCmd(ENABLE); // Enable PLL

        while(RCC_GetFlagStatus(RCC_FLAG_PLLRDY) == RESET); // Wait till PLL is ready
        RCC_SYSCLKConfig(RCC_SYSCLKSource_PLLCLK); // Select PLL as system clock source
        while(RCC_GetSYSCLKSource() != 0x08); // Wait till PLL is used as system clock source
    }

    /* Enable USART1, GPIOx, TIM1 and AFIO clocks */
    RCC_APB2PeriphClockCmd( RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOB
                           | RCC_APB2Periph_GPIOC | RCC_APB2Periph_USART1
                           | RCC_APB2Periph_TIM8 | RCC_APB2Periph_TIM1 | RCC_APB2Periph_AFIO, ENABLE);
    /* Enable USART2 clock */
    RCC_APB1PeriphClockCmd( RCC_APB1Periph_USART2
                           | RCC_APB1Periph_TIM3 | RCC_APB1Periph_TIM4, ENABLE);
}

```



```

void myTIMxR_Configuration(void)
{
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
    TIM_OCInitTypeDef TIM_OCInitStructure;

    /*
    -----
    TIM1CLK = 72 MHz, Prescaler = 0, TIM1 counter
    The Master Timer TIM1 is running at:
    TIM1 frequency = TIM1 counter clock / (TIM1_
    and the duty cycle is equal to:
    TIM1_CCR1/(TIM1_ARR + 1) = 50%
    -----
    */

    /* TIM1 Peripheral Configuration -----
    /* Time Base configuration */
    TIM_TimeBaseStructure.TIM_Prescaler = 0;
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_
    TIM_TimeBaseStructure.TIM_Period = 255;
    TIM_TimeBaseStructure.TIM_ClockDivision = 0;
    TIM_TimeBaseStructure.TIM_RepetitionCounter

    /* Channel 1 Configuration in PWM mode */
    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_
    TIM_OCInitStructure.TIM_OutputState = TIM_OC
    TIM_OCInitStructure.TIM_OutputPolarity = TIM_
    TIM_OCInitStructure.TIM_Pulse = 127;
    TIM_OCInitStructure.TIM_OCPolarity = TIM_OCI
    TIM_OCInitStructure.TIM_OCNPolarity = TIM_OC
    TIM_OCInitStructure.TIM_OCIdleState = TIM_OC
    TIM_OCInitStructure.TIM_OCNIdleState = TIM_

    TIM_OC3Init(TIM1, &TIM_OCInitStructure);

    //----- Timer3: Encoder Steering, and Time
    TIM_TimeBaseStructure.TIM_Period = 65535;
    TIM_TimeBaseStructure.TIM_Prescaler = 0x00;
    TIM_TimeBaseStructure.TIM_ClockDivision = 0;
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_

    void ADC_setup(void)
    {
        GPIO_InitTypeDef GPIO_InitStructure;
        ADC_InitTypeDef ADC_InitStructure;

        /* Enable ADC1 clock */
        RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);

        /* ADC Channel14 config -----*/
        /* Configure PC.04 (ADC Channel14) as analog input -----*/
        GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4; //PC4 = ADC
        GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;
        GPIO_Init(GPIOC, &GPIO_InitStructure);

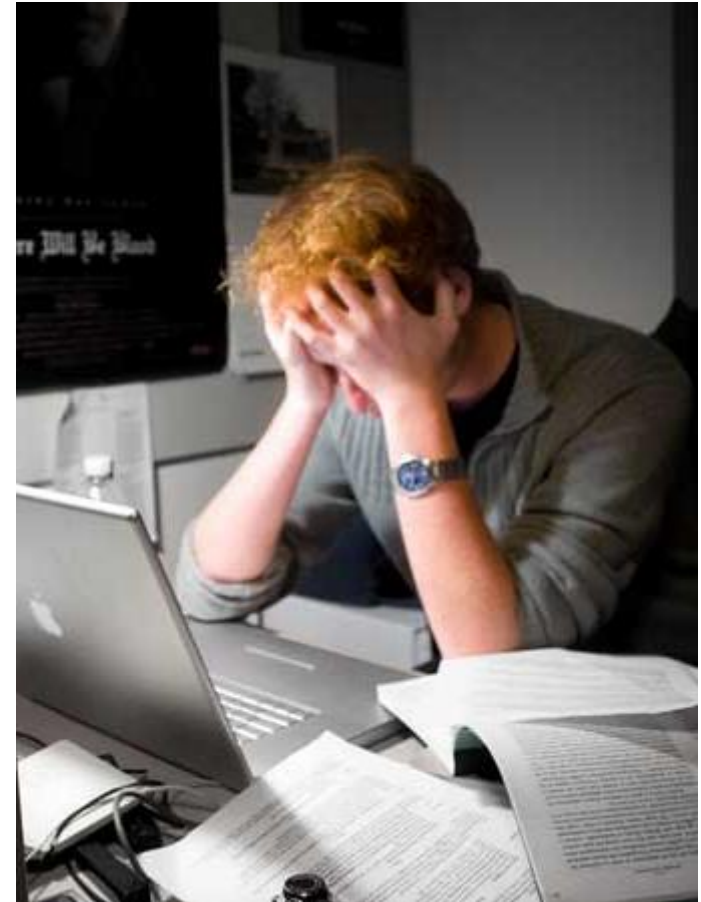
        /* ADC1 Configuration -----*/
        ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;
        ADC_InitStructure.ADC_ScanConvMode = DISABLE;
        ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
        ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None;
        ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
        ADC_InitStructure.ADC_NbrOfChannel = 1;
        ADC_Init(ADC1, &ADC_InitStructure);

        /* ADC1 regular channel14 configuration */
        /* Sampling Time 13.5 cycle of ADC Clock */
        ADC_RegularChannelConfig(ADC1, ADC_Channel_14, 1, ADC_SampleTime_13Cycles5);
        /* Enable ADC1 */
        ADC_Cmd(ADC1, ENABLE);

        ADC_ResetCalibration(ADC1); // Enable ADC1 reset calibration register
        while(ADC_GetResetCalibrationStatus(ADC1));

        ADC_StartCalibration(ADC1); // Start ADC1 calibration
        /* Start ADC1 Software Conversion */
        ADC_SoftwareStartConvCmd(ADC1, ENABLE);
    }
}

```



How to develop Code in Maple

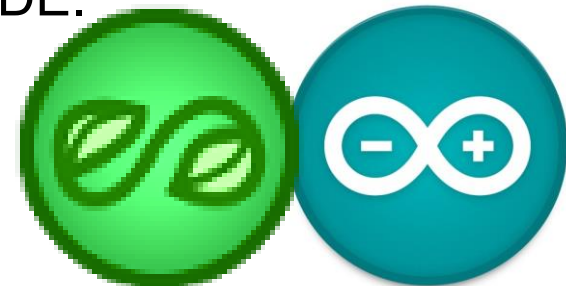
Libmaple

LeafLabs' libmaple ([source code on GitHub](#)) is the library we have developed for the [STM32](#) line of ARM Cortex M3 microcontrollers. Its high-level interfaces are [largely compatible](#) with the AVR libraries written for the [Arduino](#) and [Wiring](#) development boards.

libmaple is split into two pieces: a lower level layer written in pure C, which we call *libmaple proper* (in the [libmaple/](#) directory of the source repository), and the Wiring-style C++ API written on top of it, called *Wirish* (in [wirish/](#)). libmaple is bundled with the [Maple IDE](#).

However, it separately, and [release it standalone](#) for advanced users who might chafe at the “sketch” programming model of the IDE.

Maple and Arduino are Very Similar !!!



Software Environment

The Maple integrated development environment (IDE) is a cross-platform application written in Java, and derives from the **IDE** for the Processing programming language and the Wiring projects. It is designed to introduce programming to artists and other newcomers unfamiliar with software development. It includes a code editor with features such as syntax highlighting, brace matching, and automatic indentation, and is also capable of compiling and uploading programs to the board with a single click. A program or code written for Maple is called a "**sketch**".

Software Environment

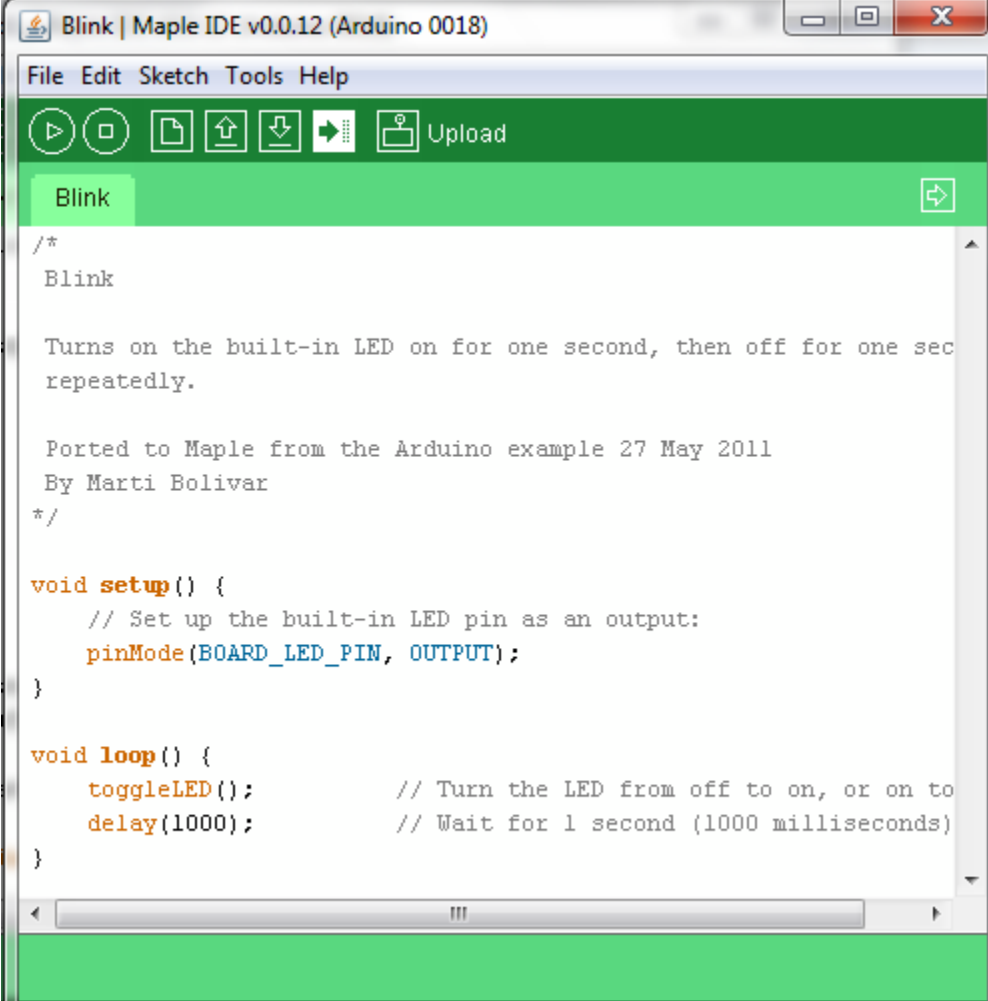
The users need only to define two functions to make an executable cyclic executive program:

`setup()` :

a function that runs once at the start of a program and that can initialize settings

`loop()` :

a function called repeatedly until the board powers off

A screenshot of the Maple IDE v0.0.12 (Arduino 0018) window. The window title is "Blink | Maple IDE v0.0.12 (Arduino 0018)". The menu bar includes "File", "Edit", "Sketch", "Tools", and "Help". Below the menu bar is a toolbar with icons for running, stopping, saving, opening, and uploading. The "Upload" button is highlighted. The main text area shows the code for the "Blink" sketch. The code includes a comment block describing the sketch, its porting to Maple, and the author. It defines a `setup()` function to initialize the LED pin and a `loop()` function to toggle the LED on and off with a 1-second delay.

