**Pre-Emptive Task Scheduling using RTOS Kernel for Multitasking Applications**

**1. Aim:**

To implement the creation of tasks, scheduling of tasks using FreeRTOS APIs and ARM Cortex-M microcontroller.

**2. Introduction:**

A real-time system is one in which tasks have deadlines. If the software and hardware are not sufficiently responsive, then the task will not complete before its deadline, leading to a system failure. Real-time scheduling analysis gives us the mathematical methods to calculate the worst-case response time for each task in such a software system. We can compare these response times to our system’s deadlines in order to verify whether the system is schedulable (will always meet its deadlines).

If the system is not schedulable, then we have several options to make it schedulable. We could change the hardware (use a faster processor) if the customer budget allows it. We could improve the application software by speeding up the code or by reducing the amount of processing needed. We could also improve the scheduling approach by changing the balance of work performed in ISRs versus deferred activities, adding or changing task priorities, or adding preemption.

A real-time kernel (RTK) or a real-time operating system (RTOS) is designed to make it easier to create real-time systems. Preemptive scheduling is typically used to provide short response times. Prioritized task scheduling also reduces response times. The kernel is designed and built to execute with consistent and predictable timing, rather than with widely varying behavior.

**Example:**

The focus changes in most microcontroller applications. Often, meeting strict timing deadlines is critically important. Think about a motor controller or an automatic braking system in an electric vehicle. In these cases, if timing is off by even a few milliseconds, human lives could be in danger. As a result, an RTOS would be your best bet to manage several jobs running concurrently.

In this second example, a vehicle’s electronic control unit (ECU) might be in charge of controlling and assisting with braking. Job 2 monitors the driver’s input and helps apply the brakes and turns on the taillights. However, let’s say that our ECU gets notification that the car’s sensors have detected an impending crash. As a result, job 1 will preempt job 2 to control the brakes. This all assumes, of course, that this is a vehicle equipped with automatic braking assist.

A picture containing screenshot, line, diagram, plot

Description automatically generated

**3. Problem Statement:**

Consider a system with an MCU, two switches, and an RGB (red, green, blue) LED. When

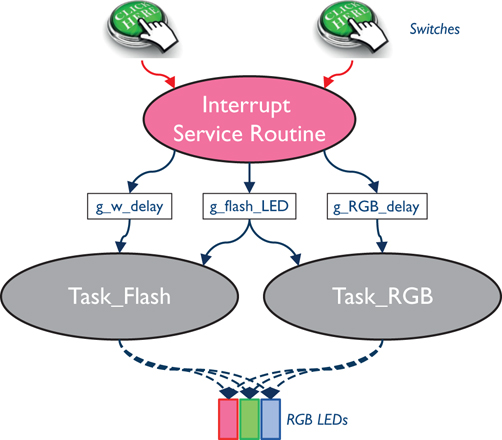
switch 1 is not pressed, the system displays a repeating sequence of colors (red, then green,

then blue). When switch 1 is pressed, the system makes the LED flash white (all LEDs on) and

off (all LEDs off) until the switch is released. Assume that the IRQ handler can tell the scheduler

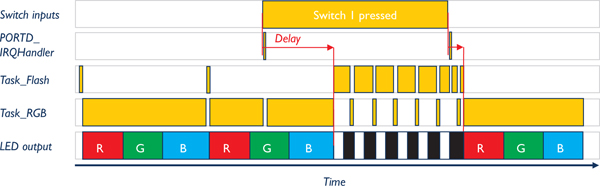
to change which task to run, and that tasks can preempt each other. IRQHandler starts

executing as soon as the switch changes from pressed to released or from released to pressed.



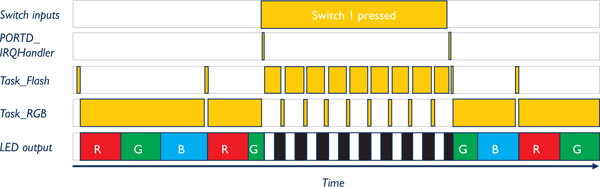
The cooperative multitasking approach we have been examining does not switch to running a different task until the currently running task yields the processor. This delays the system’s response. A preemptive multitasking approach uses task preemption so that an urgent task is not delayed by a currently running task of lower urgency. Consider that task A is already running, and task C becomes ready to run, perhaps due to an ISR executing and saving some deferred work for C. A preemptive scheduler can temporarily halt the processing of task A, run task C, and then resume the processing of task A.

The LED flasher with two tasks and an interrupt handler is shown in Figure. Pressing or releasing the switch triggers an interrupt event. The interrupt support hardware forces the CPU to preempt the currently running Task\_RGB, execute the code of PORTD\_IRQHandler (the ISR), and then resume Task\_RGB where it left off. However, we still have to wait for Task\_RGB to complete.



An LED flasher with two tasks and an interrupt handler is still delayed by Task\_RGB after interrupt.

With task preemption in Figure, we can use kernel features so that Task\_Flash (not Task\_RGB) runs after PORTD\_IRQHandler. Once Task\_Flash completes its work and waits, then Task\_RGB can run.



An LED flasher with task preemption. Task\_Flash prempts Task\_RGB in the green cycle when the switch is pressed.

