
# How to Build Your Own RTOS kernel? (Mero Samek; https://www.state-machine.com/quickstart/) Lesson/Video 22

***Sherif Hammad***

# *Agenda*

- **Can one main handle independent tasks?**
- **How to write two “while” independent main tasks?**
- **How to hack HW Cortex M4 Interrupt mechanism?**
- **Manual “Partial” switch of context between two tasks**
- **“Full” Manual switch of context between two tasks**

# Only "Blinky2" to Run!

```
void main_blinky1() {
    while (1) {
        BSP_ledGreenOn();
        BSP_delay(BSP_TICKS_PER_SEC / 4U);
        BSP_ledGreenOff();
        BSP_delay(BSP_TICKS_PER_SEC * 3U / 4U);
    }
}
void main_blinky2() {
    while (1) {
        BSP_ledBlueOn();
        BSP_delay(BSP_TICKS_PER_SEC / 2U);
        BSP_ledBlueOff();
        BSP_delay(BSP_TICKS_PER_SEC / 3U);
    }
}
/* background code: sequential with blocking version */
int main() {
    uint32_t volatile run = 0U;
    BSP_init();
        if (run) {
            main_blinky1();
        }
        else {
            main_blinky2();
        }
    while (1) {
    }
void SysTick_Handler(void) {
    ++l_tickCtr;
}
```

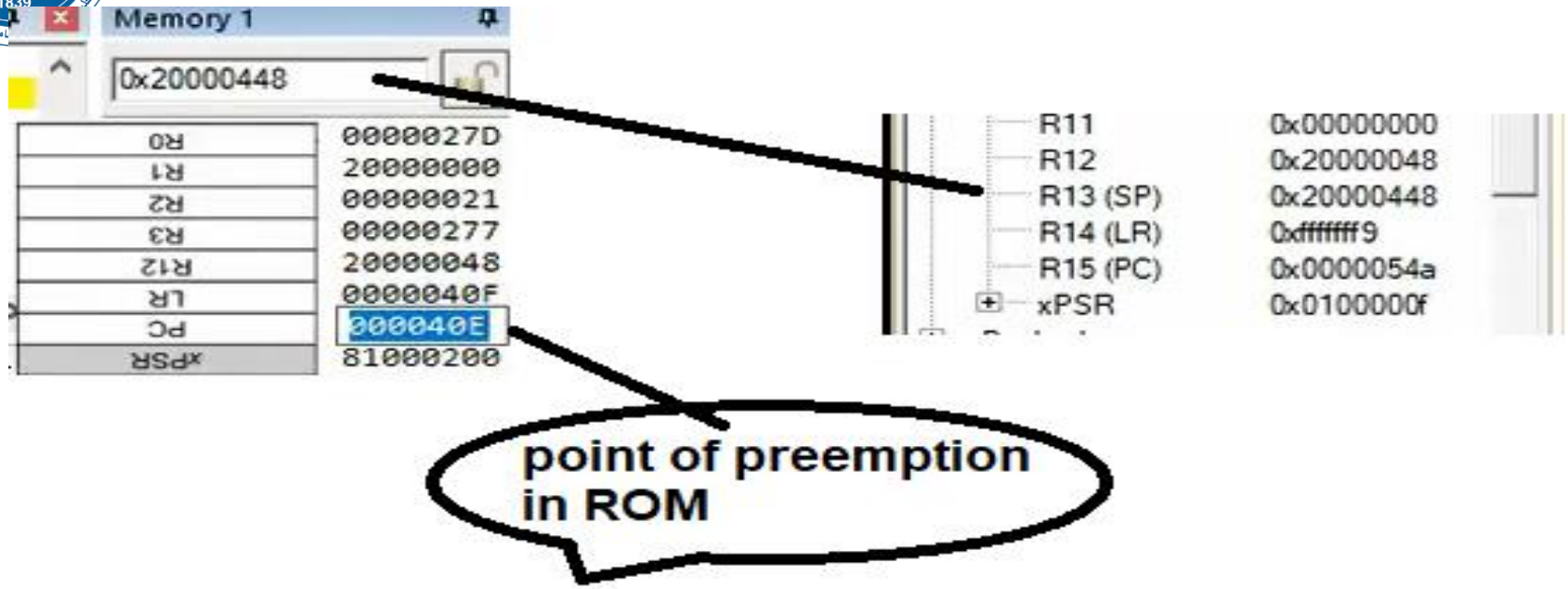
Kind of continuous task to blink Green led

Kind of continuous task to blink Blue led

Choose either blinky1 or blinky2 to run

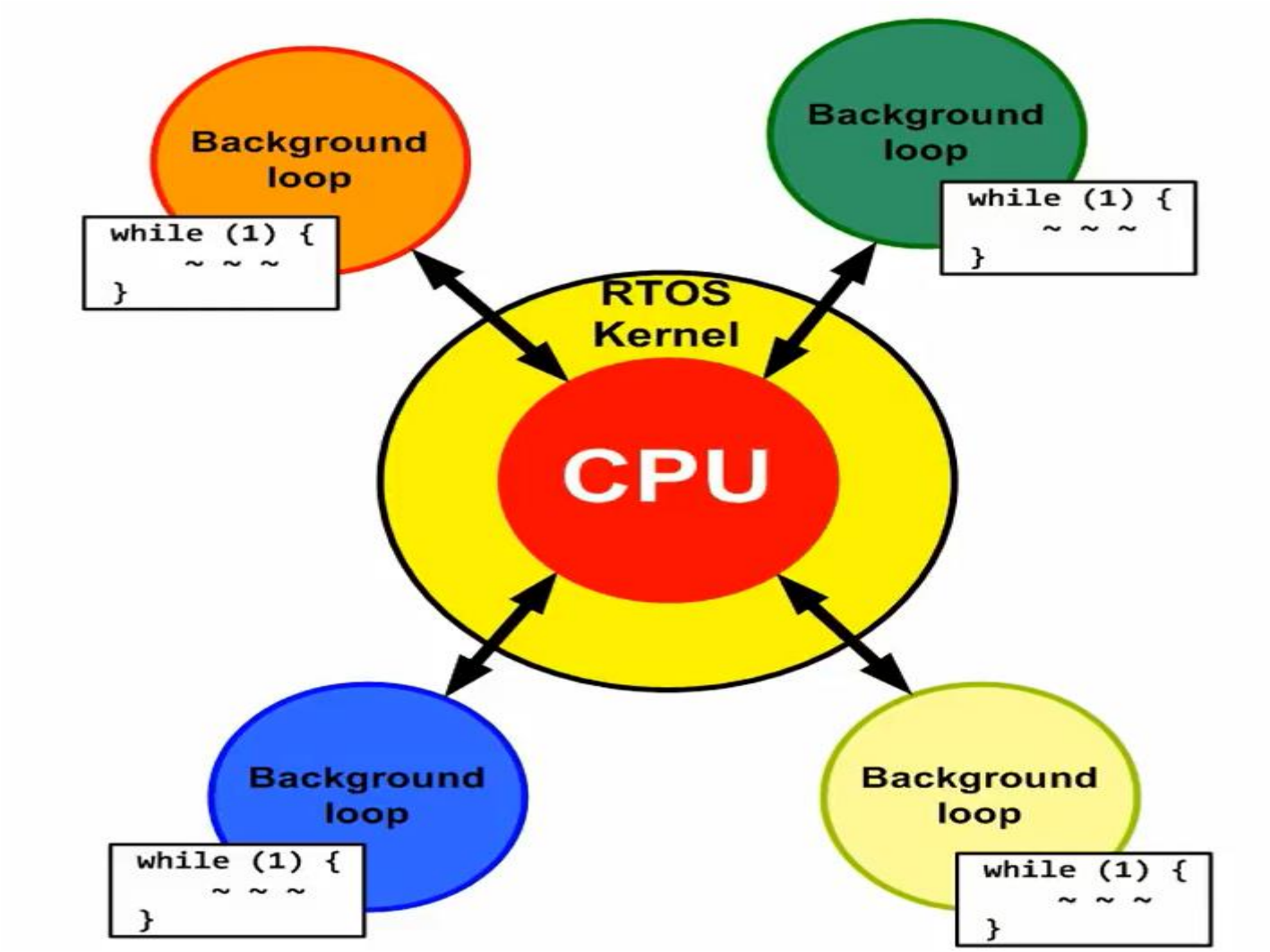
Systick is a foreground free running

# STEP 1 : ICD; Change to only “Blinky1” to Run!

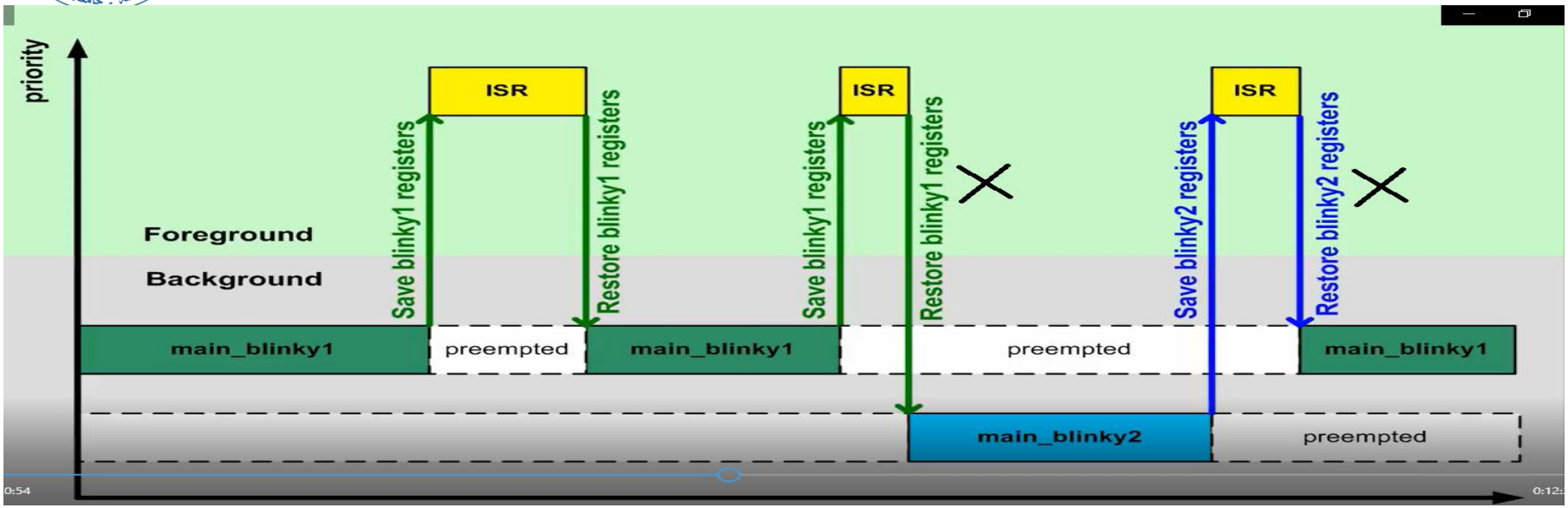


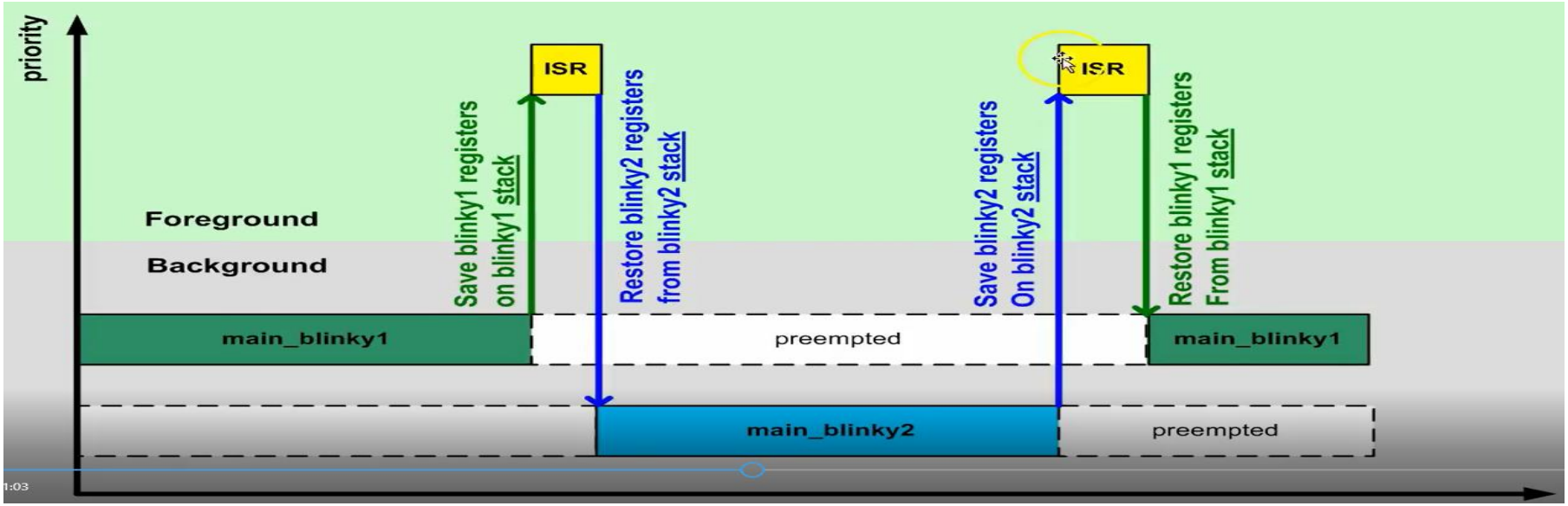

- **Put a BP inside Systick ISR, once hit, check change of context in Stack and find point of preemption inside “blinky2”**
- **What is the start address of “blinky1”?**
- **Is there a way to hack current context and change return address to run “blinky1”? (exploit the interrupt hardware processing)**
- **The exercise illustrates the general idea of multitasking on a single CPU, which is to switch the CPU between executing different background loops, like your main_blinky1 and main_blinky2 here.**

# Multitasking IDEA



# Manual Context Switch Problem & "Solution"



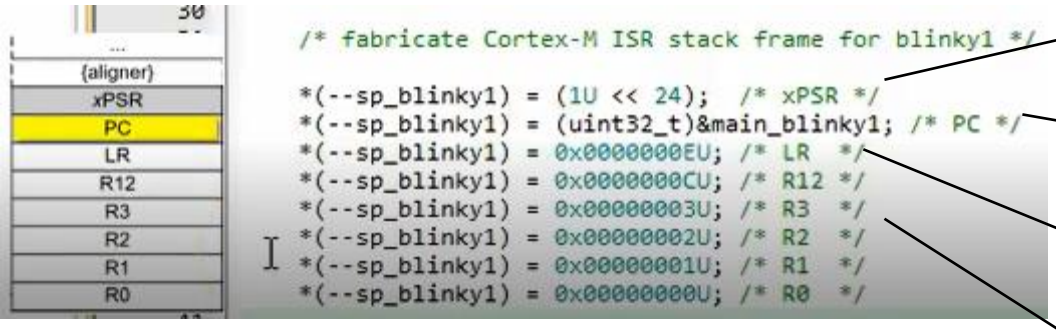
# STEP 2: Stack Initialization to Each Thread

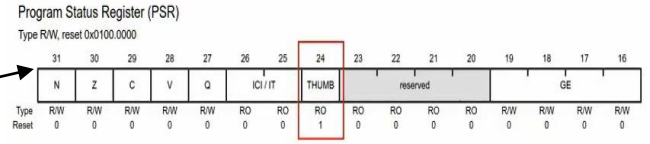
- **Registers saved for blinky1 cannot be restored for blinky2 and vice-versa.**
- **Add a user created stack to each thread.**
- **A stack is an area in RAM and a pointer that points to its current top.**

```
uint32_t stack_blinky1[40];
uint32_t *sp_blinky1 = &stack_blinky1[40];
```

```
uint32_t stack_blinky2[40];
uint32_t *sp_blinky2 = &stack_blinky2[40];
```