```
/*
Blink

Turns on the built-in LED on for one second, then off for one sec
repeatedly.

Ported to Maple from the Arduino example 27 May 2011
By Marti Bolivar
*/

void setup() {
    // Set up the built-in LED pin as an output:
    pinMode(BOARD_LED_PIN, OUTPUT);
}

void loop() {
    toggleLED();           // Turn the LED from off to on, or on to
    delay(1000);           // Wait for 1 second (1000 milliseconds)
}
```


Complete Language Index

This is the index of Maple's *language reference* documentation. The "Maple API" column provides API references for documented libmaple functionality. The "C++ for Maple" pages are intended as a minimal reference/refresher for programmers familiar with the Arduino language.

Looking for Something Else?:

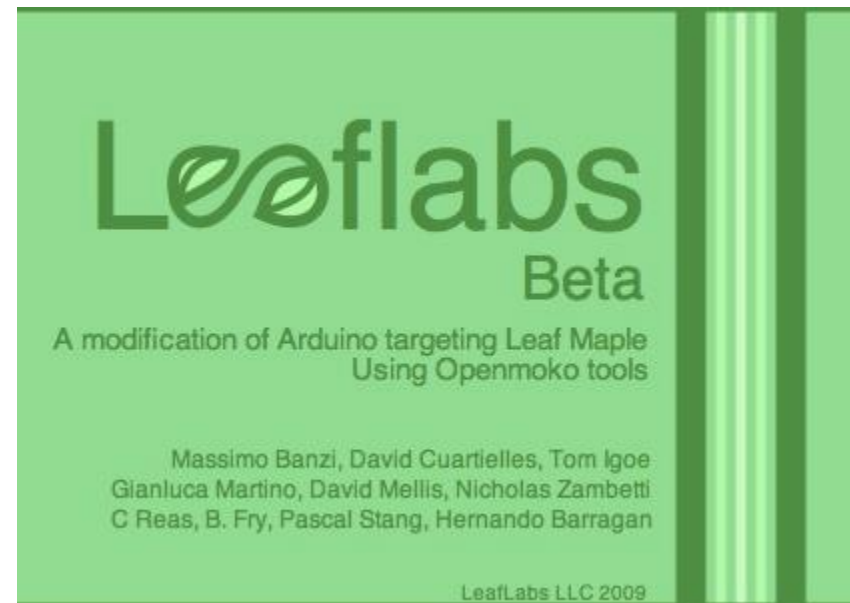
- See the *Maple Library Reference* for extra built-in libraries for dealing with different kinds of hardware.
- If you're looking for something from the C standard library (like `atoi()`, for instance): the *CodeSourcery GCC compiler* used to compile your programs is configured to link against *newlib*, and allows the use of any of its header files. However, dynamic memory allocation (`malloc()`, etc.) is not available.
- If you're looking for pointers to low-level details, see the *Language Recommended Reading* and *libmaple* pages.

Maple API

- `abs()`
- `analogRead()`
- `analogWrite()`
- `ASSERT(...)`
- `attachInterrupt()`
- `bit()`
- `bitClear()`
- `bitRead()`
- `bitSet()`
- `bitWrite()`
- Board-Specific Values
- `boardUsesPin()`
- Constants
- `constrain()`
- `cos()`

C++ for Maple

- Arithmetic Operators (+, -, *, /)
- Arrays
- Assignment Operator (=)
- Bit Shift Operators (<<, >>)
- Bitwise Operators (&, |, ^, ~)
- Boolean Operators
- Booleans
- `break`
- Built-in Types
- `byte`
- `byte()` (cast)
- `char`
- `char()` (cast)
- Comments



For Windows 8 – 10

Open Easydriver folder

The screenshot shows a Windows File Explorer window with the 'win' folder selected. The folder contains 'boards' and 'InstallDriver'. A red dashed arrow points from the 'win' folder to the 'install_drivers' file in the 'src' folder. A right-click context menu is open over 'install_drivers', with 'Run as administrator' highlighted. A SmartScreen warning dialog is displayed in the bottom right corner, with the 'Run' button circled in red.

Right Click

SmartScreen can't be reached right now

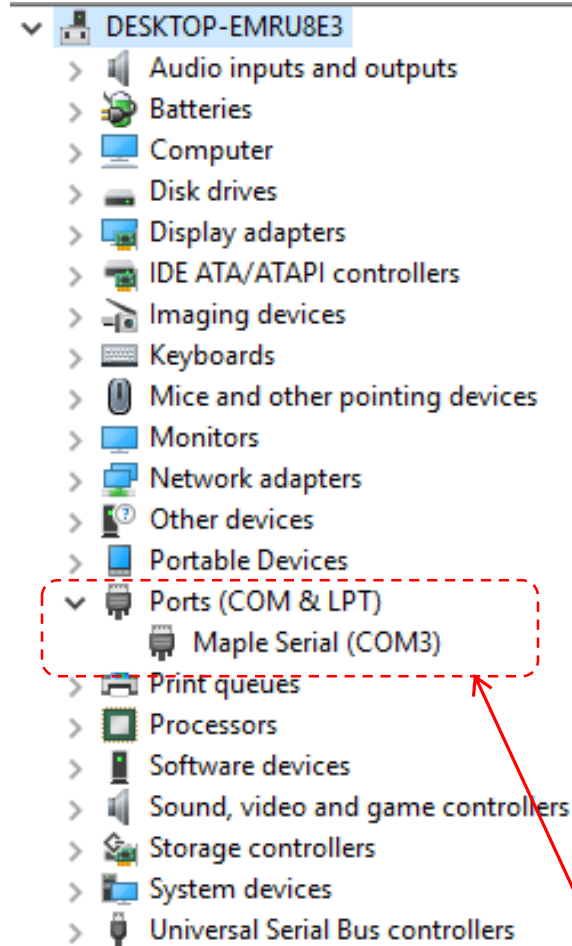
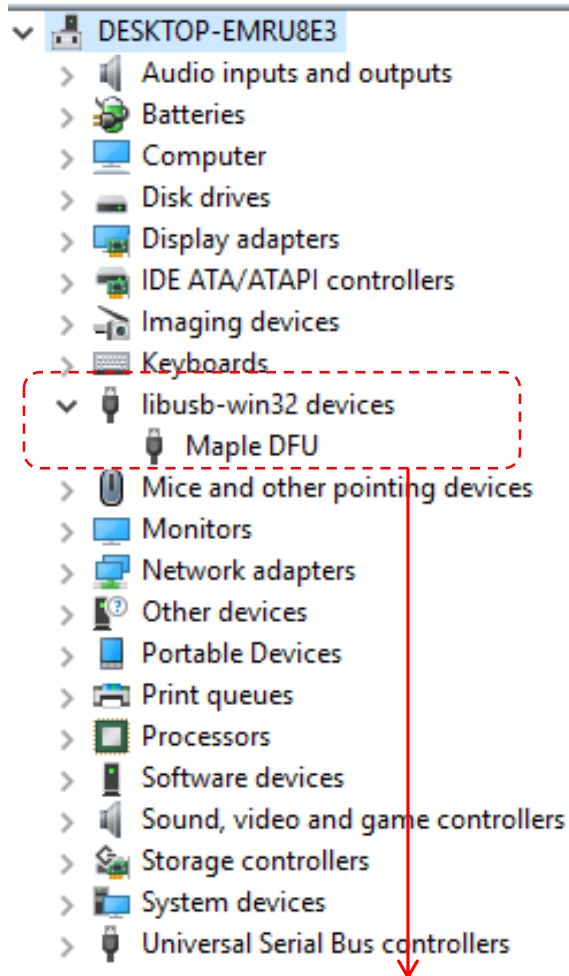
Check your Internet connection. Windows Defender SmartScreen is unreachable and can't help you decide if this app is ok to run.

Publisher: Unknown Publisher
App: install_drivers.bat

Run Don't Run

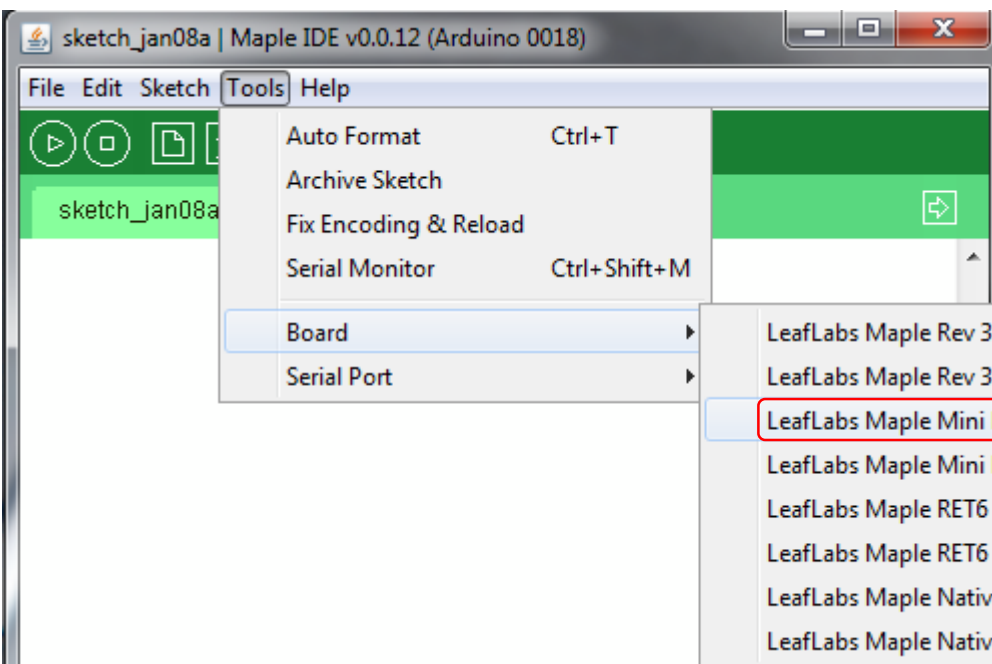
Driver Installation !

For Windows 8 – 10

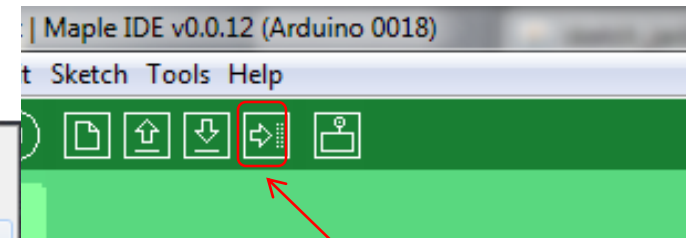


1 DFU install → Open IDE and download without select Serial Port!

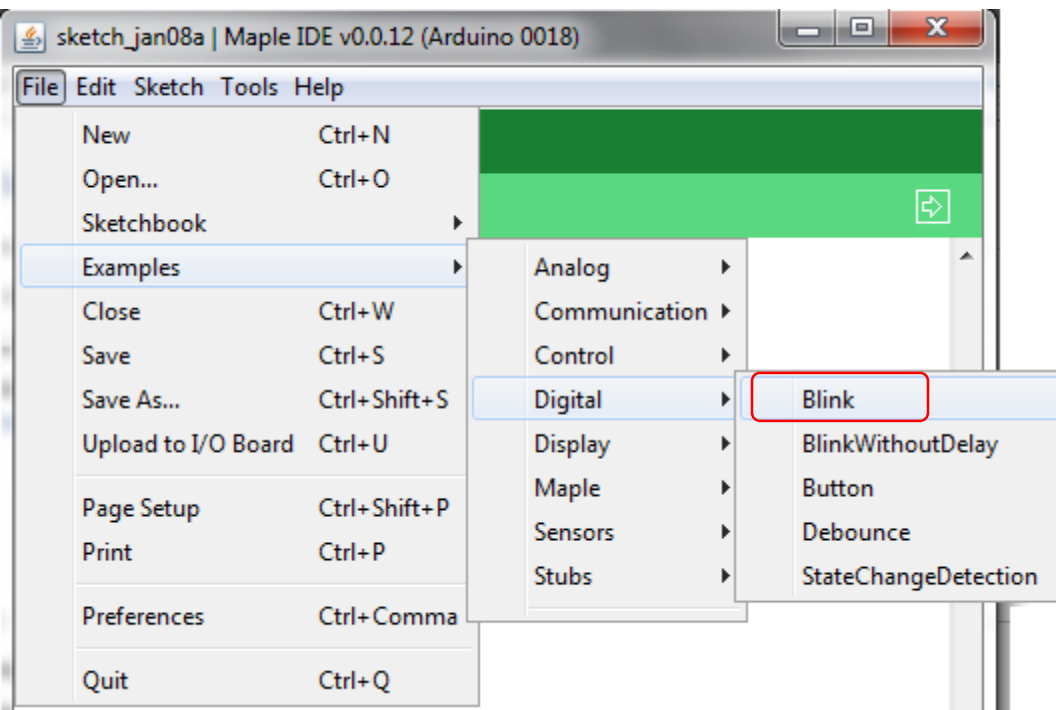
2 The maple serial port will show after the first burn ! 35



First Look !



Download to board



on the built-in LED on for one second, the

tedly.

ed to Maple from the Arduino example 27 May

By Marti Bolivar

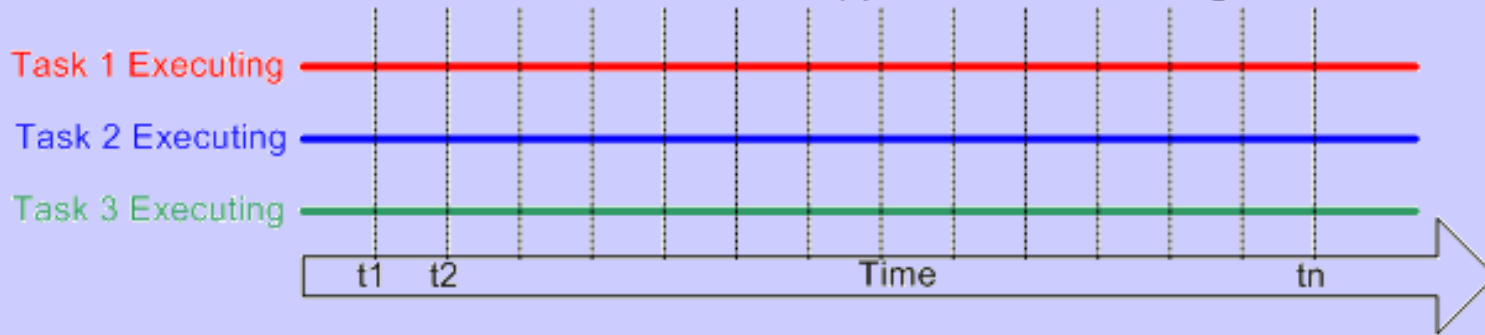
*/