**4. FreeRTOS:**

FreeRTOS is a market-leading real-time operating system [(RTOS)](https://www.freertos.org/about-RTOS.html) for microcontrollers and small microprocessors. Distributed freely under the MIT open source license, FreeRTOS includes a kernel and a growing set of IoT libraries suitable for use across all industry sectors. FreeRTOS is built with an emphasis on reliability and ease of use.

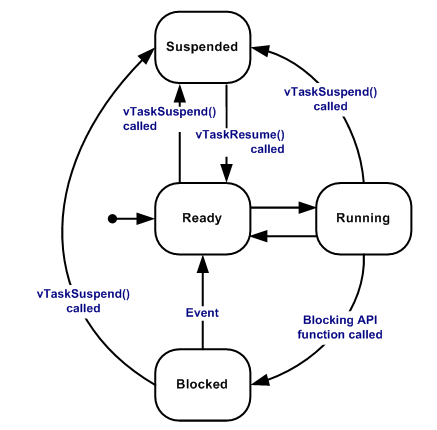
**4.1 Features:**

 Has a minimal ROM, RAM and processing overhead. Typically, an RTOS kernel binary image will be in the region of 6K to 12K bytes.

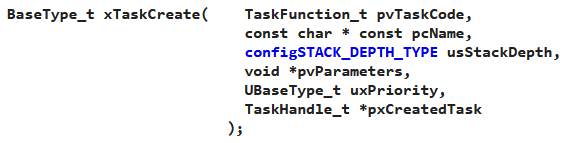
 Very simple - the core of the RTOS kernel is contained in [only 3 C files](https://www.freertos.org/a00017.html). The majority of the many files included in the .zip file download relate only to the numerous demonstration applications.

 Truly free for use in commercial applications .

**4.2 FreeRTOS Task State Diagram:**

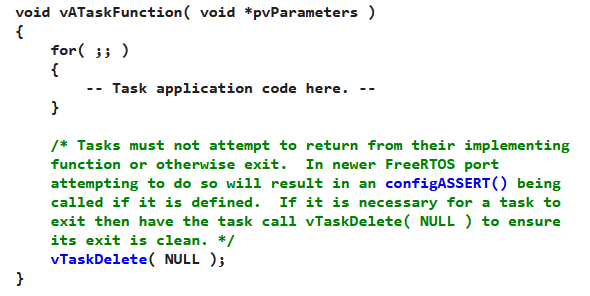


1. **Creating a Task**



1. **Implementing a Task**

A task should have the following structure:



**4.3 FreeRTOS vs. CMSIS-RTOS:**

It’s important to understand how STM32CubeIDE has bundled FreeRTOS. While FreeRTOS is an underlying software framework that allows for switching tasks, scheduling, etc., we won’t be making calls to FreeRTOS directly. ARM has created the CMSIS-RTOS library, which allows us to make calls to an underlying RTOS, thus improving the portability of code among various ARM processors.

**5. Program Code:**

**Procedure:**

1. Create two tasks called blink01 and blink02.
2. Assign the required stack and priority.
3. Implement the tasks blink01 and blink02 as per the application requirement.
4. Schedule the tasks.
5. Trace the task execution.
6. Change the priorities of the tasks and analyze the behavior.

/\* Includes ------------------------------------------------------------------\*/

#include "main.h"

#include "cmsis\_os.h"

/\* Definitions for blink01 \*/

osThreadId\_t blink01Handle;

const osThreadAttr\_t blink01\_attributes = {

.name = "blink01",

.stack\_size = 128 \* 4,

.priority = (osPriority\_t) osPriorityBelowNormal,

};

/\* Definitions for blink02 \*/

osThreadId\_t blink02Handle;

const osThreadAttr\_t blink02\_attributes = {

.name = "blink02",

.stack\_size = 128 \* 4,

.priority = (osPriority\_t) osPriorityNormal,

};

void StartBlink01(void \*argument);

void StartBlink02(void \*argument);

//GPIO Initialization

void configureLED(void)

{

RCC->AHB1ENR |=(1UL<<3);

GPIOD->MODER &= ~(0xFFUL<<12\*2);

GPIOD->MODER |= (0x55UL<<12\*2);

}

void msDelay(int msTime)

{

//Assume for loop take 12 clock cycles and system clock is 16MHz

int Time=msTime\*1333;

for(int i=0;i<Time;i++);

}

int main(void)

{

HAL\_Init();

configureLED();

osKernelInitialize();

blink01Handle = osThreadNew(StartBlink01, NULL, &blink01\_attributes);

/\* creation of blink02 \*/

blink02Handle = osThreadNew(StartBlink02, NULL, &blink02\_attributes);

/\* Start scheduler \*/

osKernelStart();

while (1)

{

}

}

void StartBlink01(void \*argument)

{

for(;;)

{

GPIOD->ODR ^= (0x1UL<<12);

msDelay(2000);

osDelay(2000);

}

}

/\* USER CODE END Header\_StartBlink02 \*/

void StartBlink02(void \*argument)

{

for(;;)

{

GPIOD->ODR ^= (0x2UL<<12);;

osDelay(200);

}

}

**6. Conclusion:**

Studied in detail about RTOS and implemented a simple RTOS system via embedded C code.

**7. References:**

1. FreeRTOS APIs: https://www.freertos.org/a00106.html