- **Pre-fill each thread's stack with a fabricated Cortex-M interrupt stack frame.**



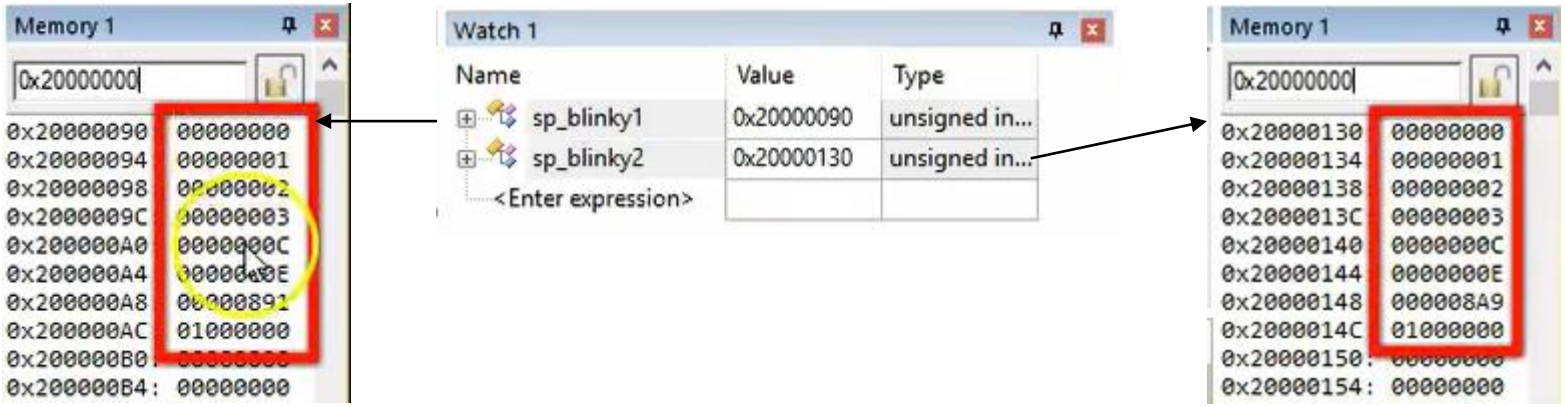

**PC is initialized to Thread first instruction in ROM**

**LR is not important since Thread is an infinite loop that never returns**

**Intialization insignificant values**

# STEP 2: Stack Initialization to Each Thread
# ISR Returns and Runs Blinky1

- Run to show that no Threads/LEDs are activated
- Check the stack initialization for both threads



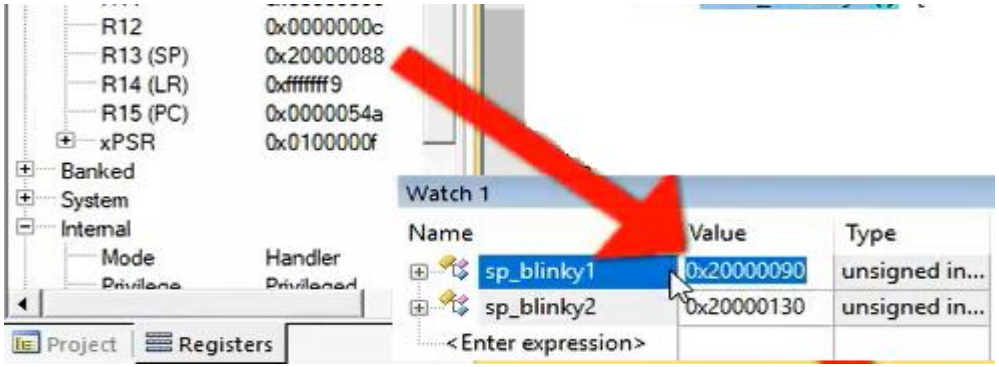

- Put/Hit BP at the end of ISR
- Change SP to be as

  sp_blinky1

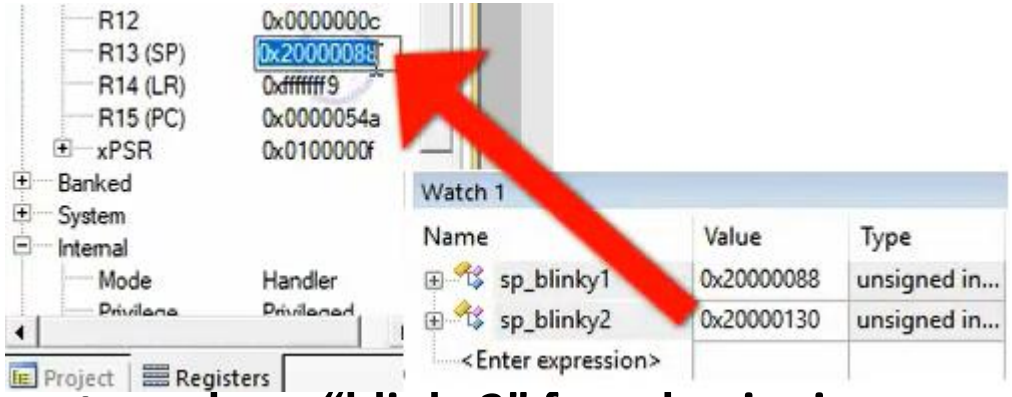

- This forces ISR to return to and run “blinky1” from beginning
- Remove BP and Blink Green Led

# STEP 2: ISR Preempts Blinky1 and Returns to Run Blinky2

- **Put/Hit BP at the end of ISR**
- **Copy from SP to sp_blinky1 pointer value since it's different from initial value**



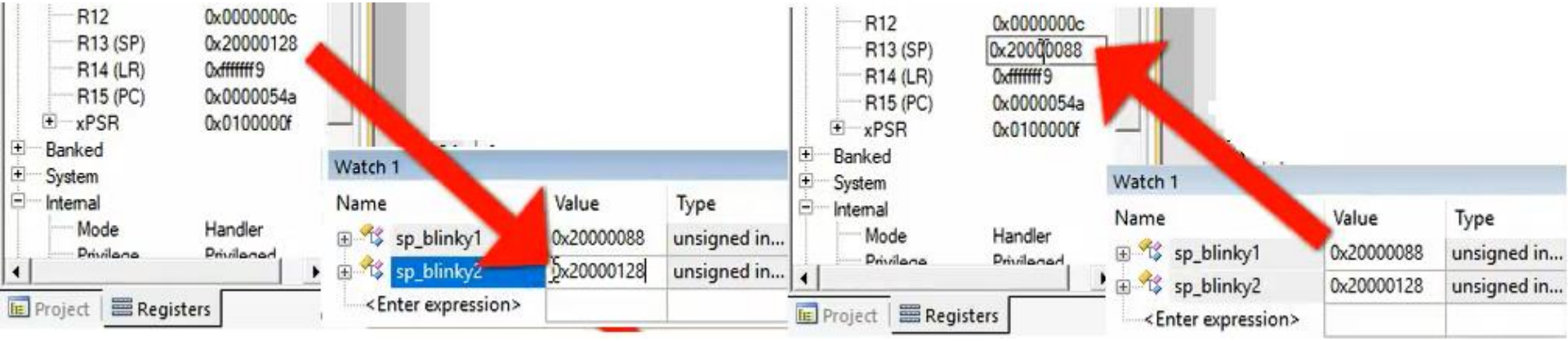

- **Change SP to be as sp_blinky2**




- **This forces ISR to return to and run "blinky2" from beginning**
- **Remove BP and Blink Blue Led**

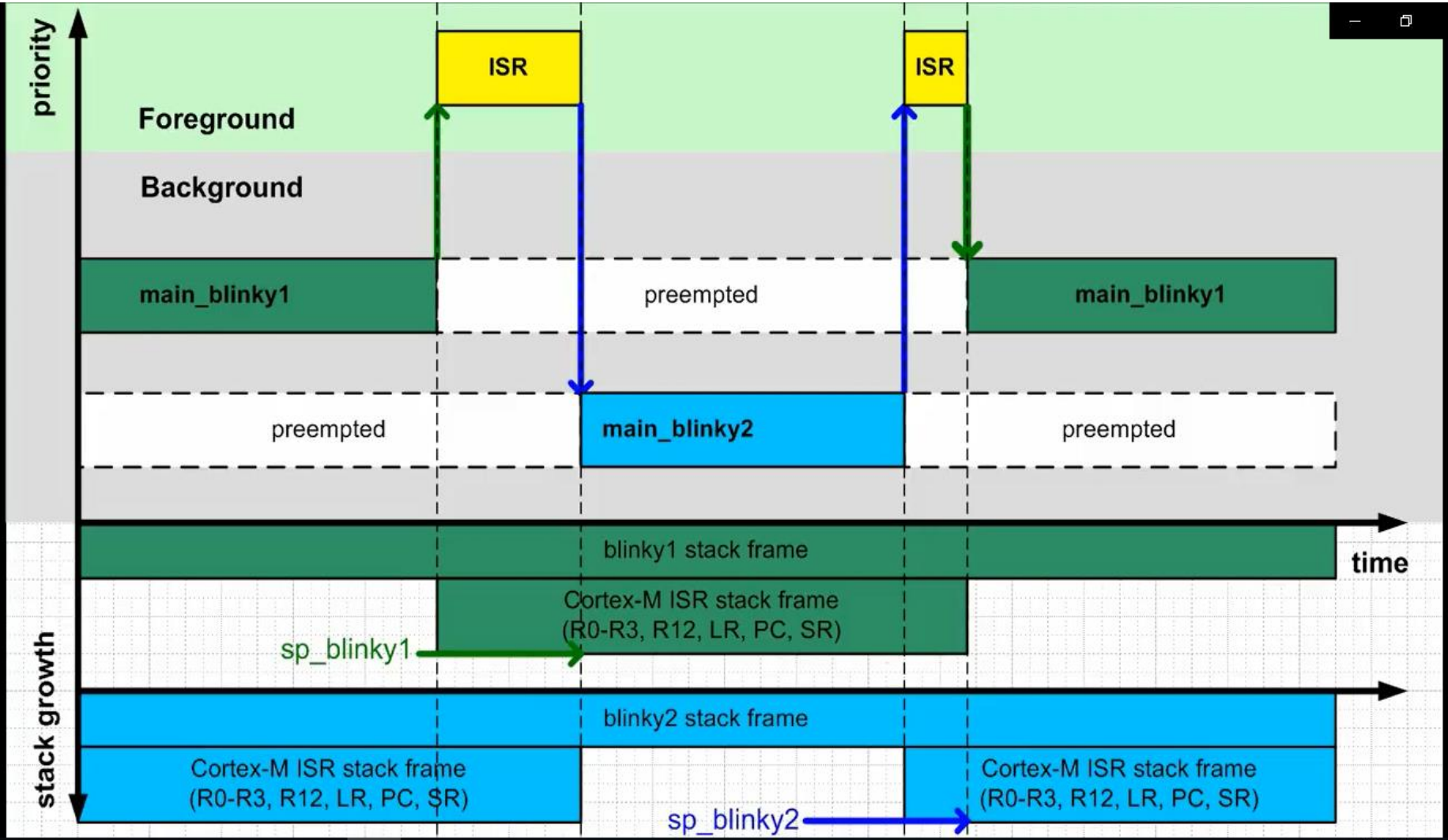
# STEP 2: Manual "Partial" Switch of Context

- **Manual context switch back to Blinky1**

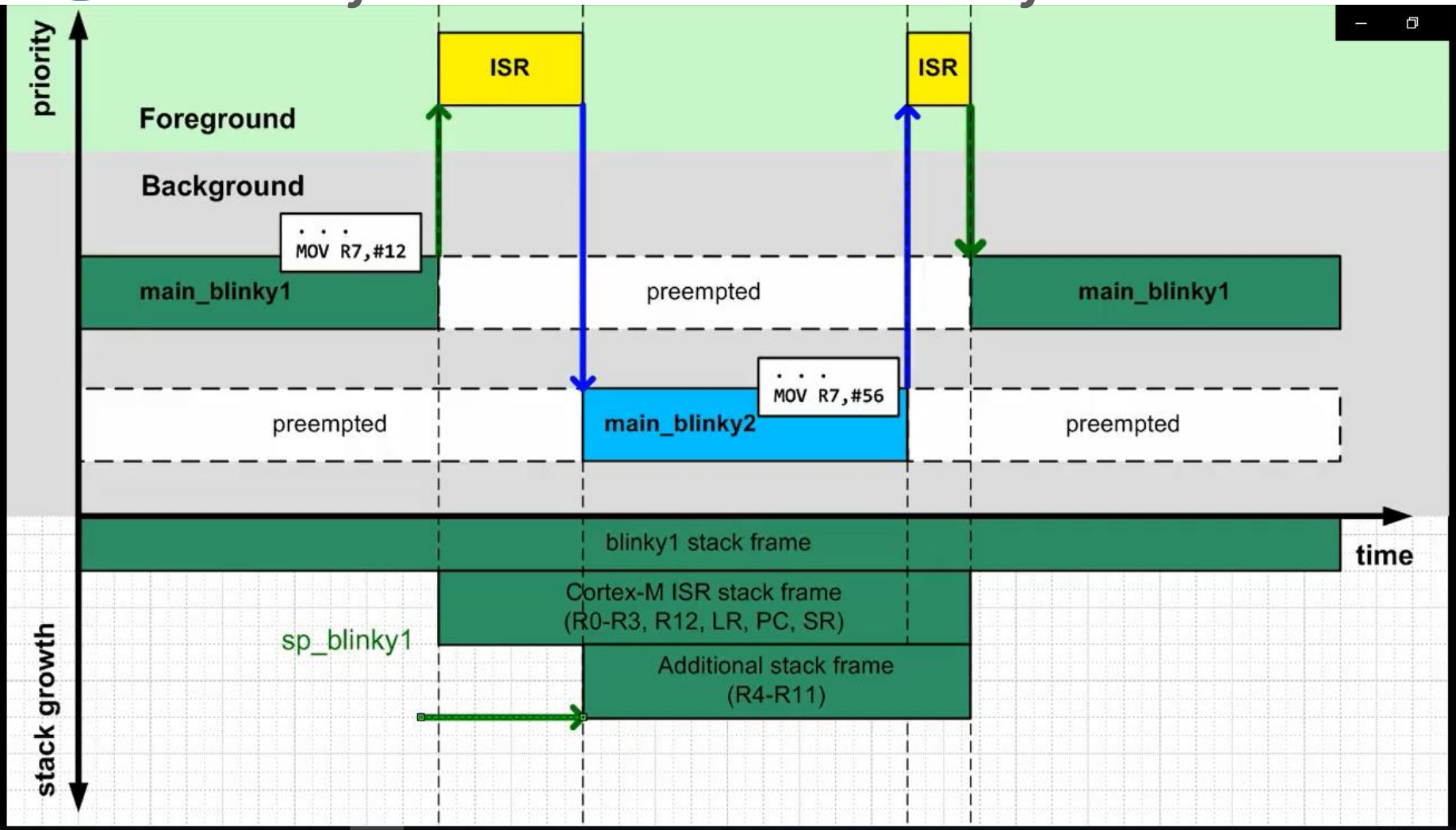



- **The sequence is as follows:**
  - **Timer ISR preempts Blinky2; means HW saves Blinky2 into its stack**
  - **BP hit; Above Manual switch; means ISR returns to Blinky1 instead**
  - **When Timer ISR preempts Blinky1; means HW saves Blinky1 into its stack**
  - **BP hit; Above Manual switch; means ISR returns to Blinky2 instead**
  - **Always returns exactly to the point of preemption**
  - **N.B.: It's always the HW interrupt mechanism that pushes and pops into user stack. Manually, you are only hacking or faking SP**