```
void setup() {  
    // Set up the built-in LED pin as an output:  
    pinMode(BOARD_LED_PIN, OUTPUT);  
}
```

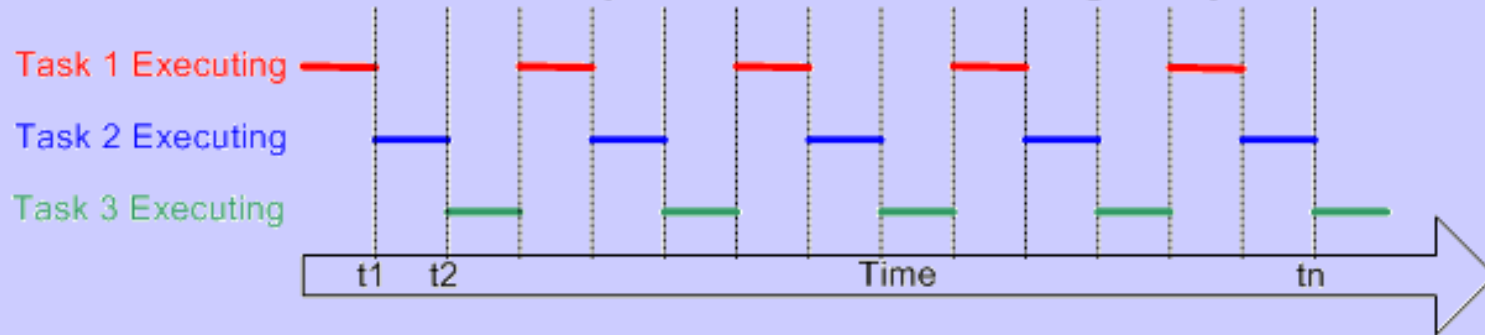
Done uploading.

```
Opening USB Device 0xleaf:0x0003...  
Found Runtime: [0xleaf:0x0003] devnum=1, cfg=0, i  
Setting Configuration 1...  
Claiming USB DFU Interface...  
Setting Alternate Setting ...  
Determining device status: state = dfuIDLE, statu  
dfuIDLE, continuing
```

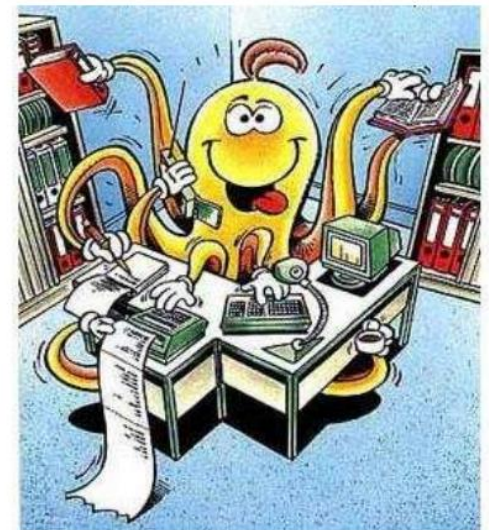
All available tasks appear to be executing ...



... but only one task is ever executing at any time.



A real-time operating system (RTOS) is an operating system (OS) intended to serve real-time application process data as it comes in, typically without buffering delays. Processing time requirements (including any OS delay) are measured in tenths of seconds or shorter.



The Cortex-M3 Port of FreeRTOS

The Cortex-M3 port includes all the standard FreeRTOS features:

- Pre-emptive or co-operative operation
- Very flexible task priority assignment
- Queues
- Binary semaphores
- Counting semaphores
- Recursive semaphores
- Mutexes
- Tick hook functions
- Idle hook functions
- Stack overflow checking
- Trace hook macros
- Optional commercial licensing and support

Resources Used By FreeRTOS

FreeRTOS makes use of the Cortex-M3 SysTick, PendSV, and SVC interrupts. These interrupts are not available for use by the application.

FreeRTOS has a very small footprint. A typical kernel build will consume approximately 6K bytes of Flash space and a few hundred bytes of RAM. Each task also requires RAM to be allocated for use as the task stack.

Task Functions

Tasks are implemented as C functions. The only thing special about them is their prototype, which must return void and take a void pointer parameter. The prototype is demonstrated by Listing 1.