# STEP 2: Manual "Partial" Switch of Context



# STEP 3: Problem & Solution of Manual "Partial" Switch of Context



# STEP 3: Problem & Solution of Manual "Partial" Switch of Context

- **First: Initialize the user stack with full "context"**

```
/* fabricate Cortex-M ISR stack frame for blinky1 */
*(--sp_blinky1) = (1U << 24);  /* xPSR */
*(--sp_blinky1) = (uint32_t)&main_blinky1; /* PC */
*(--sp_blinky1) = 0x0000000EU; /* LR  */
*(--sp_blinky1) = 0x0000000CU; /* R12 */
*(--sp_blinky1) = 0x00000003U; /* R3  */
*(--sp_blinky1) = 0x00000002U; /* R2  */
*(--sp_blinky1) = 0x00000001U; /* R1  */
*(--sp_blinky1) = 0x00000000U; /* R0  */
/* additionally, fake registers R4-R11 */
*(--sp_blinky1) = 0x0000000BU; /* R11 */
*(--sp_blinky1) = 0x0000000AU; /* R10 */
*(--sp_blinky1) = 0x00000009U; /* R9 */
*(--sp_blinky1) = 0x00000008U; /* R8 */
*(--sp_blinky1) = 0x00000007U; /* R7 */
*(--sp_blinky1) = 0x00000006U; /* R6 */
*(--sp_blinky1) = 0x00000005U; /* R5 */
*(--sp_blinky1) = 0x00000004U; /* R4 */
```

```
/* fabricate Cortex-M ISR stack frame for blinky2 */
*(--sp_blinky2) = (1U << 24);  /* xPSR */
*(--sp_blinky2) = (uint32_t)&main_blinky2; /* PC */
*(--sp_blinky2) = 0x0000000EU; /* LR  */
*(--sp_blinky2) = 0x0000000CU; /* R12 */
*(--sp_blinky2) = 0x00000003U; /* R3  */
*(--sp_blinky2) = 0x00000002U; /* R2  */
*(--sp_blinky2) = 0x00000001U; /* R1  */
*(--sp_blinky2) = 0x00000000U; /* R0  */
/* additionally, fake registers R4-R11 */
*(--sp_blinky2) = 0x0000000BU; /* R11 */
*(--sp_blinky2) = 0x0000000AU; /* R10 */
*(--sp_blinky2) = 0x00000009U; /* R9 */
*(--sp_blinky2) = 0x00000008U; /* R8 */
*(--sp_blinky2) = 0x00000007U; /* R7 */
*(--sp_blinky2) = 0x00000006U; /* R6 */
*(--sp_blinky2) = 0x00000005U; /* R5 */
*(--sp_blinky2) = 0x00000004U; /* R4 */
```

- **When saving the current thread context, you need to manually save the R11 down to R4 on top of the current ISR stack frame.**
- **Subtract 0x20 from the SP CPU register before saving it in the thread's stack pointer**
- **When restoring the next thread, you need to manually restore R11 down to R4 from the thread's stack to the CPU registers.**
- **Add 0x20 to the thread's stack pointer before writing it to the CPU SP register.**

**Need Automated Switch Context Process**

# Questions ?


# How to Build Your Own RTOS kernel? (Mero Samek; https://www.state-machine.com/quickstart/) Lesson/Video 23

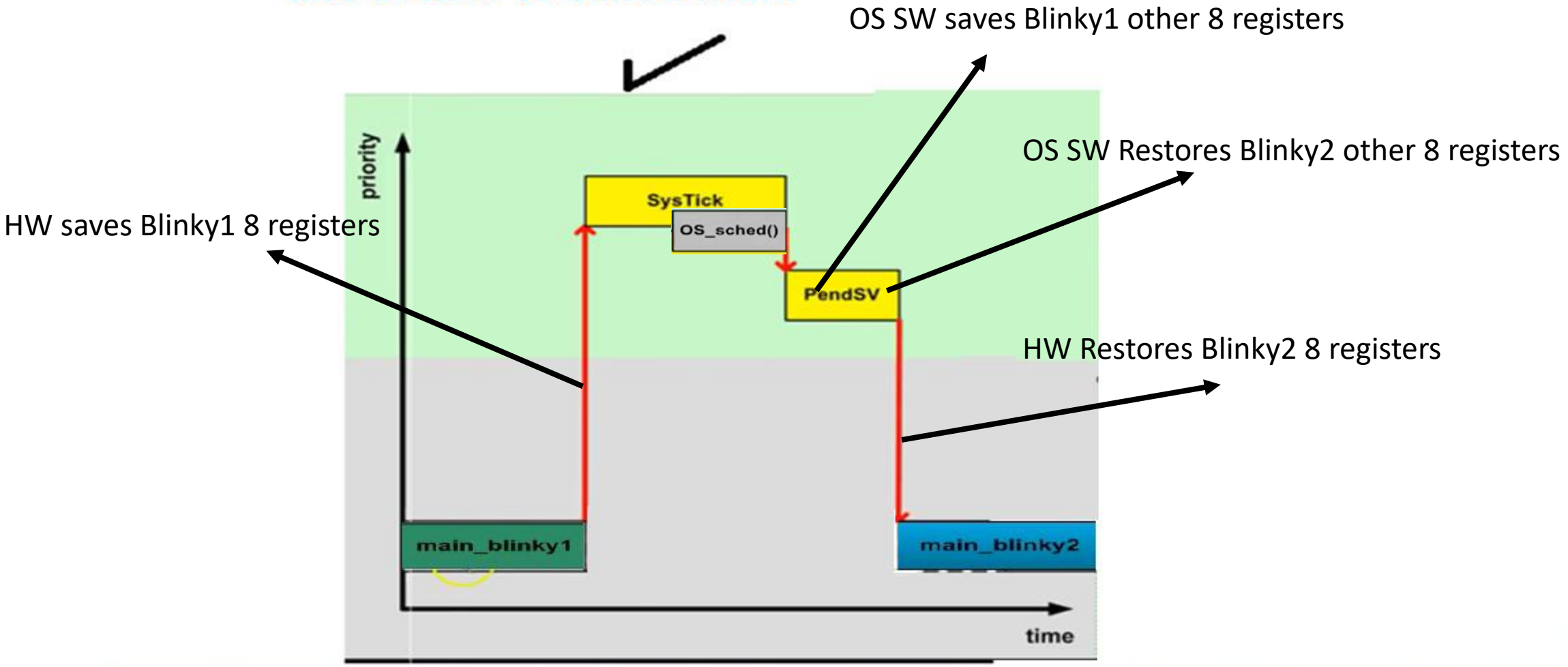
*Sherif Hammad*

# Agenda

- MiROS function and components
- OSThreadStart API definition
- How to hack HW Cortex M4 Interrupt mechanism?
- Manual “Partial” switch of context between two tasks
- “Full” Manual switch of context between two tasks

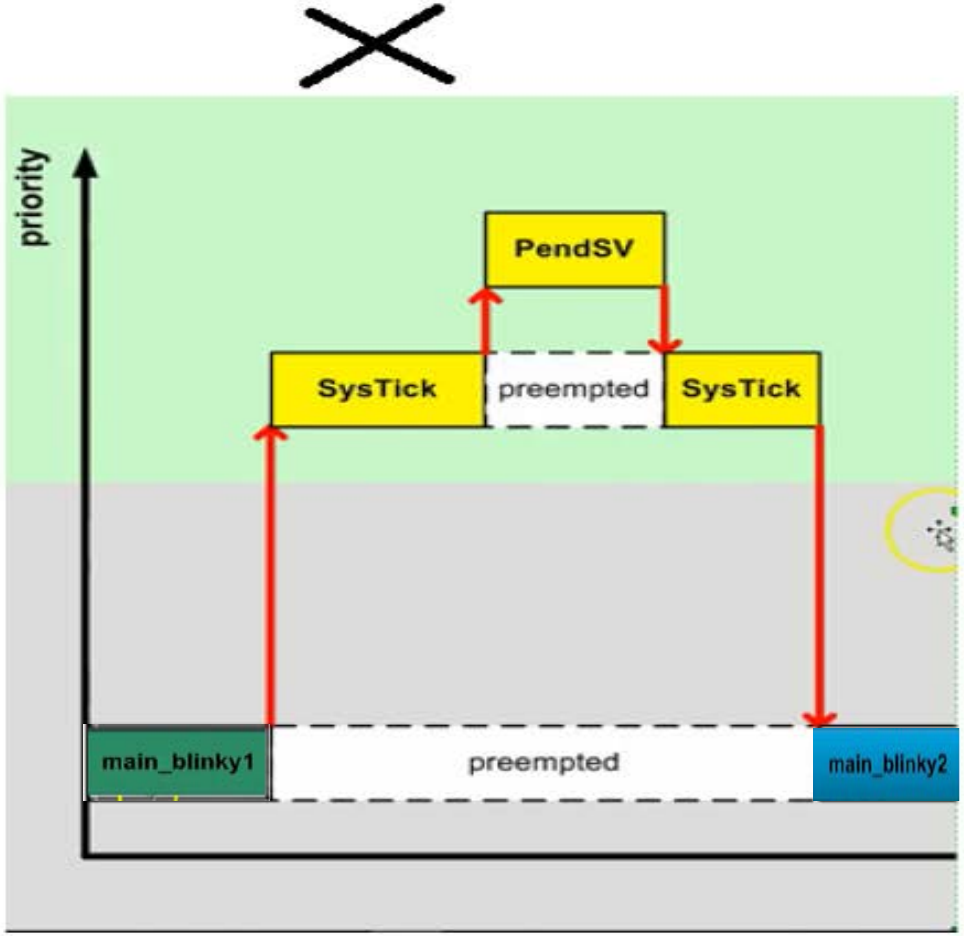
# Automated Context Switch; Cont.

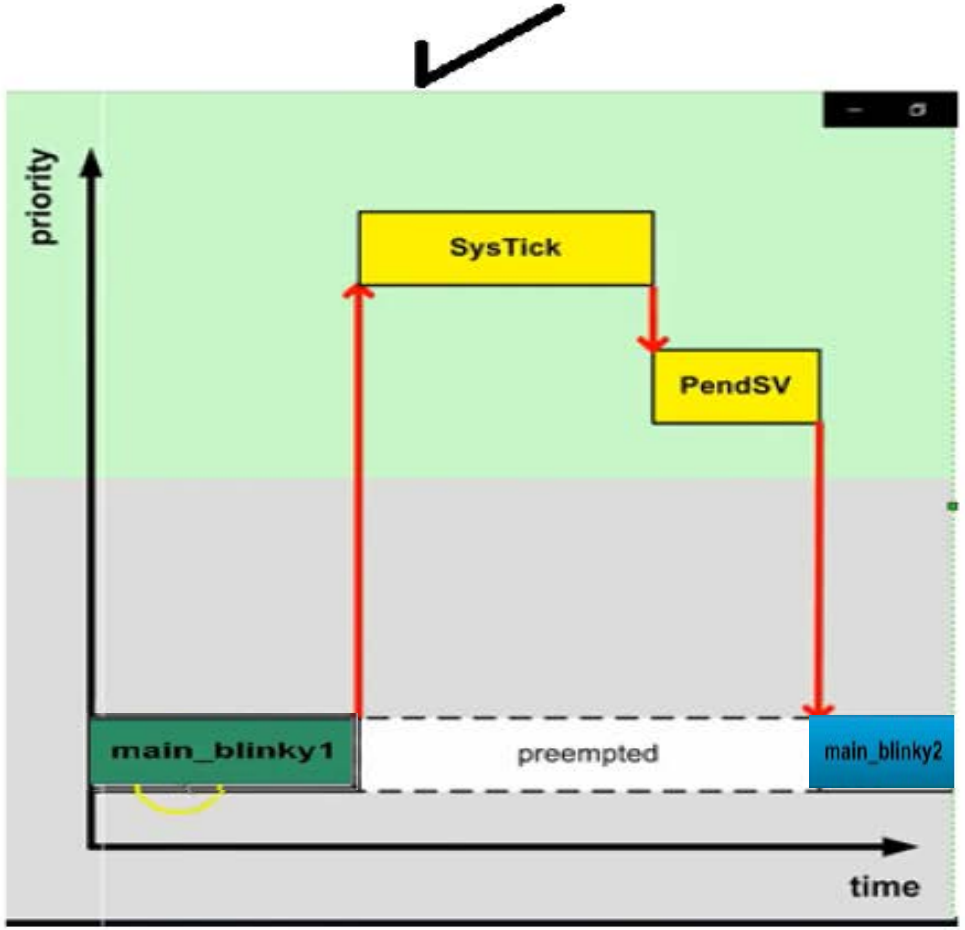
- One ISR is dedicated to context switch; PENDSV that should
  1) Be **Triggered** by the end of SysTick ISR
  2) Have lower priority than SysTick in order not to preempt it and return to called main



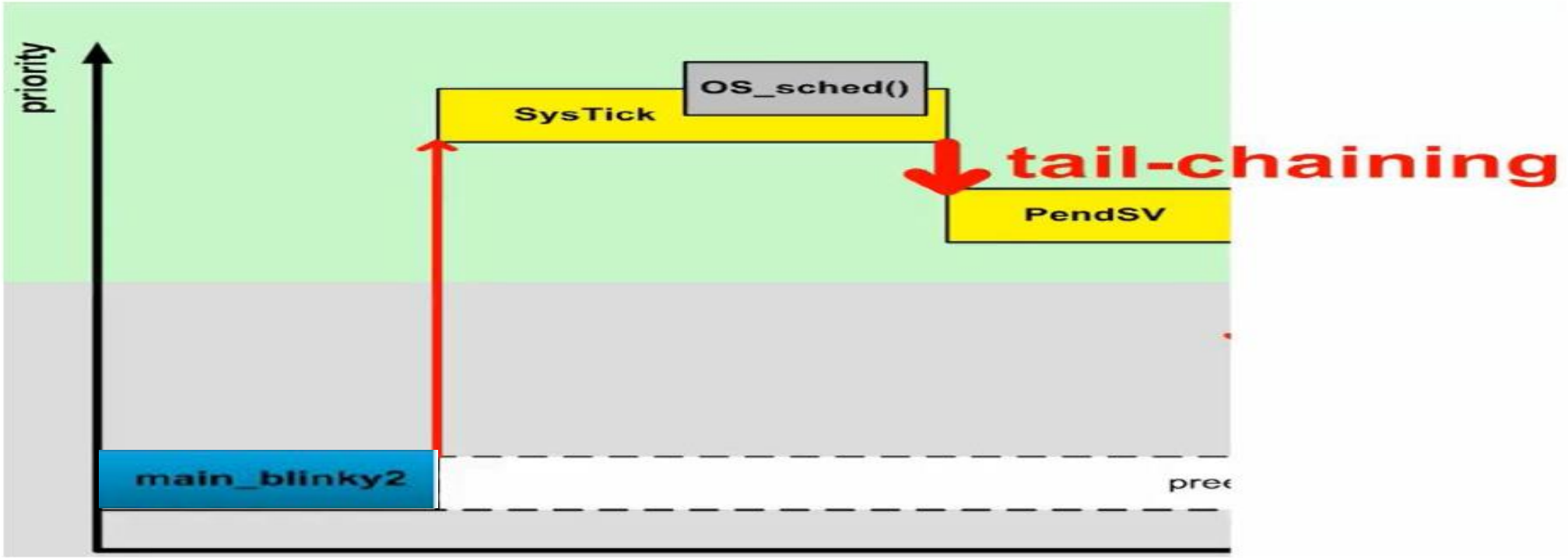
# Automated Context Switch; Cont.

- One ISR is dedicated to context switch; PENDSV that should
  1) Be **Triggered** by the end of SysTick ISR
  2) Have lower priority than SysTick in order not to preempt it and return to called main



# One Nice Side Comment




- Don’t worry of overheads of systick interrupt HW POP and PendSV PUSH
- Tail chaining results in HW optimization for Cortex M4
- The ARM Cortex-M core skips the popping and pushing registers in the hardware optimization called "tail-chaining". So, the overhead is comparable to a simple function call.

# Exception entry and return



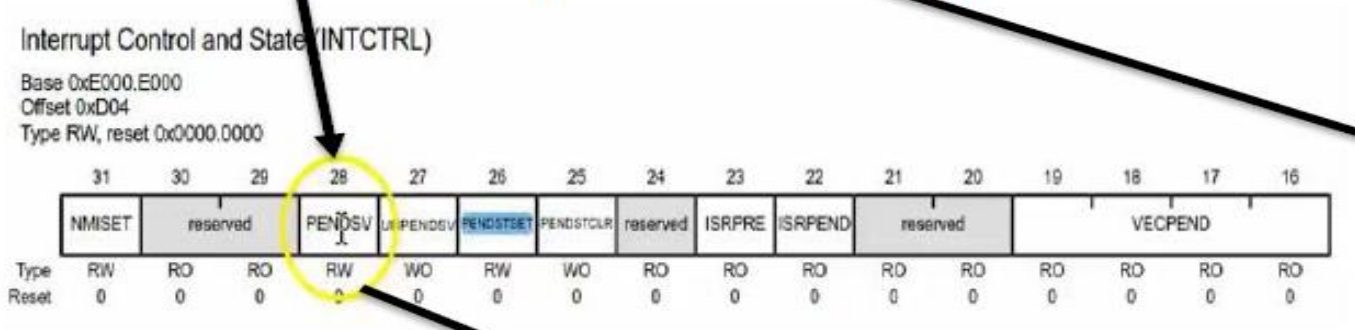
- Preemption and interrupt nesting
  - The execution of an interrupt handler can be preempted by an exception having a higher priority


- Tail-chaining
  - When an interrupt is pending on the completion of an exception handler, the context store is skipped and the control is immediately transferred to the new exception handler when the previous handler is completed.

# Automated Context Switch

- Context switch must happen during the return from SysTick interrupt
- Context switch cannot be coded in standard C; rather be in Assembly
- One ISR is dedicated to context switch; PENDSV that should

1) Be Triggered by the end of SysTick ISR




main.c | miros.c | miros.h | bsp.c

```
11  #define LED_GREEN (1U << 3)
12
13  static uint32_t volatile l_tickC
14
15  void SysTick_Handler(void) {
16      ++l_tickCtr;
17
18      __disable_irq();
19      OS_sched();
20      __enable_irq();
21  }
22
```

critical section

main.c | miros.c | miros.h | bsp.c

```
41  void OS_sched(void) {
42      /* OS_next = ... */
43      if (OS_next != OS_curr) {
44          *(uint32_t volatile *)0xE000ED04 = (1U << 28);
45      }
46  }
```

Trigger PENDSV

N.B.: OS_SCHED has no scheduling algorithm yet

# Automated Context Switch; Cont.

- One ISR is dedicated to context switch; PENDSV that should
  2) Have lower priority than SysTick in order not to preempt it and return to called main

System Handler Priority 3 (SYSPRI3)

Base 0xE000.E000
Offset 0xD20
Type RW, reset 0x0000.0000

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| TICK | | | reserved | | | | | PENDSV | | | reserved | | | | |
| Type | RW | RW | RW | RO | RO | RO | RO | RO | RW | RW | RW | RO | RO | RO | RO | RO |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

main.c | miros.c | miros.h | bsp.c

```
23 void BSP_init(void) {
24     SYSCTL->RCGCGPIO  |= (1U << 5); /* enable Run mode for GPIOF */
25     SYSCTL->GPIOHBCTL |= (1U << 5); /* enable AHB for GPIOF */
26     GPIOF_AHB->DIR |= (LED_RED | LED_BLUE | LED_GREEN);
27     GPIOF_AHB->DEN |= (LED_RED | LED_BLUE | LED_GREEN);
28
29     SystemCoreClockUpdate();
30     SysTick_Config(SystemCoreClock / BSP_TICKS_PER_SEC);
31
32     /* set the SysTick interrupt priority (highest) */
33     NVIC_SetPriority(SysTick_IRQn, 0U);
34
35     __enable_irq();
```

```
27  /* background code:
28  int main() {
29      BSP_init();
30      OS_init();
31
```

main.c | miros.c | miros.h | bsp.c

```
32
33  OSThread * volatile OS_curr; /* pointer to the current thread */
34  OSThread * volatile OS_next; /* pointer to the next thread to run */
35
36  void OS_init(void) {
37      /* set the PendSV interrupt priority to the lowest level 0xFF */
38      *(uint32_t volatile *)0xE000ED20 |= (0xFFU << 16);
39  }
```

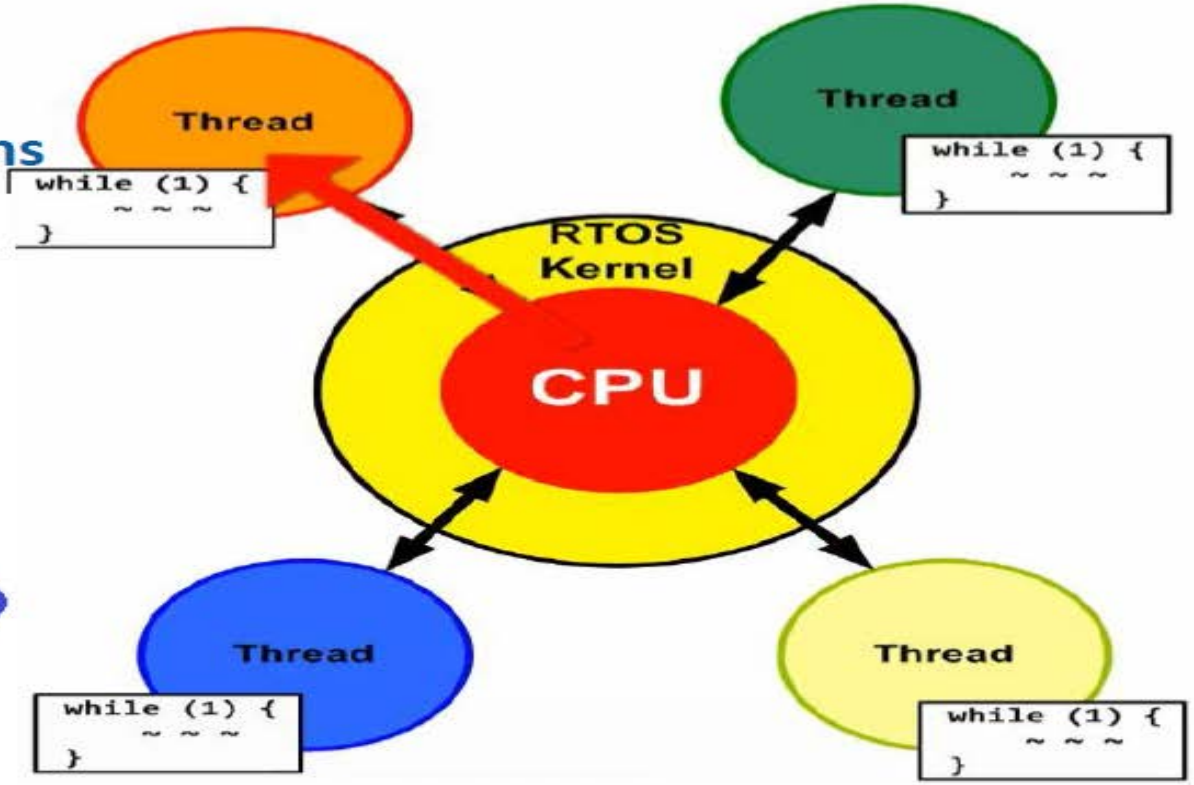
# Minimal Real Time OS

- Blinked both green and blue LEDs "Threads" independently.
- Manually switch the CPU between the two threads (context switch).
- MiROS is RTOS from scratch
- MiROS.h will contain RTOS APIs
- MiROS.c will contain RTOS APIs definitions

```
#ifndef MIROS_H
#define MIROS_H

/* Thread Control Block (TCB) */
typedef struct {
    void *sp; /* stack pointer */
    /* ... other attributes associated with a thread */
} OSThread;

typedef void (*OSThreadHandler)();

void OS_init(void);

/* this function must be called with interrupts DISABLED */
void OS_sched(void);

void OSThread_start(
    OSThread *me,
    OSThreadHandler threadHandler,
    void *stkSto, uint32_t stkSize);

#endif /* MIROS_H */
```

which thread to scheule next (Scheduler)

Looks Like xTaskCreate



# OSThread Start Definition and Calls

## Definition

```
void OSThread_start(
    OSThread *me,
    OSThreadHandler threadHandler,
    void *stkSto, uint32_t stkSize)
{
    /* round down the stack top to the 8-byte boundary
    * NOTE: ARM Cortex-M stack grows down from hi -> low memory */
    uint32_t *sp = (uint32_t *)((((uint32_t)stkSto + stkSize) / 8) * 8);
    uint32_t *stk_limit;
    *(--sp) = (1U << 24);  /* xPSR */
    *(--sp) = (uint32_t)threadHandler; /* PC */
    *(--sp) = 0x0000000EU; /* LR  */
    *(--sp) = 0x0000000CU; /* R12 */
    *(--sp) = 0x00000003U; /* R3  */
    *(--sp) = 0x00000002U; /* R2  */
    *(--sp) = 0x00000001U; /* R1  */
    *(--sp) = 0x00000000U; /* R0  */
    /* additionally, fake registers R4-R11 */
    *(--sp) = 0x0000000BU; /* R11 */
    *(--sp) = 0x0000000AU; /* R10 */
    *(--sp) = 0x00000009U; /* R9 */
    *(--sp) = 0x00000008U; /* R8 */
    *(--sp) = 0x00000007U; /* R7 */
    *(--sp) = 0x00000006U; /* R6 */
    *(--sp) = 0x00000005U; /* R5 */
    *(--sp) = 0x00000004U; /* R4 */
    /* save the top of the stack in the thread's attibute */
    me->sp = sp;
    /* round up the bottom of the stack to the 8-byte boundary */
    stk_limit = (uint32_t *)(((((uint32_t)stkSto - 1U) / 8) + 1U) * 8);
    /* pre-fill the unused part of the stack with 0xDEADBEEF */
    for (sp = sp - 1U; sp >= stk_limit; --sp) {
        *sp = 0xDEADBEEFU;
    }
}
```

Task Stack Pointer

Task Name

**CALLS**

```
/* background code: sequential with blocking version */
int main() {
    BSP_init();
    OS_init();

    /* fabricate Cortex-M ISR stack frame for blinky1 */
    OSThread_start(&blinky1,
                   &main_blinky1,
                   stack_blinky1, sizeof(stack_blinky1));

    /* fabricate Cortex-M ISR stack frame for blinky2 */
    OSThread_start(&blinky2,
                   &main_blinky2,
                   stack_blinky2, sizeof(stack_blinky2));

    while (1) {
    }
```

Stack Array Name

Stack Array Size

# Verifying Stack Initialization

stkSto==stk_limit

## Blinky1

Memory 1

0x20000000

```
0x20000010:  00000000 00000000
0x20000018:  DEADBEEF DEADBEEF
0x20000020:  DEADBEEF DEADBEEF
0x20000028:  DEADBEEF DEADBEEF
0x20000030:  DEADBEEF DEADBEEF
0x20000038:  DEADBEEF DEADBEEF
0x20000040:  DEADBEEF DEADBEEF
0x20000048:  DEADBEEF DEADBEEF
0x20000050:  DEADBEEF DEADBEEF
0x20000058:  DEADBEEF DEADBEEF
0x20000060:  DEADBEEF DEADBEEF
0x20000068:  DEADBEEF DEADBEEF
0x20000070:  DEADBEEF DEADBEEF
0x20000078:  00000004 00000005
0x20000080:  00000006 00000007
0x20000088:  00000008 00000009
0x20000090:  0000000A 0000000B
0x20000098:  00000000 00000001
0x200000A0:  00000002 00000003
0x200000A8:  0000000C 0000000E
0x200000B0:  00000921 01000000
0x200000B8:  00000000 00000000
```

Command | Memory 1

stkSize

## Blinky2

Memory 1

0x200000a0

```
0x200000B0:  00000921 01000000
0x200000B8:  DEADBEEF DEADBEEF
0x200000C0:  DEADBEEF DEADBEEF
0x200000C8:  DEADBEEF DEADBEEF
0x200000D0:  DEADBEEF DEADBEEF
0x200000D8:  DEADBEEF DEADBEEF
0x200000E0:  DEADBEEF DEADBEEF
0x200000E8:  DEADBEEF DEADBEEF
0x200000F0:  DEADBEEF DEADBEEF
0x200000F8:  DEADBEEF DEADBEEF
0x20000100:  DEADBEEF DEADBEEF
0x20000108:  DEADBEEF DEADBEEF
0x20000110:  DEADBEEF DEADBEEF
0x20000118:  00000004 00000005
0x20000120:  00000006 00000007
0x20000128:  00000008 00000009
0x20000130:  0000000A 0000000B
0x20000138:  00000000 00000001
0x20000140:  00000002 00000003
0x20000148:  0000000C 0000000E
0x20000150:  00000939 01000000
0x20000158:  00000000 00000000
```

Command | Memory 1

# STEP 3: PENDSV Implementation of Context Switch

- **PENDSV should be in Assembly**
- **PENDSV code should be in critical section**
- **First, here is a framework in C:**

```
void PendSV_Handler(void) {
    void *sp;

    __disable_irq();
    if (OS_curr != (OSThread *)0) {
        /* push registers r4-r11 on the stack */
        OS_curr->sp = sp;
    }
    sp = OS_next->sp;
    OS_curr = OS_next;
    /* pop registers r4-r11 */
    __enable_irq();
}
```

- Fake SP till assembly coding
- critical section
- Current & Next Threads are initially zero
- Place holder of PUSH assembly instruction
- If not initialization state, make the next thread as current

# STEP 3: PENDSV Implementation of Context Switch

```c
void PendSV_Handler(void) {
    void *sp;

    __disable_irq();
    if (OS_curr != (OSThread *)0) {
        /* push registers r4-r11 on the stack */
        OS_curr->sp = sp;
    }
    sp = OS_next->sp;
    OS_curr = OS_next;
    /* pop registers r4-r11 */
    __enable_irq();
}
```

```
__asm
void PendSV_Handler(void) {
    IMPORT  OS_curr  /* extern variable */
    IMPORT  OS_next  /* extern variable */

    /* __disable_irq(); */
    CPSID           I

    /* if (OS_curr != (OSThread *)0) { */
    LDR             r1,=OS_curr
    LDR             r1,[r1,#0x00]
    CBZ             r1,PendSV_restore

    /*      push registers r4-r11 on the stack */
    PUSH            {r4-r11}

    /*      OS_curr->sp = sp; */
    LDR             r1,=OS_curr
    LDR             r1,[r1,#0x00]
    STR             sp,[r1,#0x00]
    /* } */

PendSV_restore
    /* sp = OS_next->sp; */
    LDR             r1,=OS_next
    LDR             r1,[r1,#0x00]
    LDR             sp,[r1,#0x00]

    /* OS_curr = OS_next; */
    LDR             r1,=OS_next
    LDR             r1,[r1,#0x00]
    LDR             r2,=OS_curr
    STR             r1,[r2,#0x00]

    /* pop registers r4-r11 */
    POP             {r4-r11}

    /* __enable_irq(); */
    CPSIE           I

    /* return to the next thread */
    BX              lr
}
```