```
void ATaskFunction( void *pvParameters );
```

Listing 1. The task function prototype

Each task is a small program in its own right. It has an entry point, will normally run forever within an infinite loop, and will not exit.

FreeRTOS tasks must **not** be allowed to return from their implementing function in any way—they must not contain a 'return' statement and must not be allowed to execute past the end of the function. If a task is no longer required, it should instead be explicitly deleted.

```
void ATaskFunction( void *pvParameters )
{
/* Variables can be declared just as per a normal function. Each instance
of a task created using this function will have its own copy of the
iVariableExample variable. This would not be true if the variable was
declared static - in which case only one copy of the variable would exist
and this copy would be shared by each created instance of the task. */
int iVariableExample = 0;

/* A task will normally be implemented as an infinite loop. */
for( ;; )
{
    /* The code to implement the task functionality will go here. */
}

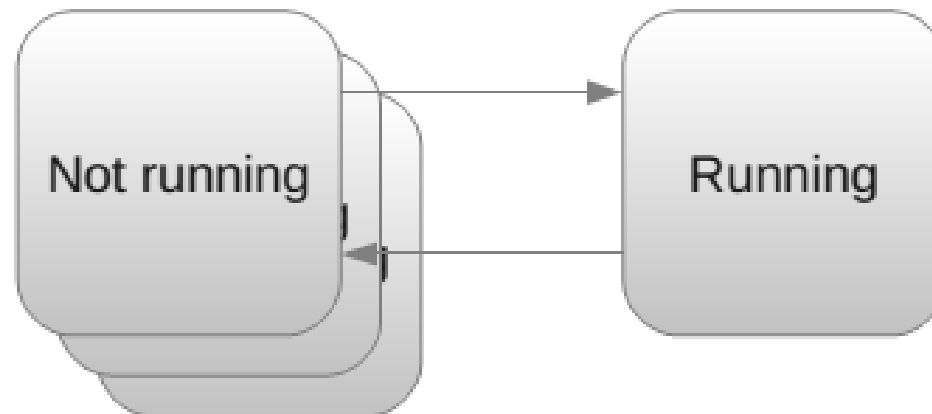
/* Should the task implementation ever break out of the above loop
then the task must be deleted before reaching the end of this function.
The NULL parameter passed to the vTaskDelete() function indicates that
the task to be deleted is the calling (this) task. */
vTaskDelete( NULL );
}
```

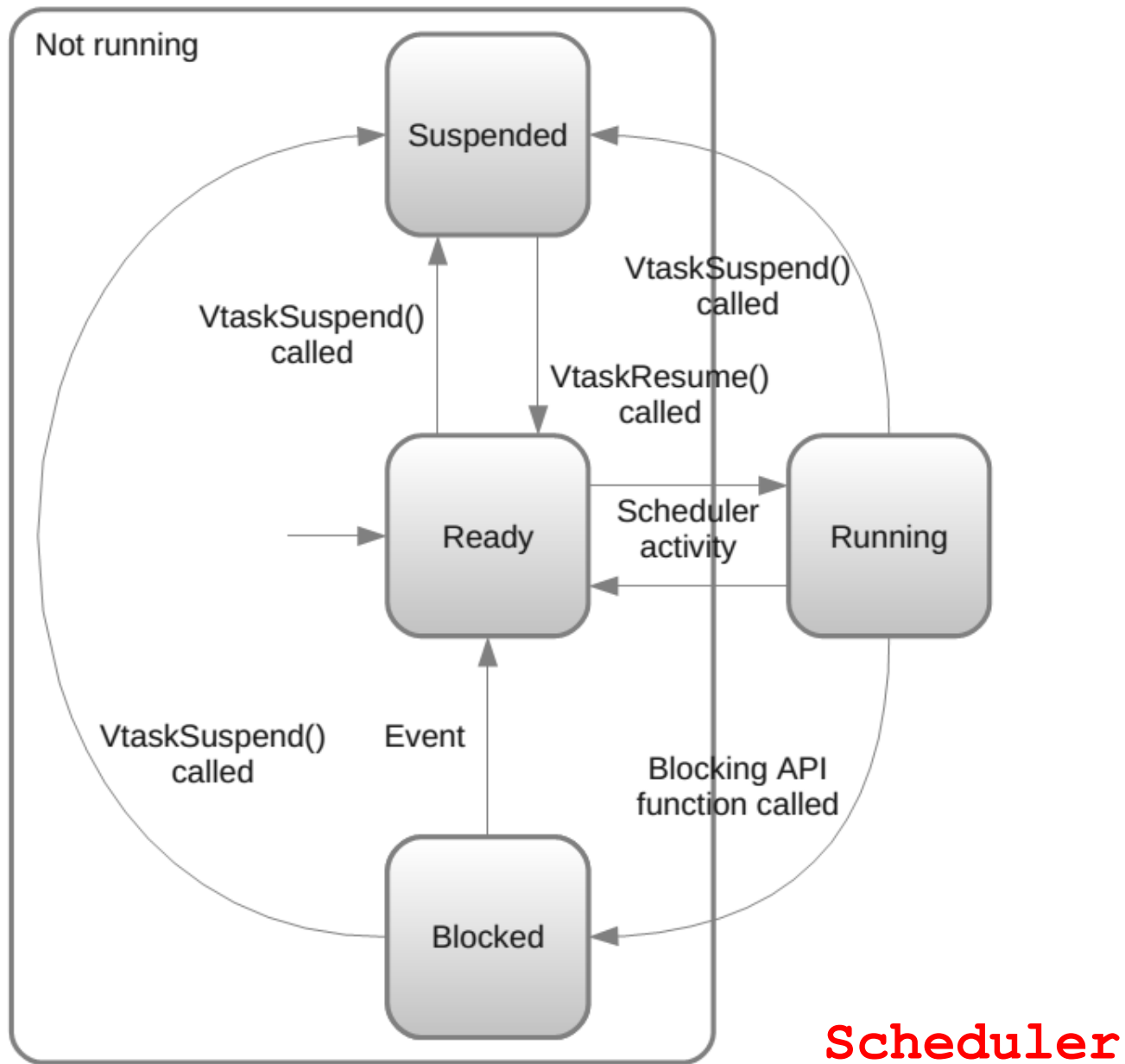
Listing 2. The structure of a typical task function

Top Level Task States

An application can consist of many tasks. If the microcontroller running the application contains a single core, then only one task can be executing at any given time. This implies that a task can exist in one of two states, Running and Not Running. We will consider this simplistic model first—but keep in mind that this is an over-simplification as later we will see that the Not Running state actually contains a number of sub-states.

When a task is in the Running state, the processor is executing its code. When a task is in the Not Running state, the task is dormant, its status having been saved ready for it to resume execution the next time the scheduler decides it should enter the Running state. When a task resumes execution, it does so from the instruction it was about to execute before it last left the Running state.





Life cycle of a task

Creating Tasks

The xTaskCreate() API Function

Tasks are created using the FreeRTOS xTaskCreate() API function. This is probably the most complex of all the API functions, so it is unfortunate that it is the first encountered, but tasks must be mastered first as they are the most fundamental component of a multitasking system. All the examples make use of the xTaskCreate() function,

```
portBASE_TYPE xTaskCreate( pdTASK_CODE pvTaskCode,  
                           const signed char * const pcName,  
                           unsigned short usStackDepth,  
                           void *pvParameters,  
                           unsigned portBASE_TYPE uxPriority,  
                           xTaskHandle *pxCreatedTask  
                           );
```

MAPLE FreeRTOS

```
#include <MapleFreeRTOS.h>

void vLEDFlashTask( void *pvParameters ) {
    for(;;) {
        vTaskDelay( 1000 / portTICK_RATE_MS );
        digitalWrite(BOARD_LED_PIN, HIGH );
        vTaskDelay( 1000 / portTICK_RATE_MS );
        digitalWrite(BOARD_LED_PIN, LOW );
    }
}

// The setup() method runs once, when the sketch starts
void setup() {
    // initialize the digital pin as an output:
    pinMode(BOARD_LED_PIN, OUTPUT);
    digitalWrite(BOARD_LED_PIN, HIGH ); // Board is running //
    xTaskCreate(vLEDFlashTask, (const signed char*)"task_LED",100,NULL,1,NULL);
    vTaskStartScheduler();
}

void loop()
{
    while(1); // Insert background code here
}
```

Example 1. Creating tasks

This example demonstrates the steps needed to create two simple tasks then start the tasks executing. The tasks simply print out a string periodically, using a crude null loop to create the period delay. Both tasks are created at the same priority and are identical except for the string they print out

```
void vTask1( void *pvParameters )
{
    const char *pcTaskName = "Task 1 is running\n";
    volatile unsigned long ul;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation. There is
            nothing to do in here. Later examples will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}
```

```
void vTask2( void *pvParameters )
{
const char *pcTaskName = "Task 2 is running\n";
volatile unsigned long ul;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation. There is
            nothing to do in here. Later examples will replace this crude
            loop with a proper delay/sleep function. */