# Testing The Process of Context Switch

- Try the code to switch between Blinky1 and Blinky2 with automated switch of context
- Still we don't have a scheduling algorithm. So far, we trigger either thread manually

# SysTick Fires During Main: While Loop

D:\sherif hammad\2020\CSE316-318_FAll_18-19-spring'20\Mero-Semec\lesson23\lesson.uvprojx - µVision

File Edit View Project Flash Debug Peripherals Tools SVCS Window Help

GPIO registers (PORTF)

**Registers**

| Register | Value |
|---|---|
| Core | |
| R0 | 0x00000111 |
| R1 | 0x00000001 |
| R2 | 0x00000002 |
| R3 | 0x00000003 |
| R4 | 0x00000000 |
| R5 | 0x20000158 |
| R6 | 0x00000000 |
| R7 | 0x00000000 |
| R8 | 0x00000000 |
| R9 | 0x00000000 |
| R10 | 0x00000970 |
| R11 | 0x00000000 |
| R12 | 0x0000000c |
| R13 (SP) | 0x200005a0 |
| R14 (LR) | 0xfffffff9 |
| R15 (PC) | 0x00000642 |
| xPSR | 0x8100000f |
| Banked | |
| System | |
| Internal | |
| Mode | Handler |
| Privilege | Privileged |
| Stack | MSP |
| States | 0 |
| Sec | 0.00000000 |
| FPU | |

**Disassembly**

```
    16:     ++l_tickCtr;
    17:
0x00000642 4805      LDR      r0,[pc,#20]  ; @0x00000658
0x00000644 6800      LDR      r0,[r0,#0x00]
```

miros.c | miros.h | bsp.c | startup_TM4C123GH6PM.s

```
 7
 8  /* on-board LEDs */
 9  #define LED_RED   (1U << 1)
10  #define LED_BLUE  (1U << 2)
11  #define LED_GREEN (1U << 3)
12
13  static uint32_t volatile l_tickCtr;
14
15  void SysTick_Handler(void) {
16      ++l_tickCtr;
17
18      __disable_irq();
19      OS_sched();
20      __enable_irq();
21  }
22
23  void BSP_init(void) {
24      SYSCTL->RCGCGPIO  |= (1U << 5); /* enable Run mode for
25      SYSCTL->GPIOHBCTL |= (1U << 5); /* enable AHB for GPIO
26      GPIOF_AHB->DIR |= (LED_RED | LED_BLUE | LED_GREEN);
27      GPIOF_AHB->DEN |= (LED_RED | LED_BLUE | LED_GREEN);
28
29      SystemCoreClockUpdate();
30      SysTick_Config(SystemCoreClock / BSP_TICKS_PER_SEC);
31
32      /* set the SysTick interrupt priority (highest) */
33      NVIC_SetPriority(SysTick_IRQn, 0U);
34
```

ISR Calls OS_sched

SysTick ISR Saved Context

**Watch 1**

| Name | Value |
|---|---|
| OS_curr | 0x00000000 |
| OS_next | 0x00000000 |
| <Enter expression> | |

No Task Switch of Context

**Call Stack + Locals**

| Name | Location/V |
|---|---|
| SysTick_Handler | 0x00000642 |
| main | 0x00000000 |

**Memory 1**

Address: 0x20000010

```
0x2000001C: DEADBEEF DEADBEEF DEADBEEF
0x20000028: DEADBEEF DEADBEEF DEADBEEF
0x20000034: DEADBEEF DEADBEEF DEADBEEF
0x20000040: DEADBEEF DEADBEEF DEADBEEF
0x2000004C: DEADBEEF DEADBEEF DEADBEEF
0x20000058: DEADBEEF DEADBEEF DEADBEEF
0x20000064: DEADBEEF DEADBEEF DEADBEEF
0x20000070: DEADBEEF DEADBEEF 00000004
0x2000007C: 00000005 00000006 00000007
0x20000088: 00000008 00000009 0000000A
0x20000094: 0000000B 00000000 00000001
0x200000A0: 00000002 00000003 0000000C
0x200000AC: 0000000E 00000921 01000000
```

**Memory 2**

Address: 0x200000B8

```
0x200000B8: DEADBEEF DEADBEEF DEADBEEF
0x200000C4: DEADBEEF DEADBEEF DEADBEEF
0x200000D0: DEADBEEF DEADBEEF DEADBEEF
0x200000DC: DEADBEEF DEADBEEF DEADBEEF
0x200000E8: DEADBEEF DEADBEEF DEADBEEF
0x200000F4: DEADBEEF DEADBEEF DEADBEEF
0x20000100: DEADBEEF DEADBEEF DEADBEEF
0x2000010C: DEADBEEF DEADBEEF DEADBEEF
0x20000118: 00000004 00000005 00000006
0x20000124: 00000007 00000008 00000009
0x20000130: 0000000A 0000000B 00000000
0x2000013C: 00000001 00000002 00000003
0x20000148: 0000000C 0000000E 00000939
0x20000154: 01000000 00000000 00000000
```

**Memory 3**

0x200005a0

```
0x200005A0: 00000000
0x200005A4: FFFFFFF9
0x200005A8: 00000111
0x200005AC: 00000001
0x200005B0: 00000002
0x200005B4: 00000003
0x200005B8: 0000000C
0x200005BC: 00000905
0x200005C0: 00000906
0x200005C4: 81000000
```

Project | Registers

Start code execution | Stellaris ICDI | t1: 0.00000000 sec | L:16 C:1 | CAP NUM SCRL OVR R/W

# Manual Switch of Context to Blinky1 Thread

D:\sherif hammad\2020\CSE316-318_FAll_18-19-spring'20\Mero-Semec\lesson23\lesson.uvprojx - µVision

File Edit View Project Flash Debug Peripherals Tools SVCS Window Help

GPIO registers (PORTF)

**Registers**

| Register | Value |
|---|---|
| Core | |
| R0 | 0x20000004 |
| R1 | 0x00000000 |
| R2 | 0x00000002 |
| R3 | 0x00000003 |
| R4 | 0x00000000 |
| R5 | 0x20000158 |
| R6 | 0x00000000 |
| R7 | 0x00000000 |
| R8 | 0x00000000 |
| R9 | 0x00000000 |
| R10 | 0x00000970 |
| R11 | 0x00000000 |
| R12 | 0x0000000c |
| R13 (SP) | 0x200005a0 |
| R14 (LR) | 0x00000653 |
| R15 (PC) | 0x000005e4 |
| xPSR | 0x2100000F |
| Banked | |
| System | |
| Internal | |
| Mode | Handler |
| Privilege | Privileged |
| Stack | MSP |
| States | 0 |
| Sec | 0.00000000 |
| FPU | |

Project | Registers

**Disassembly**

```
    44:         *(uint32_t volatile *)0xE000ED04 = (1
    45:     }
0x000005E4 F04F5080  MOV     r0,#0x10000000
0x000005E8 4903      LDR     r1,[pc,#12]  ; @0x000005F8
```

main.c | miros.c | miros.h | bsp.c | startup_TM4C123GH6PM.s

```
35
36  void OS_init(void) {
37      /* set the PendSV interrupt priority to the lowest
38      *(uint32_t volatile *)0xE000ED20 |= (0xFF << 16);
39  }
40
41  void OS_sched(void) {
42      /* OS_next = ... */
43      if (OS_next != OS_curr) {
44          *(uint32_t volatile *)0xE000ED04 = (1U << 28);
45      }
46  }
47
48  void OSThread_start(
49      OSThread *me,
50      OSThreadHandler threadHandler,
51      void *stkSto, uint32_t stkSize)
52  {
53      /* round down the stack top to the 8-byte boundary
54      * NOTE: ARM Cortex-M stack grows down from hi -> low m
55      */
56      uint32_t *sp = (uint32_t *)(((uint32_t)stkSto + stkSi
57      uint32_t *stk_limit;
58
59      *(--sp) = (1U << 24);  /* xPSR */
60      *(--sp) = (uint32_t)threadHandler; /* PC */
61      *(--sp) = 0x0000000EU; /* LR  */
62      *(--sp) = 0x0000000CU; /* R12 */
```

**Watch 1**

| Name | Value |
|---|---|
| OS_curr | 0x00000000 |
| OS_next | 0x20000004 &blinky1 |
| <Enter expression> | |

Switch Context to Blinky1 Task

Fire PendSV ISR

PendSV ISR Pending till SysTick ISR ends

**Call Stack + Locals**

| Name | Location/V |
|---|---|
| OS_sched | 0x000005E4 |
| SysTick_Handler | 0x00000642 |
| main | 0x00000000 |

**Memory 1**

Address: 0x20000010

```
0x2000001C: DEADBEEF DEADBEEF DEADBEEF
0x20000028: DEADBEEF DEADBEEF DEADBEEF
0x20000034: DEADBEEF DEADBEEF DEADBEEF
0x20000040: DEADBEEF DEADBEEF DEADBEEF
0x2000004C: DEADBEEF DEADBEEF DEADBEEF
0x20000058: DEADBEEF DEADBEEF DEADBEEF
0x20000064: DEADBEEF DEADBEEF DEADBEEF
0x20000070: DEADBEEF DEADBEEF 00000004
0x2000007C: 00000005 00000006 00000007
0x20000088: 00000008 00000009 0000000A
0x20000094: 0000000B 00000000 00000001
0x200000A0: 00000002 00000003 0000000C
0x200000AC: 0000000E 00000921 01000000
```

**Memory 2**

Address: 0x200000B8

```
0x200000B8: DEADBEEF DEADBEEF DEADBEEF
0x200000C4: DEADBEEF DEADBEEF DEADBEEF
0x200000D0: DEADBEEF DEADBEEF DEADBEEF
0x200000DC: DEADBEEF DEADBEEF DEADBEEF
0x200000E8: DEADBEEF DEADBEEF DEADBEEF
0x200000F4: DEADBEEF DEADBEEF DEADBEEF
0x20000100: DEADBEEF DEADBEEF DEADBEEF
0x2000010C: DEADBEEF DEADBEEF DEADBEEF
0x20000118: 00000004 00000005 00000006
0x20000124: 00000007 00000008 00000009
0x20000130: 0000000A 0000000B 00000000
0x2000013C: 00000001 00000002 00000003
0x20000148: 0000000C 0000000E 00000939
0x20000154: 01000000 00000000 00000000
```

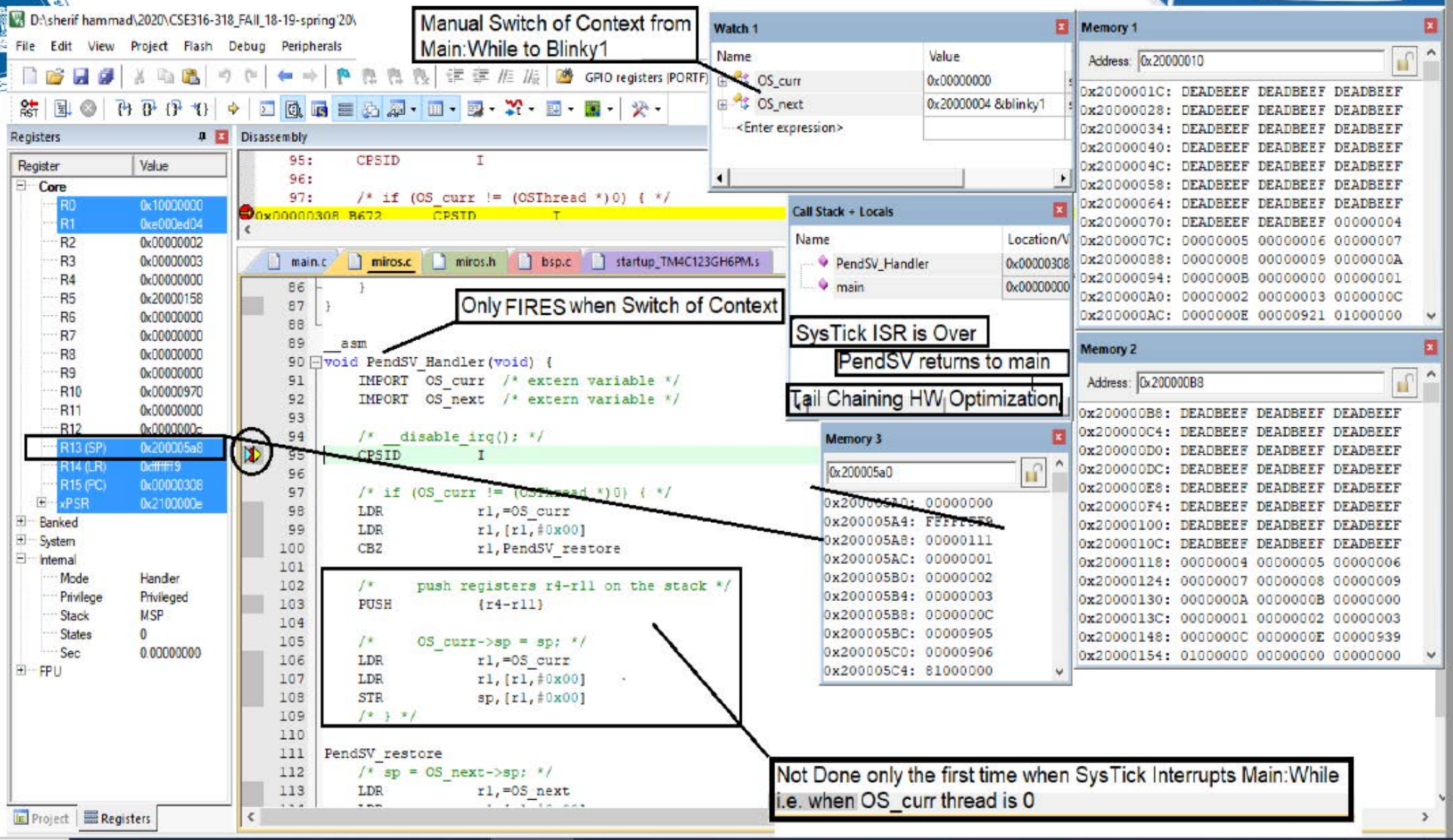
**Memory 3**

0x200005a0

```
0x200005A0: 00000000
0x200005A4: FFFFFFF9
0x200005A8: 00000111
0x200005AC: 00000001
0x200005B0: 00000002
0x200005B4: 00000003
0x200005B8: 0000000C
0x200005BC: 00000905
0x200005C0: 00000906
0x200005C4: 81000000
```

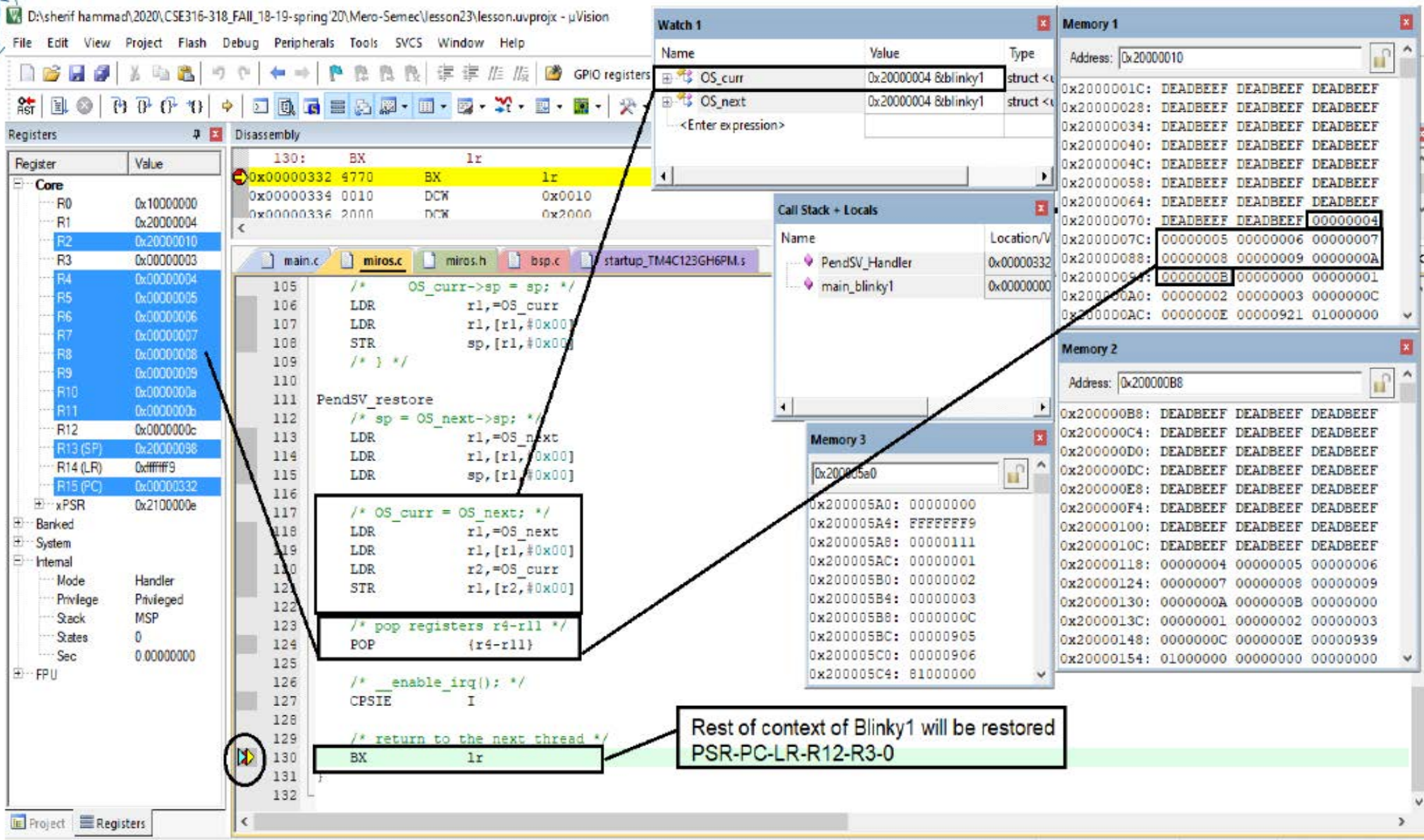
Stellaris ICDI | t1: 0.00000000 sec | L:44 C:1 | CAP NUM SCRL OVR R

# PendSV Fires During SysTick ISR; Serviced After

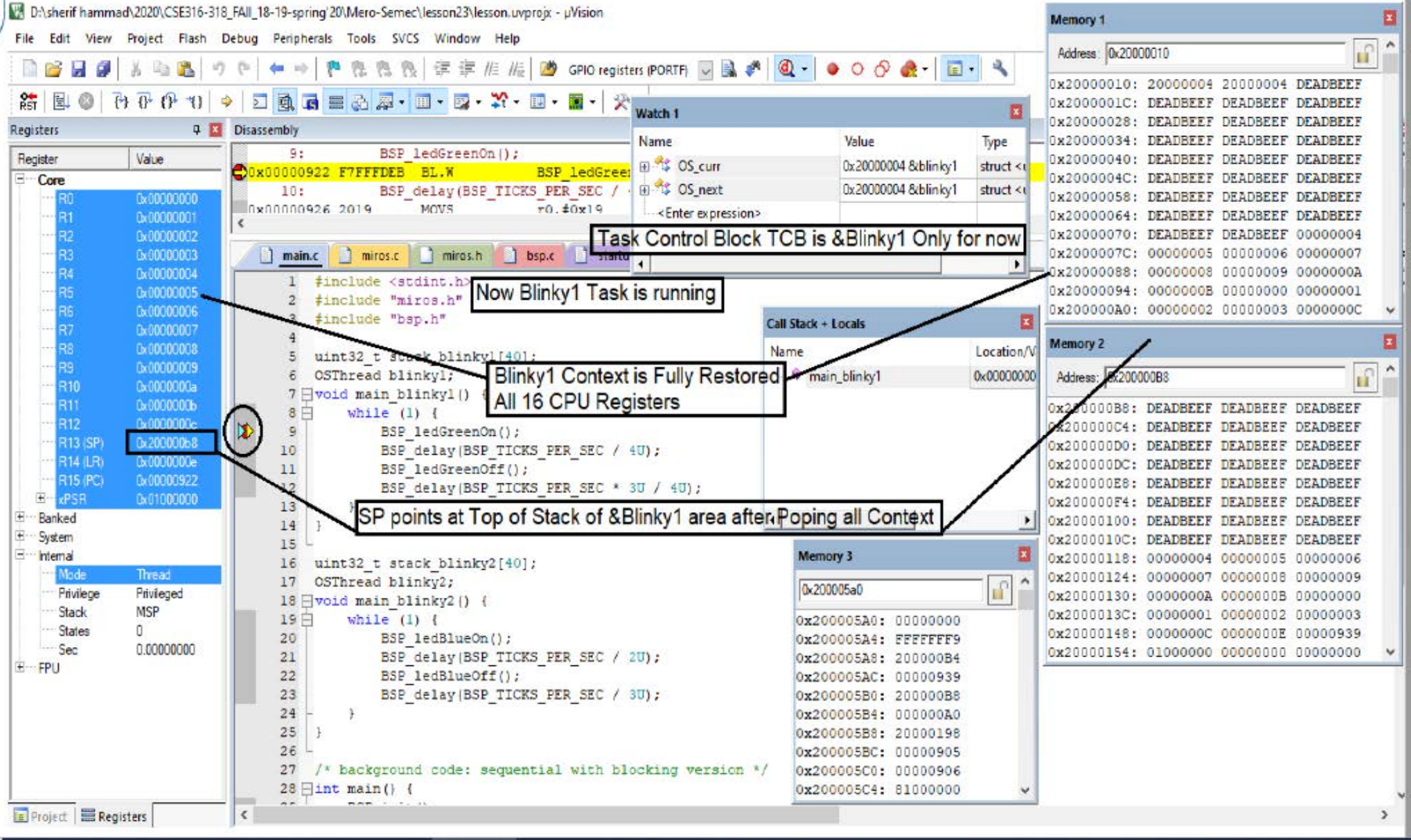


# Beginning of Blinky1 Thread Restore of Context

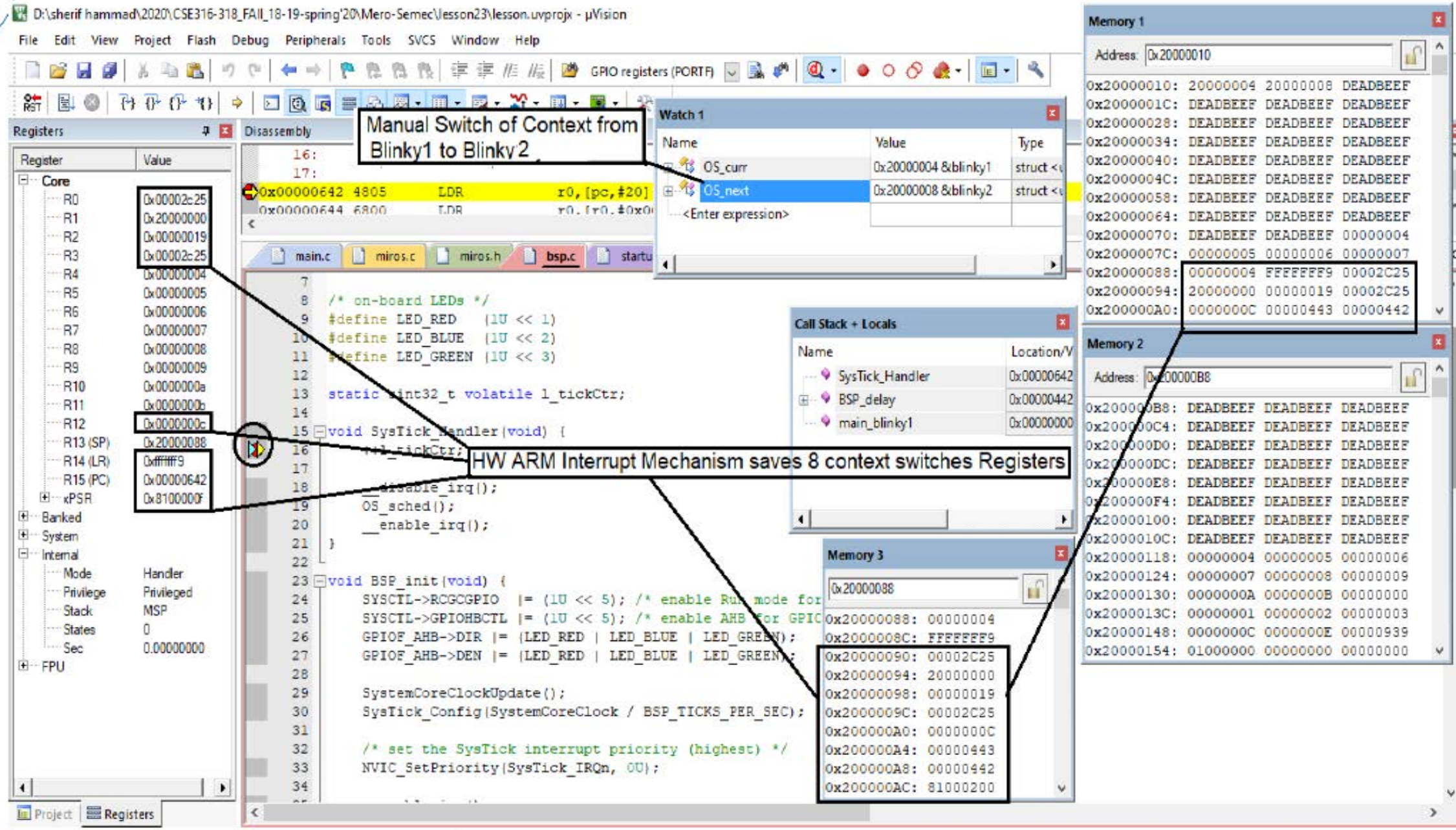
# Restore of Blinky1 Thread R4-R14 by SW POP



# Complete Restore of Blinky1 Remaining 8 ISR HW Registers



# Manual Switch of Context to Blinky2 Thread



# Save Blinky1 Thread R4-R11 by SW Push

D:\sherif hammad\2020\CSE316-318_FALL_18-19-spring'20\Mero-Semec\lesson23\lesson.uvprojx - µVision

File Edit View Project Flash Debug Peripherals Tools SVCS Window Help

GPIO registers (PORTF)

**Registers**

| Register | Value |
|---|---|
| Core | |
| R0 | 0x10000000 |
| R1 | 0x20000004 |
| R2 | 0x00000019 |
| R3 | 0x00002c25 |
| R4 | 0x00000004 |
| R5 | 0x00000005 |
| R6 | 0x00000006 |
| R7 | 0x00000007 |
| R8 | 0x00000008 |
| R9 | 0x00000009 |
| R10 | 0x0000000a |
| R11 | 0x0000000b |
| R12 | 0x0000000c |
| R13 (SP) | 0x20000090 |
| R14 (LR) | 0xfffffff9 |
| R15 (PC) | 0x00000310 |
| xPSR | 0x2100000e |
| Banked | |
| System | |
| Internal | |
| Mode | Handler |
| Privilege | Privileged |
| Stack | MSP |
| States | 0 |
| Sec | 0.00000000 |
| FPU | |

Project Registers

**Disassembly**

```
103:     PUSH        {r4-r11}
104:
105:     /*     OS_curr->sp = sp; */
0x00000310 E92D0FF0  PUSH        {r4-r11}
```

main.c | miros.c | miros.h | bsp.c | startu

```
94      /* __disable_irq(); */
95      CPSID       I
96
97      /* if (OS_curr != (OSThread *)0) { */
98      LDR         r1,=OS_curr
99      LDR         r1,[r1,#0x00]
100     CBZ         r1,PendSV_restore
101
102     /*     push registers r4-r11 on the stack */
103     PUSH        {r4-r11}
104
105     /*     OS_curr->sp = sp; */
106     LDR         r1,=OS_curr
107     LDR         r1,[r1,#0x00]
108     STR         sp,[r1,#0x00]
109     /* } */
110
111 PendSV_restore
112     /* sp = OS_next->sp; */
113     LDR         r1,=OS_next
114     LDR         r1,[r1,#0x00]
115     LDR         sp,[r1,#0x00]
116
117     /* OS_curr = OS_next; */
118     LDR         r1,=OS_next
119     LDR         r1,[r1,#0x00]
120     LDR         r2,=OS_curr
121     STR         r1,[r2,#0x00]
```

Blinky1 remaining context registers will be saved

**Watch 1**

| Name | Value | Type |
|---|---|---|
| OS_curr | 0x20000004 &blinky1 | struct <u |
| OS_next | 0x20000008 &blinky2 | struct <u |
| <Enter expression> | | |

**Call Stack + Locals**

| Name | Location/V |
|---|---|
| PendSV_Handler | 0x00000308 |
| BSP_delay | 0x00000442 |
| main_blinky1 | 0x00000000 |

PendSV will not return to Blinky1

**Memory 3**

Address: 0x20000088

```
0x20000088: 00000004
0x2000008C: FFFFFFF9
0x20000090: 00002C25
0x20000094: 20000000
0x20000098: 00000019
0x2000009C: 00002C25
0x200000A0: 0000000C
0x200000A4: 00000443
0x200000A8: 00000442
0x200000AC: 81000200
```