        }
    }
}
```

```

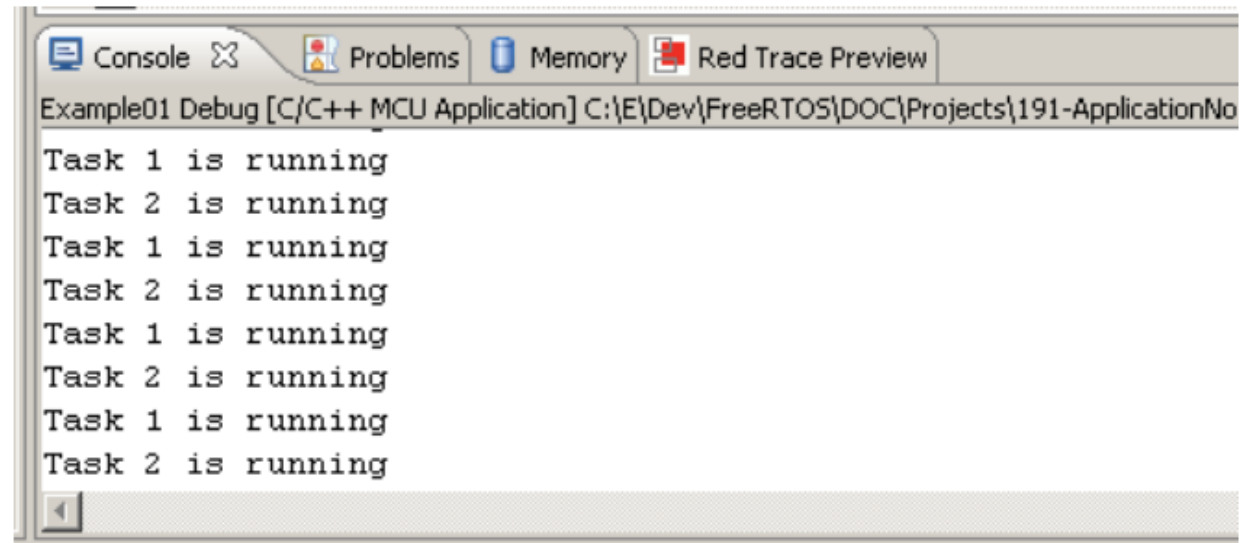
int main( void )
{
    /* Create one of the two tasks. Note that a real application should check
    the return value of the xTaskCreate() call to ensure the task was created
    successfully. */
    xTaskCreate( vTask1, /* Pointer to the function that implements the task. */
                "Task 1", /* Text name for the task. This is to facilitate
                           debugging only. */
                240,      /* Stack depth in words. */
                NULL,     /* We are not using the task parameter. */
                1,        /* This task will run at priority 1. */
                NULL );   /* We are not going to use the task handle. */

    /* Create the other task in exactly the same way and at the same priority. */
    xTaskCreate( vTask2, "Task 2", 240, NULL, 1, NULL );

    /* Start the scheduler so the tasks start executing. */
    vTaskStartScheduler();

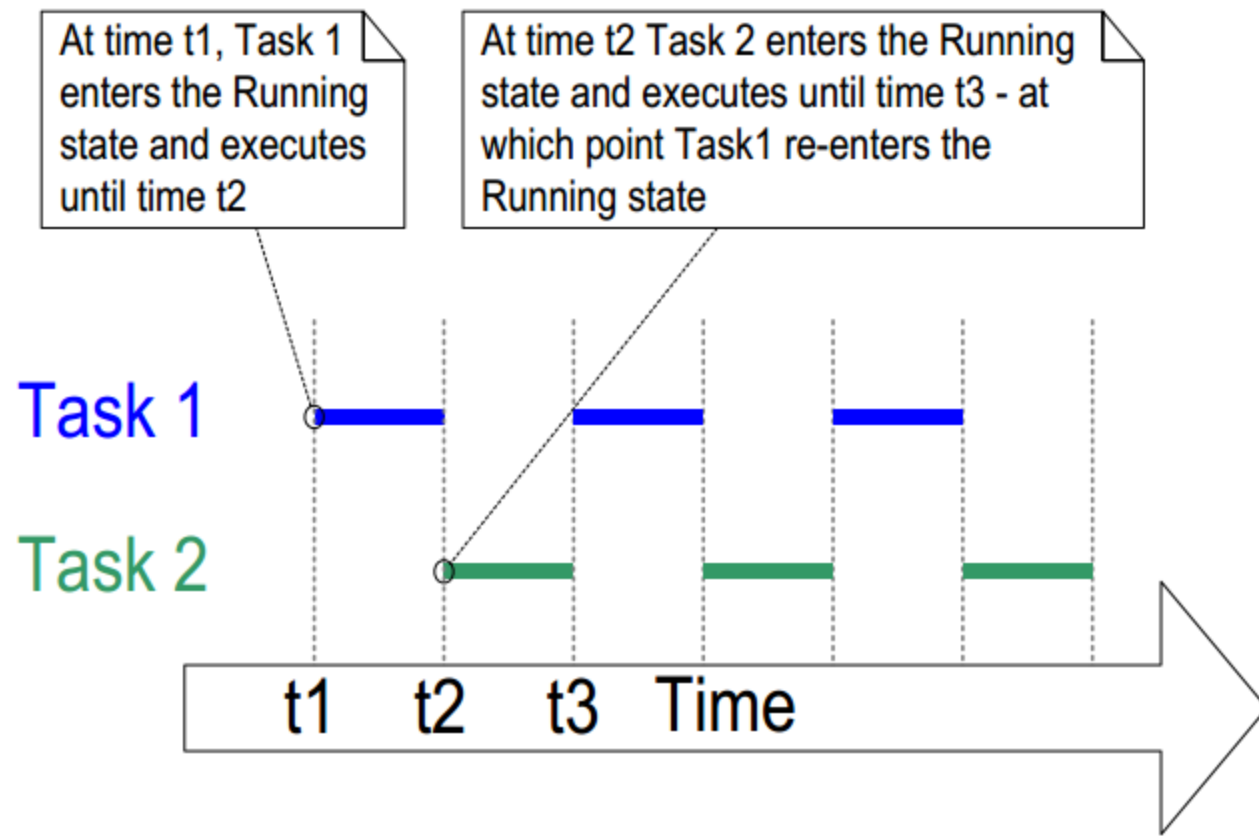
    /* If all is well then
    now be running the tas
    there was insufficient
    Chapter 5 provides mor
    for( ;; );
}

```



The output produced when Example 1 is executed

Figure shows the two tasks appearing to execute simultaneously; however, as both tasks are executing on the same processor, this cannot be the case. In reality, both tasks are rapidly entering and exiting the Running state. Both tasks are running at the same priority, and so share time on the single processor. Their actual execution pattern is shown



The execution pattern of the two Example 1 tasks

Task Priorities

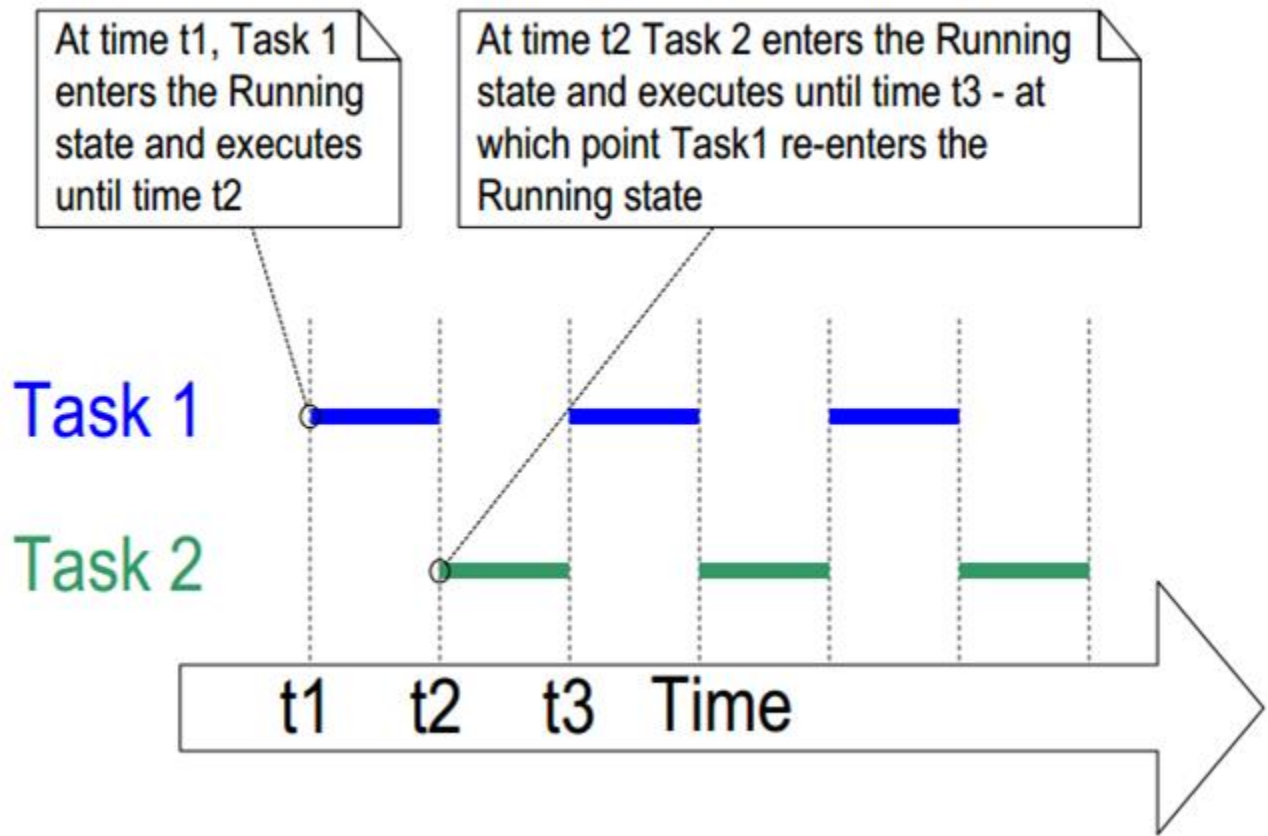
The `uxPriority` parameter of the `xTaskCreate()` API function assigns an initial priority to the task being created. The priority can be changed after the scheduler has been started by using the `vTaskPrioritySet()` API function.

The maximum number of priorities available is set by the application-defined `configMAX_PRIORITIES` compile time configuration constant within `FreeRTOSConfig.h`. FreeRTOS itself does not limit the maximum value this constant can take, but the higher the `configMAX_PRIORITIES` value the more RAM the kernel will consume, so it is always advisable to keep the value set at the minimum necessary.

FreeRTOS imposes no restrictions on how priorities can be assigned to tasks. Any number of tasks can share the same priority—ensuring maximum design flexibility. You can assign a unique priority to every task, if desired (as required by some schedule-ability algorithms), but this restriction is not enforced in any way.

Low numeric priority values denote low-priority tasks, with priority 0 being the lowest priority possible. Therefore, the range of available priorities is 0 to $(\text{configMAX_PRIORITIES} - 1)$.

The scheduler will always ensure that the highest priority task that is able to run is the task selected to enter the Running state. Where more than one task of the same priority is able to run, the scheduler will transition each task into and out of the Running state, in turn. This is the behavior observed in the examples so far, where both test tasks are created at the same priority and both are always able to run. Each such task executes for a 'time slice'; it enters the Running state at the start of the time slice and exits the Running state at the end of the time slice.



To be able to select the next task to run, the scheduler itself must execute at the end of each time slice. A periodic interrupt, called the tick interrupt, is used for this purpose. The length of the time slice is effectively set by the tick interrupt frequency, which is configured by the `configTICK_RATE_HZ` compile time configuration constant in `FreeRTOSConfig.h`. For example, if `configTICK_RATE_HZ` is set to **1000(Hz)**, then the time slice will be 100 milliseconds.

```
#define configUSE_PREEMPTION          1
#define configUSE_IDLE_HOOK          0
#define configUSE_TICK_HOOK          0
#define configCPU_CLOCK_HZ           ( ( unsigned long ) 72000000 )
#define configTICK_RATE_HZ           ( ( portTickType ) 1000 )
#define configMAX_PRIORITIES          ( ( unsigned portBASE_TYPE ) 5 )
#define configMINIMAL_STACK_SIZE      ( ( unsigned short ) 120 )
#define configTOTAL_HEAP_SIZE         ( ( size_t ) ( 8 * 1024 ) )
#define configMAX_TASK_NAME_LEN       ( 16 )
#define configUSE_TRACE_FACILITY      1
#define configUSE_16_BIT_TICKS        0
#define configIDLE_SHOULD_YIELD       1
```

/ Co-routine definitions. */*

```
#define configUSE_CO_ROUTINES          0
#define configMAX_CO_ROUTINE_PRIORITIES ( 2 )
```

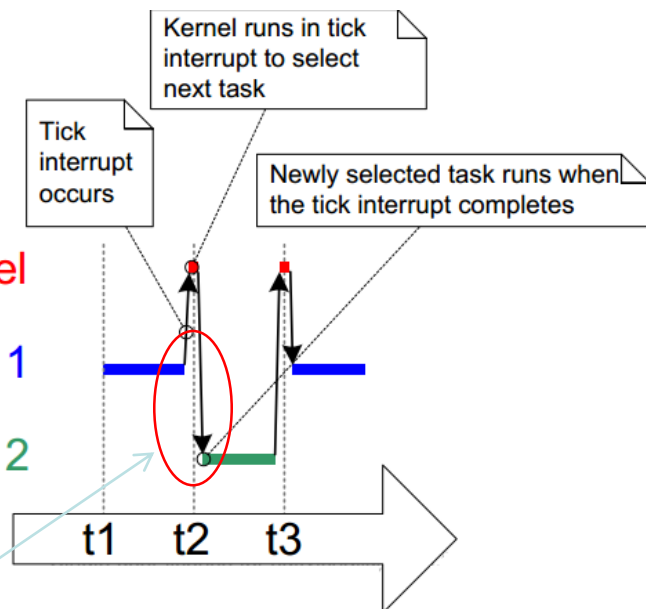
```
#define configUSE_MUTEXES              1
#define configUSE_COUNTING_SEMAPHORES 1
#define configUSE_ALTERNATIVE_API      0
#define configCHECK_FOR_STACK_OVERFLOW 2
#define configUSE_RECURSIVE_MUTEXES    1
#define configQUEUE_REGISTRY_SIZE       0
#define configGENERATE_RUN_TIME_STATS  0
```

Changeable

Kernel

Task 1

Task 2



Context switch (Overhead)

Example 3. Experimenting with priorities

The scheduler will always ensure that the highest priority task that is able to run is the task selected to enter the Running state. In our examples so far, two tasks have been created at the same priority, so both entered and exited the Running state in turn. This example looks at what happens when we change the priority of one of the two tasks created in Example 2. This time, the first task will be created at priority 1, and the second at priority 2.

```
/* Define the strings that will be passed in as the task parameters. These are
defined const and not on the stack to ensure they remain valid when the tasks are
executing. */
static const char *pcTextForTask1 = "Task 1 is running\n";
static const char *pcTextForTask2 = "Task 2 is running\n";

int main( void )
{
    /* Create the first task at priority 1. The priority is the second to last
    parameter. */
    xTaskCreate( vTaskFunction, "Task 1", 240, (void*)pcTextForTask1, 1, NULL );

    /* Create the second task at priority 2. */
    xTaskCreate( vTaskFunction, "Task 2", 240, (void*)pcTextForTask2, 2, NULL );

    /* Start the scheduler so the tasks start executing. */
    vTaskStartScheduler();

    /* If all is well we will never reach here as the scheduler will now be
    running. If we do reach here then it is likely that there was insufficient
    heap available for the idle task to be created. */
    for( ;; );
}
```

```

void vTaskFunction( void *pvParameters )
{
    char *pcTaskName;
    volatile unsigned long ul;

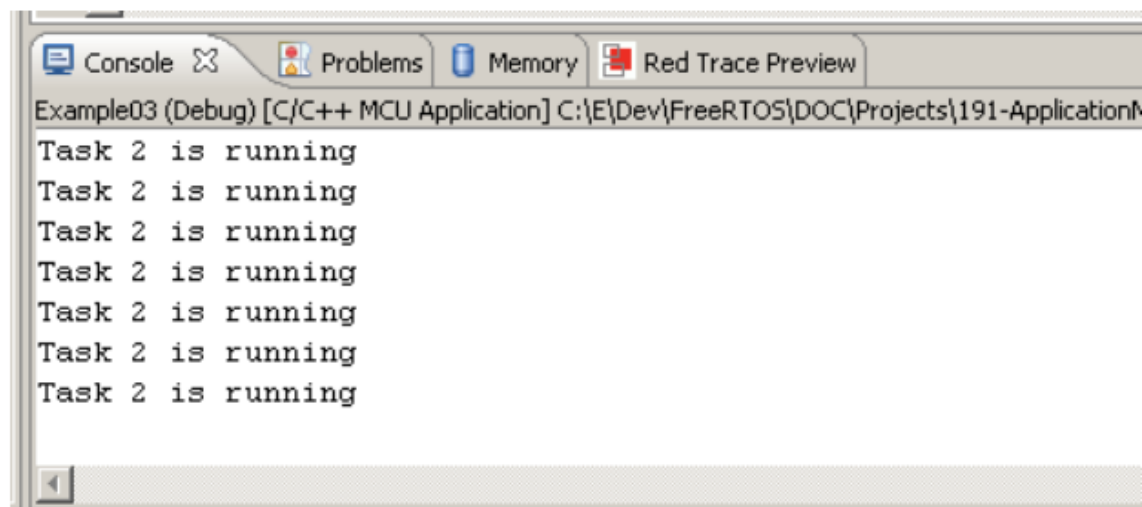
    /* The string to print out is passed in via the parameter.  Cast this to a
    character pointer. */
    pcTaskName = ( char * ) pvParameters;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation.  There is
            nothing to do in here.  Later exercises will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}