**Memory 1**

Address: 0x20000010

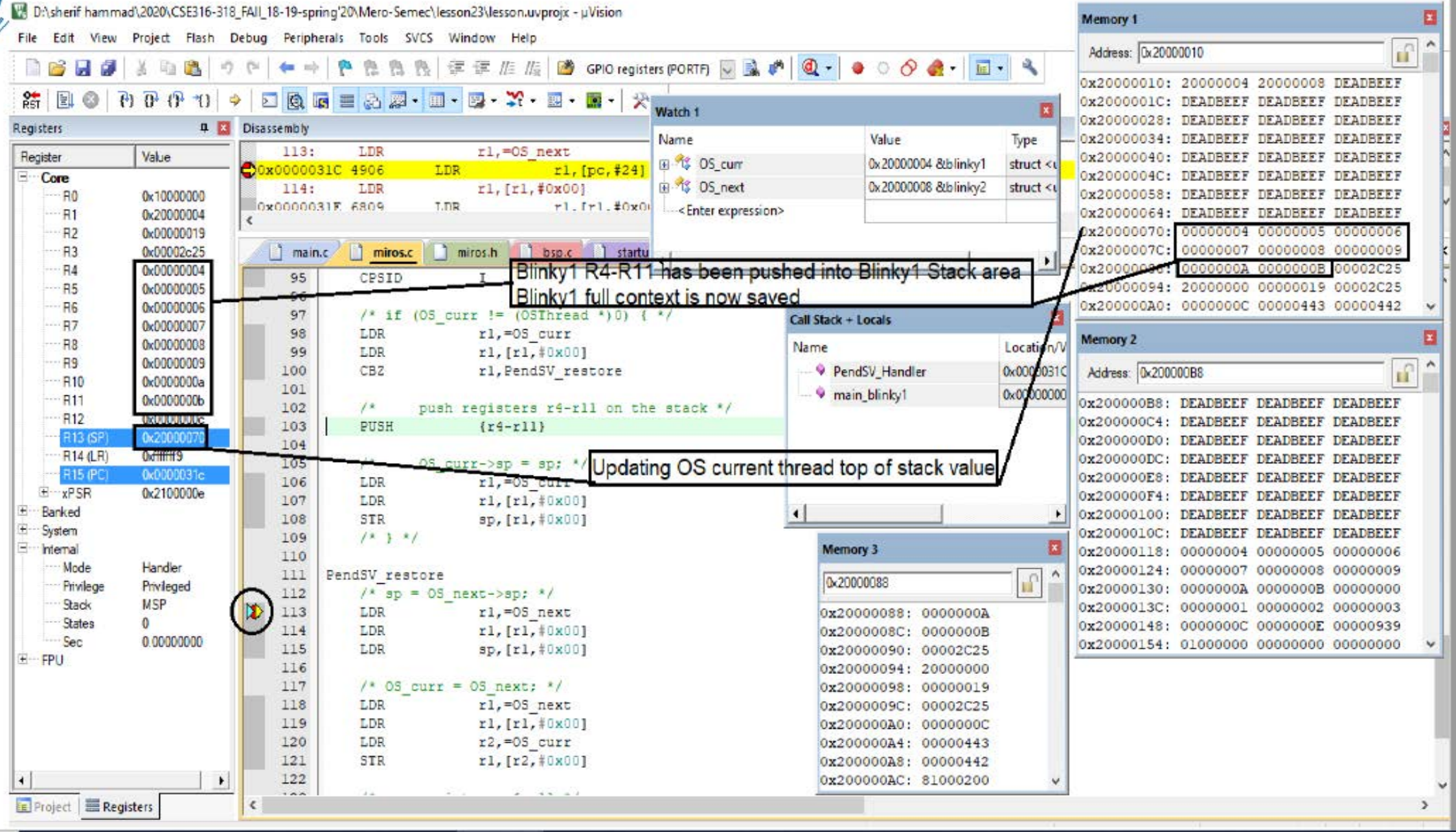
```
0x20000010: 20000004 20000008 DEADBEEF
0x2000001C: DEADBEEF DEADBEEF DEADBEEF
0x20000028: DEADBEEF DEADBEEF DEADBEEF
0x20000034: DEADBEEF DEADBEEF DEADBEEF
0x20000040: DEADBEEF DEADBEEF DEADBEEF
0x2000004C: DEADBEEF DEADBEEF DEADBEEF
0x20000058: DEADBEEF DEADBEEF DEADBEEF
0x20000064: DEADBEEF DEADBEEF DEADBEEF
0x20000070: DEADBEEF DEADBEEF 00000004
0x2000007C: 00000005 00000006 00000007
0x20000088: 00000004 FFFFFFF9 00002C25
0x20000094: 20000000 00000019 00002C25
0x200000A0: 0000000C 00000443 00000442
```

**Memory 2**

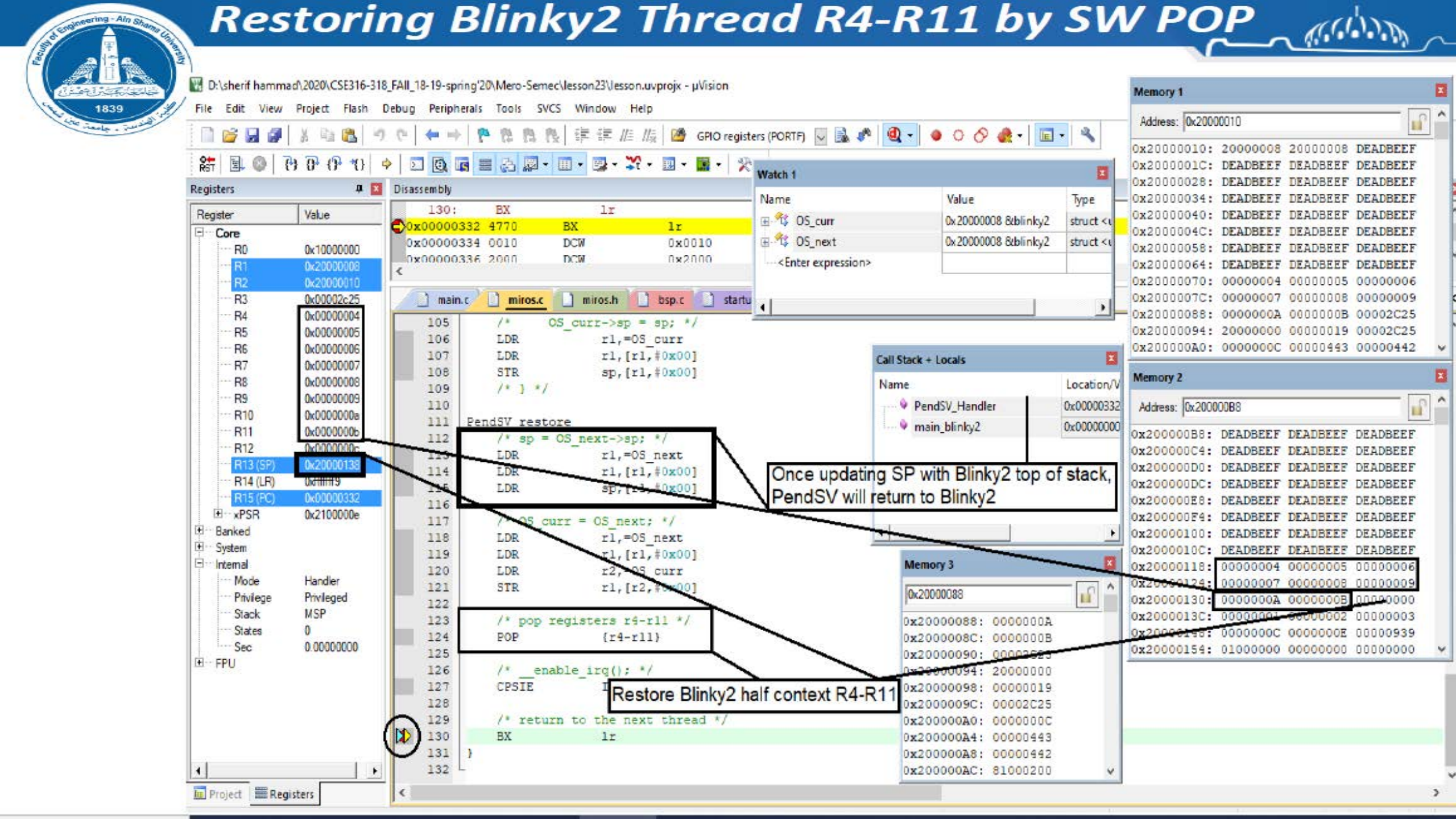
Address: 0x200000B8

```
0x200000B8: DEADBEEF DEADBEEF DEADBEEF
0x200000C4: DEADBEEF DEADBEEF DEADBEEF
0x200000D0: DEADBEEF DEADBEEF DEADBEEF
0x200000DC: DEADBEEF DEADBEEF DEADBEEF
0x200000E8: DEADBEEF DEADBEEF DEADBEEF
0x200000F4: DEADBEEF DEADBEEF DEADBEEF
0x20000100: DEADBEEF DEADBEEF DEADBEEF
0x2000010C: DEADBEEF DEADBEEF DEADBEEF
0x20000118: 00000004 00000005 00000006
0x20000124: 00000007 00000008 00000009
0x20000130: 0000000A 0000000B 00000000
0x2000013C: 00000001 00000002 00000003
0x20000148: 0000000C 0000000E 00000939
0x20000154: 01000000 00000000 00000000
```

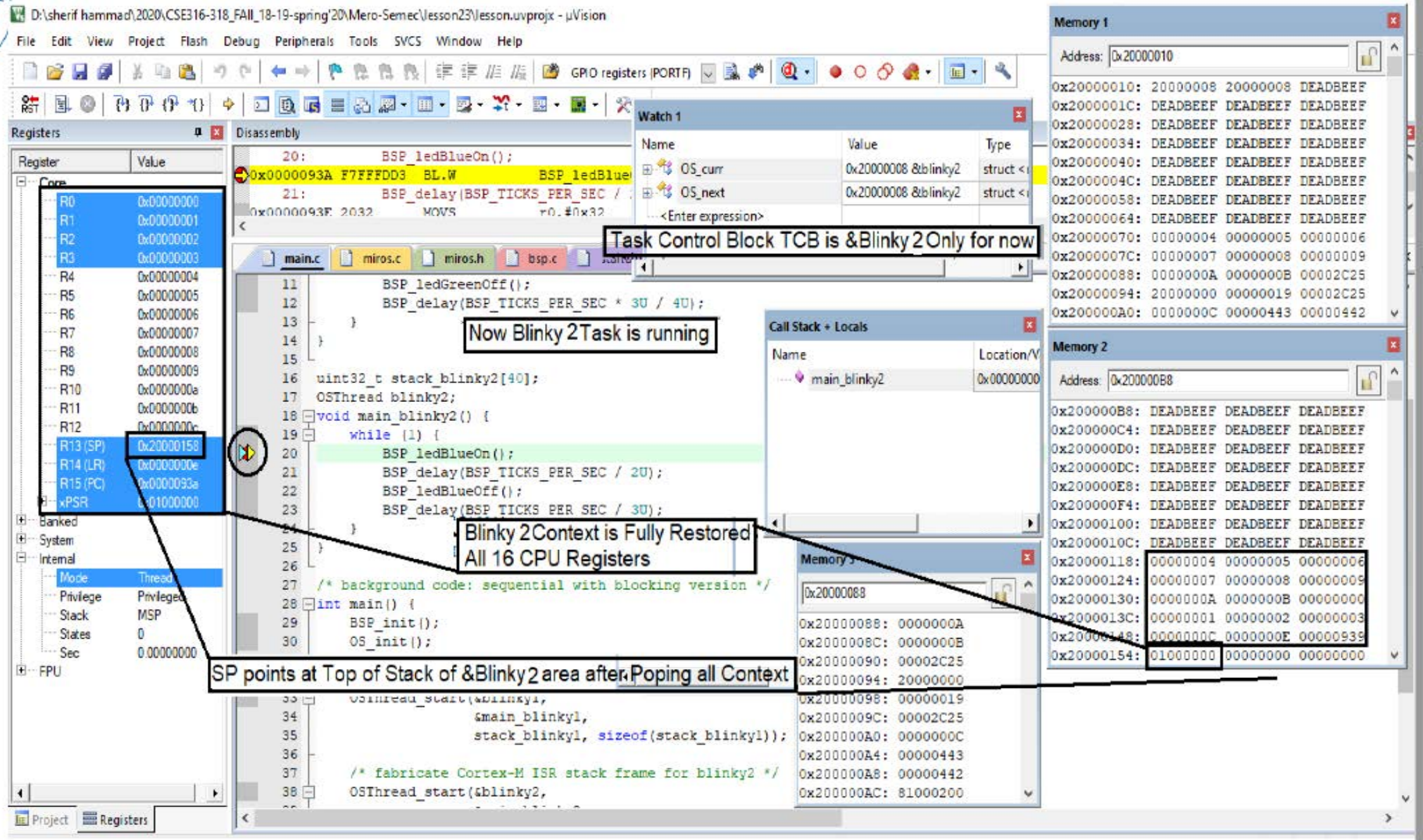
# Beginning to Restore Blinky2 Thread



# Restoring Blinky2 Thread R4-R11 by SW POP



# Complete Restore of Blinky2 Remaining 8 ISR HW Registers




# *Real Time Operating System Introduction & "FreeRTOS"*

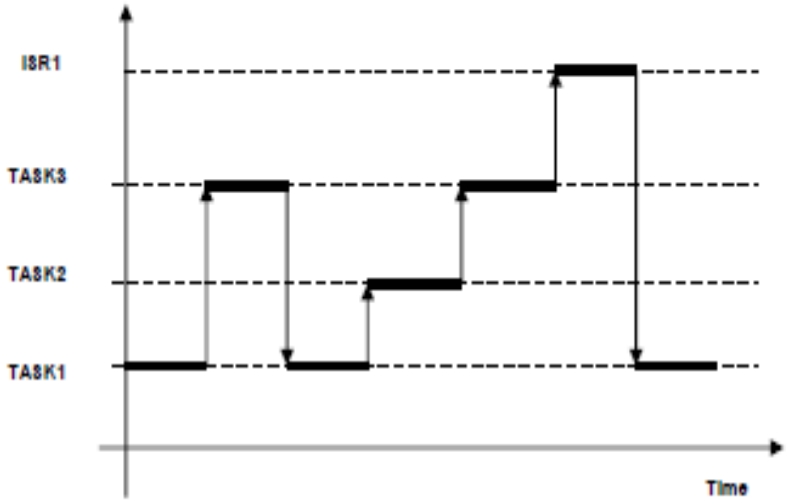
***Sherif Hammad***

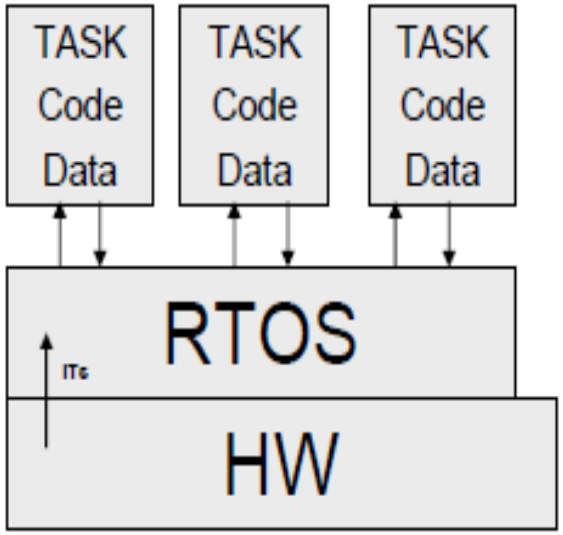

# *Agenda*

- **RTOS Basics**
- **RTOS sample State Machine**
- **RTOS scheduling criteria**
- **RTOS optimization criteria**
- **Soft/Hard Real Time requirements**
- **Tasks scheduling**

# RTOS Basics

- Kernel: schedules tasks
- Tasks: concurrent activity with its own state (PC, registers,stack, etc.)