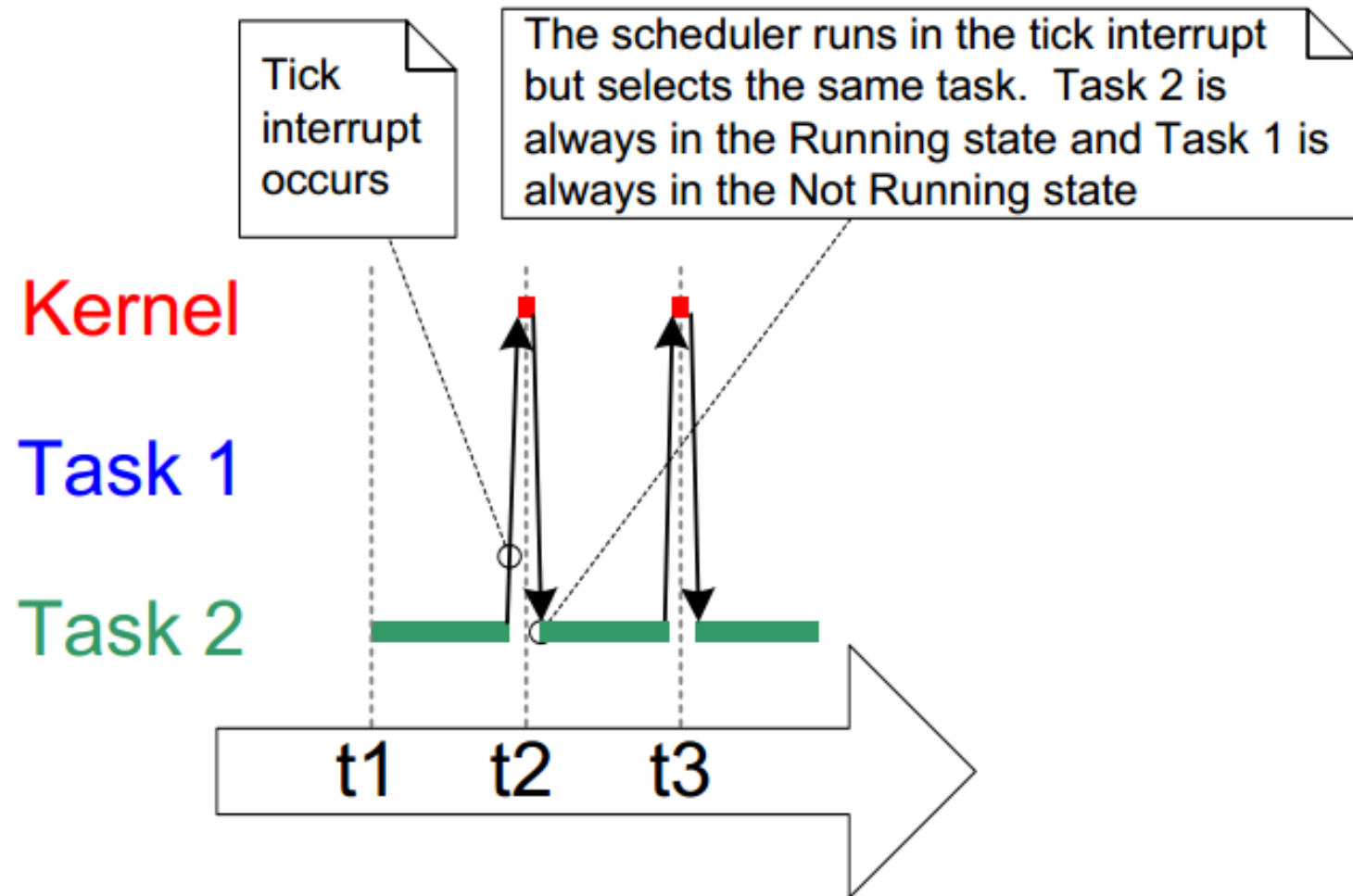
```

Task 1 is said to be 'starved' of processing time by Task 2.



Running both test tasks at different priorities

Task 2 is always able to run because it never has to wait for anything—it is either cycling around a null loop or printing to the terminal.



The execution pattern when one task has a higher priority than the other

`vTaskDelay()` places the calling task into the Blocked state for a fixed number of tick interrupts. While in the Blocked state the task does not use any processing time, so processing time is consumed only when there is work to be done.

```
void vTaskDelay( portTickType xTicksToDelay );
```

The `vTaskDelay()` API function prototype

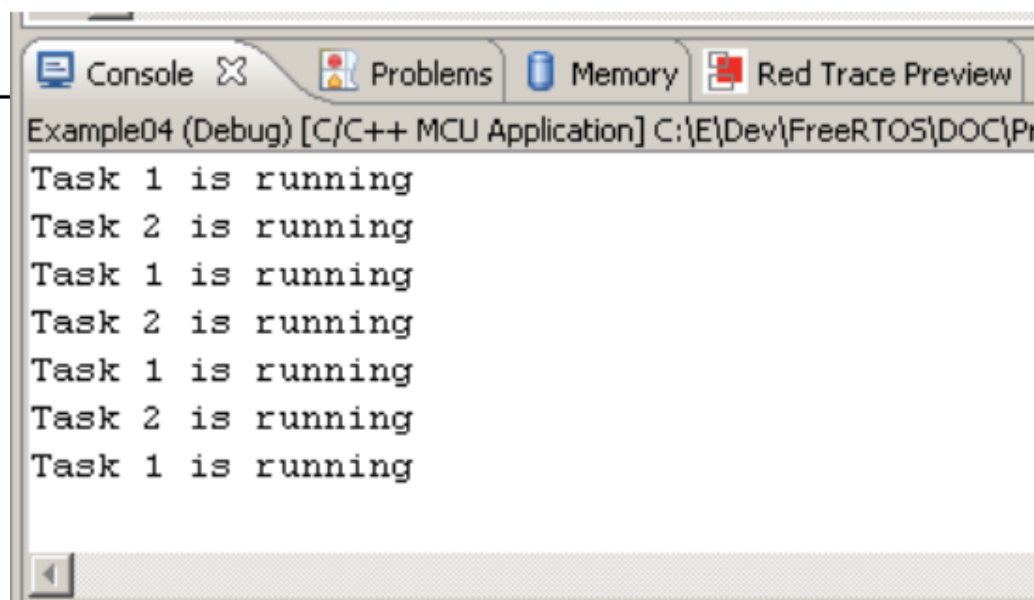
Parameter Name	Description
<code>xTicksToDelay</code>	<p>The number of tick interrupts that the calling task should remain in the Blocked state before being transitioned back into the Ready state.</p> <p>For example, if a task called <code>vTaskDelay(100)</code> while the tick count was 10,000, then it would immediately enter the Blocked state and remain there until the tick count reached 10,100.</p> <p>The constant <code>portTICK_RATE_MS</code> can be used to convert milliseconds into ticks.</p>

```
void vTaskFunction( void *pvParameters )
{
    char *pcTaskName;

    /* The string to print out is passed in via the parameter.  Cast this to a
    character pointer. */
    pcTaskName = ( char * ) pvParameters;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period.  This time a call to vTaskDelay() is used which
        places the task into the Blocked state until the delay period has expired.
        The delay period is specified in 'ticks', but the constant
        portTICK_RATE_MS can be used to convert this to a more user friendly value
        in milliseconds.  In this case a period of 250 milliseconds is being
        specified. */
        vTaskDelay( 250 / portTICK_RATE_MS );
    }
}
```



MAPLE Task Example

```
#include <MapleFreeRTOS.h>

void vLEDFlashTask( void *pvParameters ) {
    for(;;) {
        vTaskDelay( 300 / portTICK_RATE_MS ); //300 milliseconds
        digitalWrite(BOARD_LED_PIN, HIGH );
        vTaskDelay( 700 / portTICK_RATE_MS );
        digitalWrite(BOARD_LED_PIN, LOW );
    }
}

void vtask1( void *pvParameters ) {
    portTickType xLastWakeTime;

    xLastWakeTime = xTaskGetTickCount();

    SerialUSB.println("task 1 is running ..... \n\r");

    for(;;)
    {

        SerialUSB.println("Periodic Task 1 cycle \n\r");
        vTaskDelayUntil(&xLastWakeTime, (1000/ portTICK_RATE_MS));

    }
}
```

```

void vtask2( void *pvParameters ) {
    portTickType xLastWakeTime;

    xLastWakeTime = xTaskGetTickCount();

    SerialUSB.println("task 1 is runnning ..... \n\r");

    for(;;)
    {

        SerialUSB.println("Periodic Task 2 cycle \n\r");
        vTaskDelayUntil(&xLastWakeTime, (500/ portTICK_RATE_MS));

    }
}

void vtask3( void *pvParameters ) {

    for(;;)
    {
        SerialUSB.println("Continous Task Executed");
        vTaskDelay( 100 / portTICK_RATE_MS);
    }
}

```

```

// The setup() method runs once, when the sketch starts
void setup() {
    // initialize the digital pin as an output:
    pinMode(BOARD_LED_PIN, OUTPUT);
    digitalWrite(BOARD_LED_PIN, HIGH ); // Board is running //
    xTaskCreate(vLEDFlashTask, (const signed char*)"task_LED", 100, NULL, 1, NULL);
    xTaskCreate(vtask1, NULL, 100, NULL, 3, NULL);
    xTaskCreate(vtask2, NULL, 100, NULL, 1, NULL);
    xTaskCreate(vtask3, NULL, 100, NULL, 1, NULL);
    vTaskStartScheduler();
}

void loop()
{
    while(1); // Insert background code here
}

```

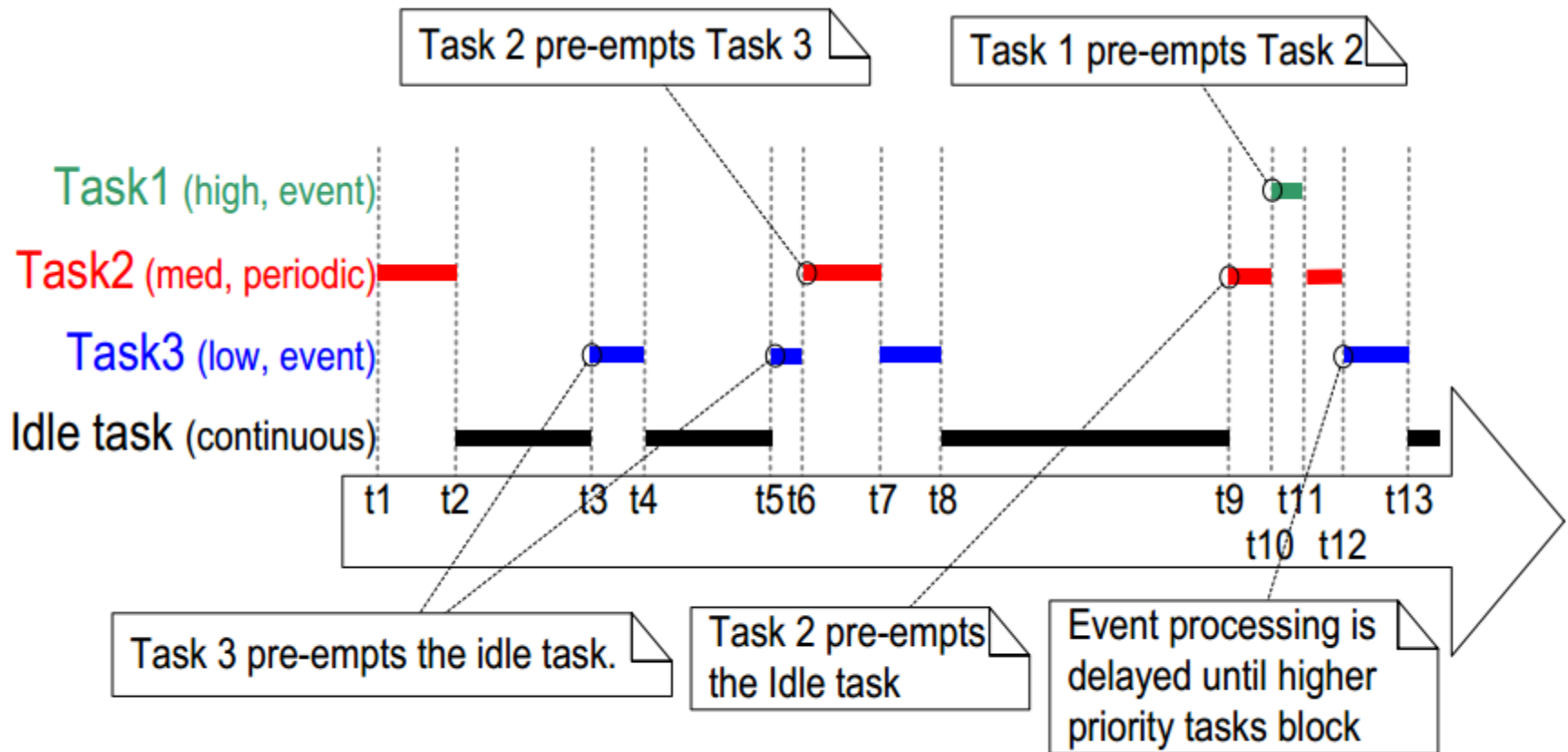
The Scheduling Algorithm—A Summary

Prioritized Pre-emptive Scheduling

The examples in this chapter illustrate how and when FreeRTOS selects which task should be in the Running state.

- Each task is assigned a priority.
- Each task can exist in one of several states.
- Only one task can exist in the Running state at any one time.
- The scheduler always selects the highest priority Ready state task to enter the Running state.

This type of scheme is called ‘Fixed Priority Pre-emptive Scheduling’—‘Fixed Priority’ because each task is assigned a priority that is not altered by the kernel itself (only tasks can change priorities); ‘Pre-emptive’ because a task entering the Ready state or having its priority altered will always pre-empt the Running state task, if the Running state task has a lower priority.



Execution pattern with pre-emption points highlighted

Critical Sections and Suspending the Scheduler

Basic Critical Sections

Basic critical sections are regions of code that are surrounded by calls to the macros `taskENTER_CRITICAL()` and `taskEXIT_CRITICAL()`, respectively, as demonstrated in Listing Critical sections are also known as critical regions.

```
/* Ensure access to the GlobalVar variable cannot be interrupted by  
placing it within a critical section. Enter the critical section. */  
taskENTER_CRITICAL();
```

```
/* A switch to another task cannot occur between the call to  
taskENTER_CRITICAL() and the call to taskEXIT_CRITICAL(). Interrupts  
may still execute, but only interrupts whose priority is above the  
value assigned to the configMAX_SYSCALL_INTERRUPT_PRIORITY constant  
– and those interrupts are not permitted to call FreeRTOS API  
functions. */
```

```
GlobalVar |= 0x01;
```

```
/* Access to GlobalVar is complete so the critical section can be exited. */  
taskEXIT_CRITICAL();
```

Using a critical section to guard access to a variable

```
void vPrintString( const char *pcString )
{
    static char cBuffer[ ioMAX_MSG_LEN ];

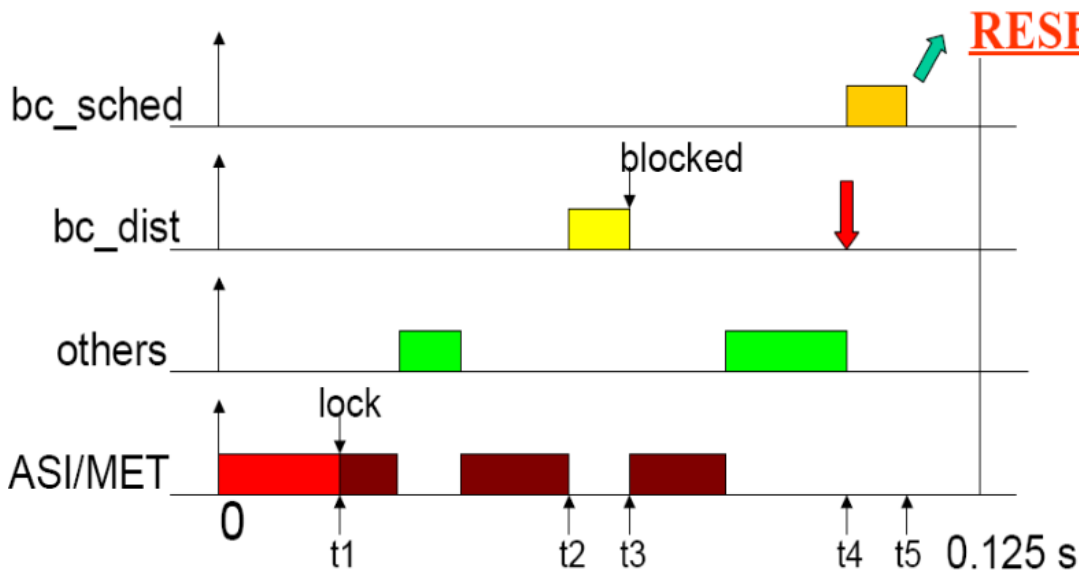
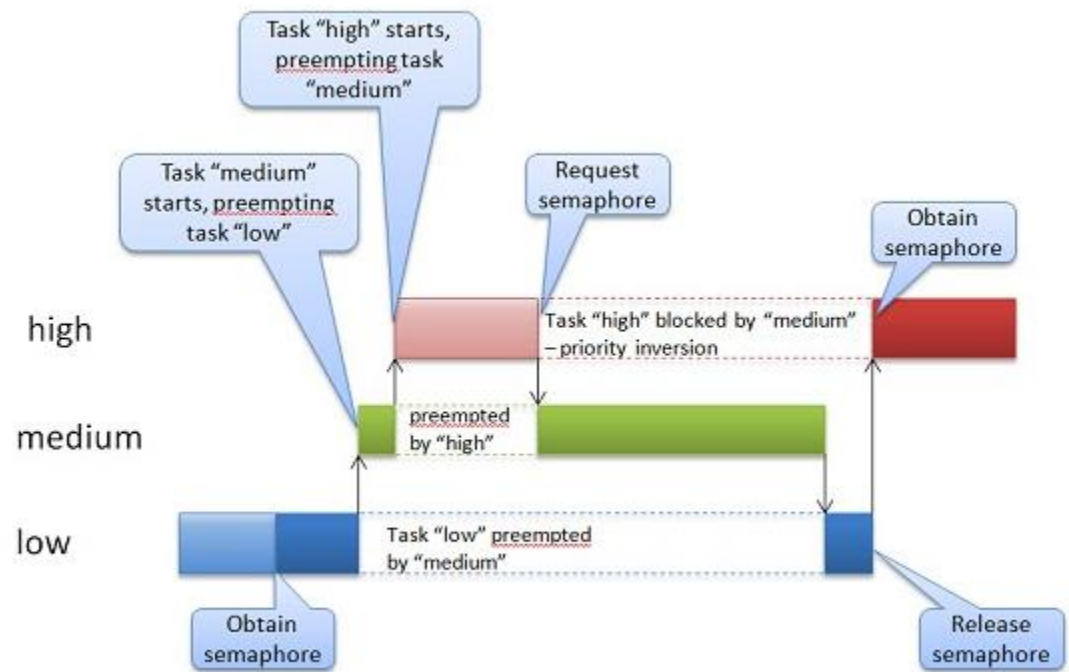
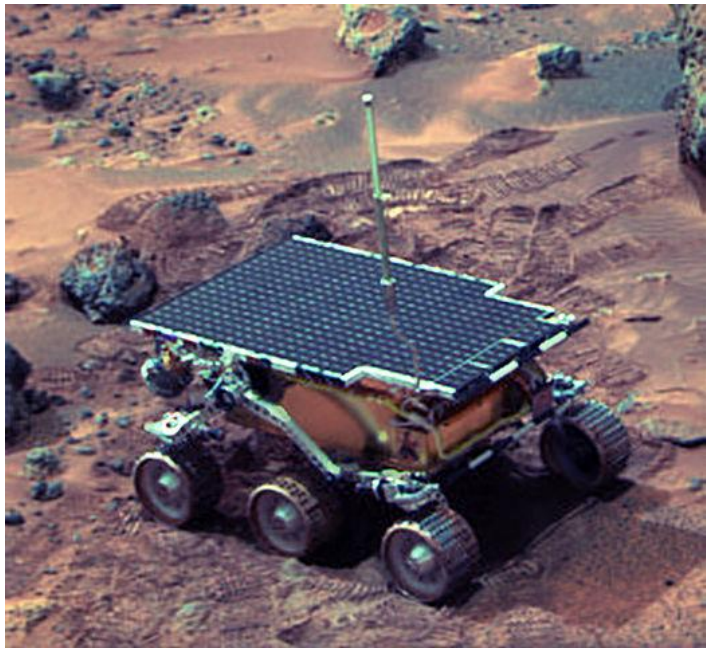
    /* Write the string to stdout, suspending the scheduler as a method
    of mutual exclusion. */
    vTaskSuspendScheduler();
    {
        sprintf( cBuffer, "%s", pcString );
        consoleprint( cBuffer );
    }
    xTaskResumeScheduler();
}
```

The vTaskSuspendAll() API Function

```
void vTaskSuspendAll( void );
```

The xTaskResumeAll() API Function

```
portBASE_TYPE xTaskResumeAll( void );
```



Classic problem in RTOS

Priority inversion
Starvation

Real-Time OS Products

- **WindowsCE**: Can run Windows application on PDAs and mobile phone
- **Embedded Linux**: Linux kernels optimized for embedded systems
- **VxWorks**: Aerospace, military (high reliability)
- **SymbianOS**: Mobile phone
- **OSEK**: Automotive electronics system
- **ITRON**: Used in many Japanese electronics devices test: print string backward, fill some missing part of program, ask what this code do.