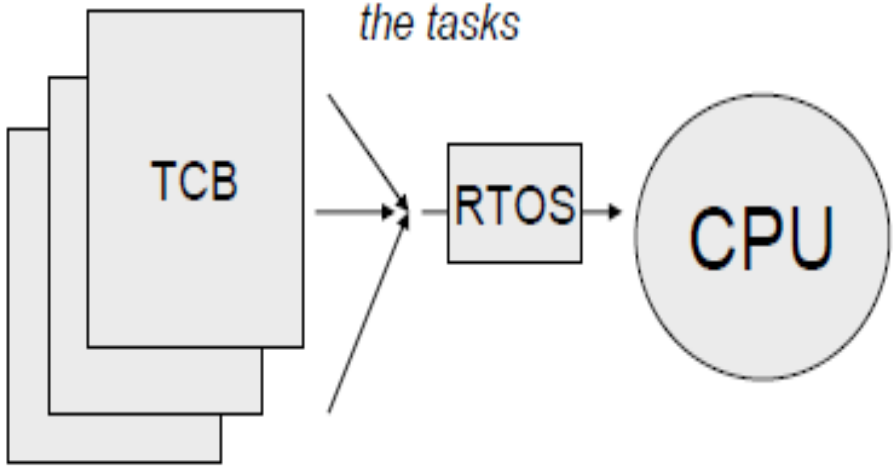



# Tasks

- Tasks = Code + Data + State (context)
- Task State is stored in a Task Control Block (TCB) when the task is not running on the processor
- Typical TCB:

| ID |
| --- |
| Priority |
| Status |
| Registers |
| Saved PC |
| Saved SP |

*The RTOS effectively multiplexes the CPU among the tasks*

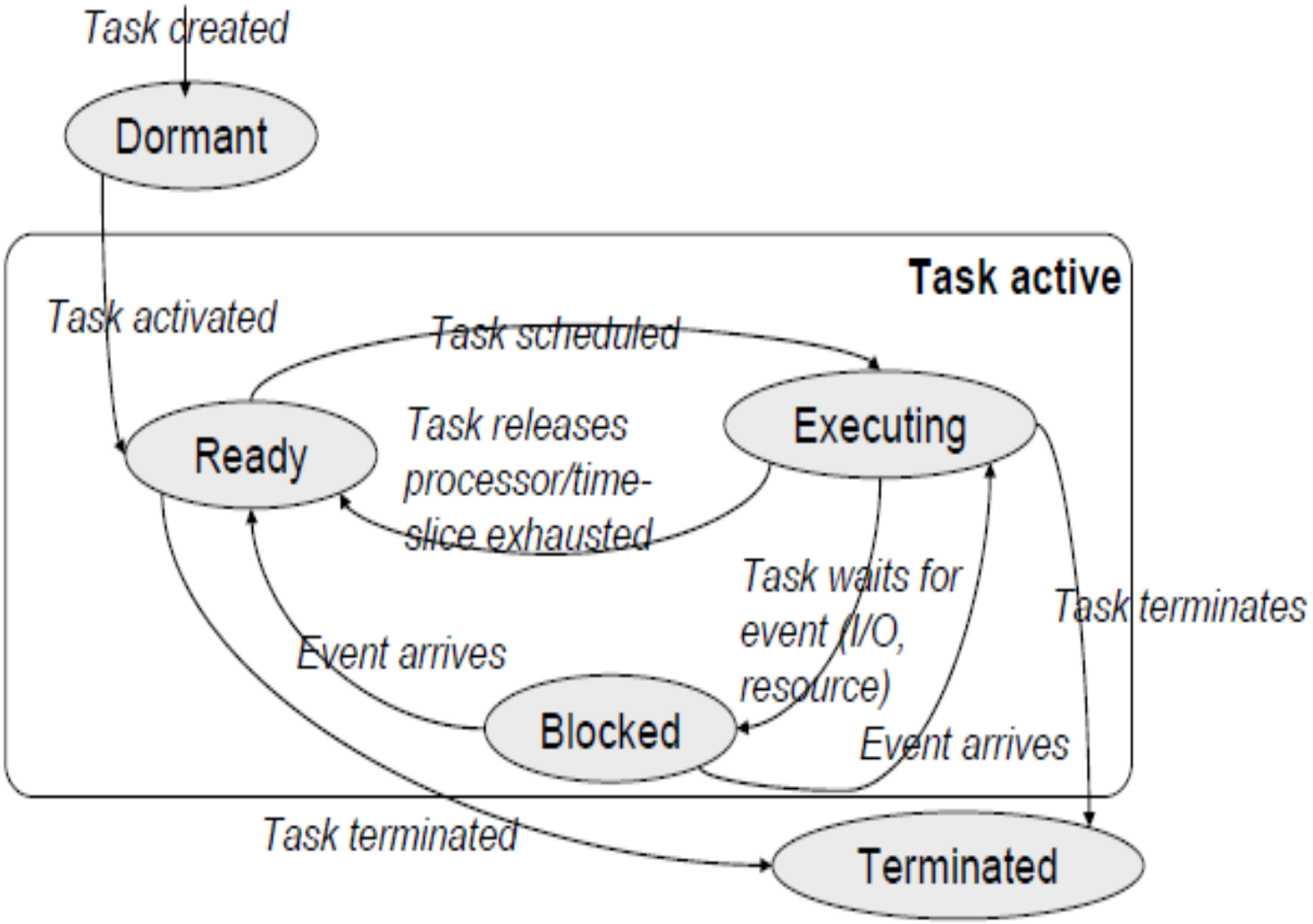


# Task states

- **Executing**: running on the CPU
- **Ready**: could run but another one is using the CPU
- **Blocked**: waits for something (I/O, signal, resource, etc.)
- **Dormant**: created but not executing yet
- **Terminated**: no longer active

The RTOS implements a Finite State Machine for each task, and manages its transitions.

# Task State Transitions



# RTOS Scheduler

- Implements task state machine
- Switches between tasks
- Context switch algorithm:
  1. Save current context into current TCB
  2. Find new TCB
  3. Restore context from new TCB
  4. Continue
- Switch between EXECUTING -> READY:
  1. Task yields processor voluntarily: **NON-PREEMPTIVE**
  2. RTOS switches because of a higher-priority task/event: **PREEMPTIVE**

# *CPU Scheduling Criteria*

- **CPU Utilization:** **CPU should be as busy as possible (40% to 90%)**
- **Throughput:** **No. of processes per unit time**
- **Turnaround time:** **For a particular process how long it takes to execute. (Interval between time of submission to completion)**
- **Waiting time:** **Total time process spends in ready queue.**
- **Response:** **First response of process after submission**

## *Optimization criteria*

- **It is desirable to**
  - **Maximize CPU utilization**
  - **Maximize throughput**
  - **Minimize turnaround time**
  - **Minimize start time**
  - **Minimize waiting time**
  - **Minimize response time**
- **In most cases, we strive to optimize the average measure of each metric**
- **In other cases, it is more important to optimize the minimum or maximum values rather than the average**

# *Soft/Hard Real Time*

- Soft real-time requirements: state a time deadline—but breaching the deadline would not render the system useless.
- Hard real-time requirements: state a time deadline—and breaching the deadline would result in absolute failure of the system.
- Cortex-M4 has only one core executing a single Thread at a time.
- The kernel decides which thread should be executing by examining the priority assigned to each thread by the application designer.
- Application designer could assign higher priorities to hard-real-time-threads and lower priorities to soft real-time

# Why Use a Real-time Kernel?

- Abstracting away timing information
- Maintainability/Reusability/Extensibility
- Modularity
- Team development
- Improved efficiency (No Polling)
- Idle time:

  The Idle task is created automatically when the kernel is started. It executes whenever there are no application tasks wishing to execute. The idle task can be used to measure spare processing capacity, to perform background checks, or simply to place the processor into a low-power mode.

- Flexible interrupt handling:

  Interrupt handlers can be kept very short by deferring most of the required processing to handler RTOS tasks.

# *Task Functions*

- **Arbitrary naming: Must return void: Must take a void pointer parameter:**

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

- **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.**

- **A single task function definition can be used to create any number of tasks—each created task being a separate execution instance with its own stack and its own copy of any automatic (stack) variables defined within the task itself.**

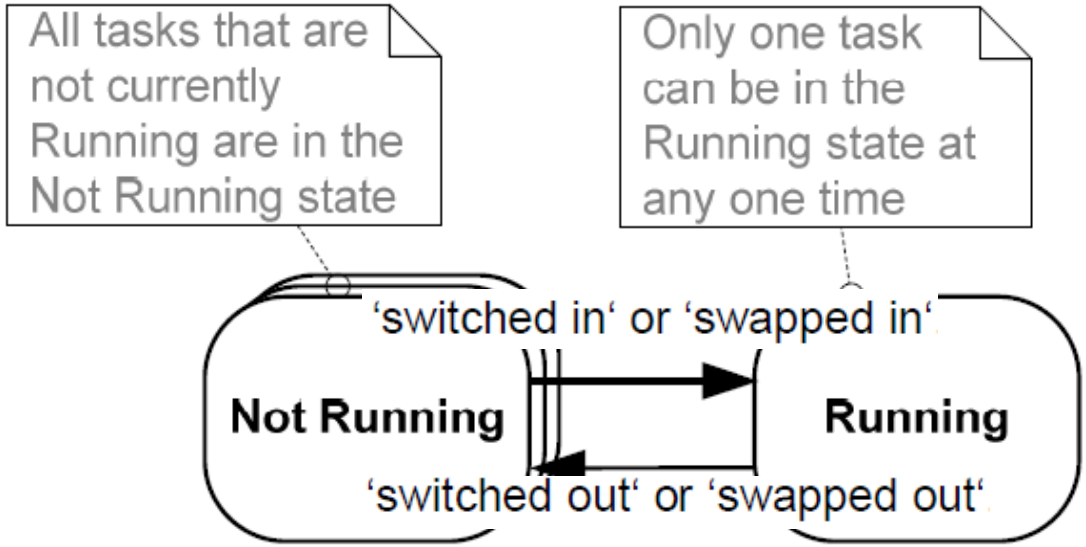
# Task Functions

```
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

- When a task is in the Running state, the processor is executing its code.
- When a task is in the Not Running state, 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.



# Creating Tasks
## The xTaskCreate() API 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 main() will never reach here as the scheduler will
    now be running the tasks.  If main() does reach here then it is likely that
    there was insufficient heap memory available for the idle task to be created.
    Chapter 5 provides more information on memory management. */
    for( ;; );
}
```

# Example 1: Task Functions

```
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. */
        }
    }
}
```

# Run Example 1


**Figure 2. The output produced when Example 1 is executed**

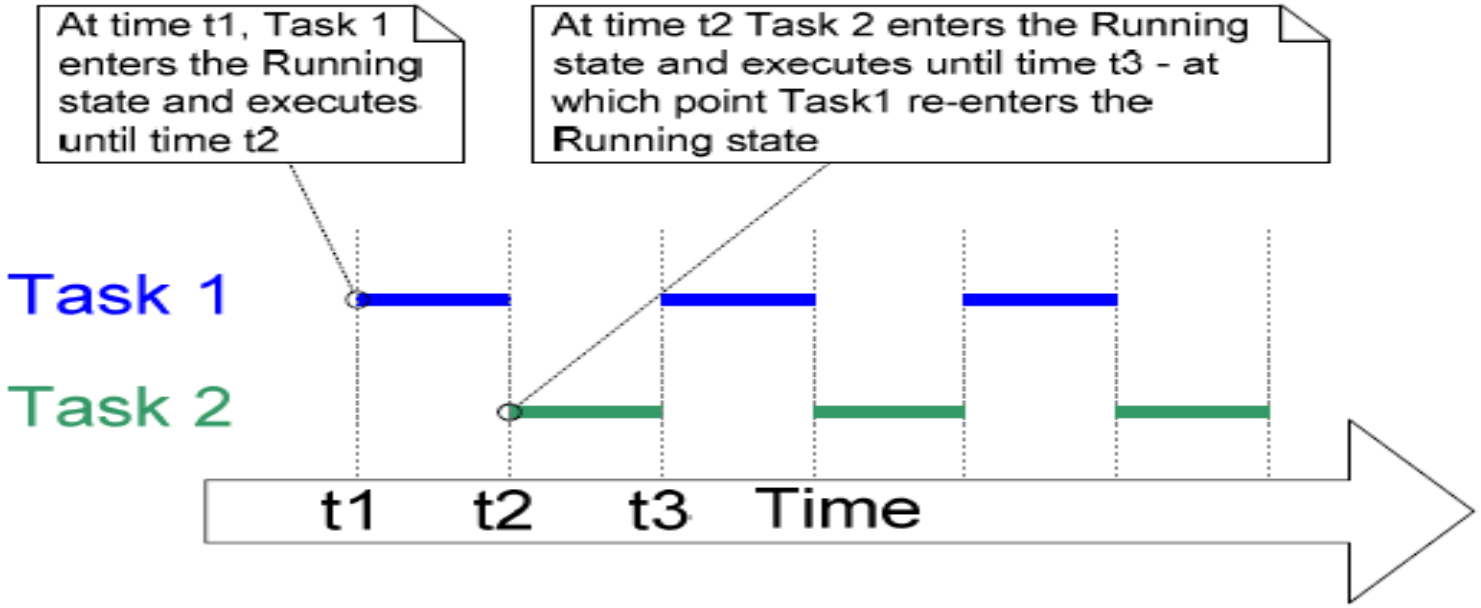



**Figure 3. The execution pattern of the two Example 1 tasks**

# Tick Interrupt

- 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.

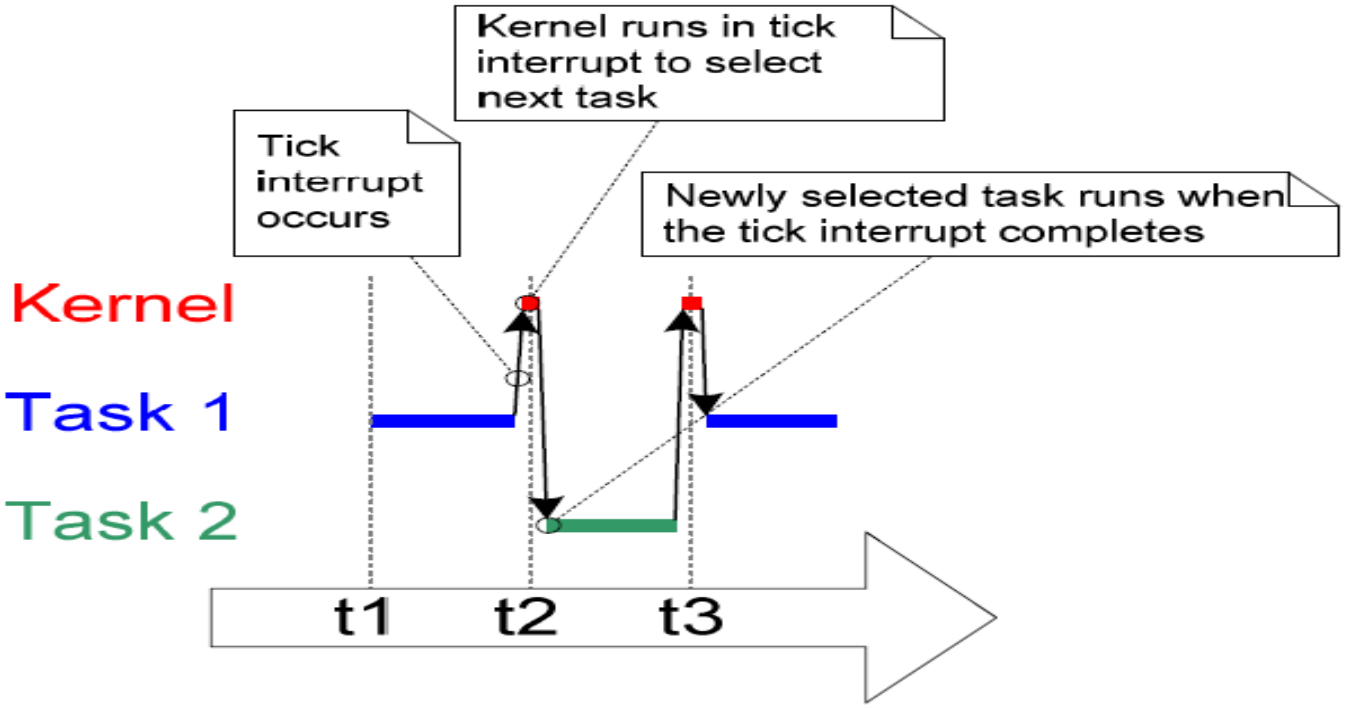



Figure 4. The execution sequence expanded to show the tick interrupt executing


# *Real Time Operating System Introduction & "FreeRTOS"*

***Sherif Hammad***

# *Agenda*

- **Tasks priorities**
- **Task State Machine**
- **Periodic tasks**
- **Tasks Blocking**

# Example 1: Task Functions (Cont.)

```
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. */
        }
    }
}
```

# Run Example 1


**Figure 2. The output produced when Example 1 is executed**

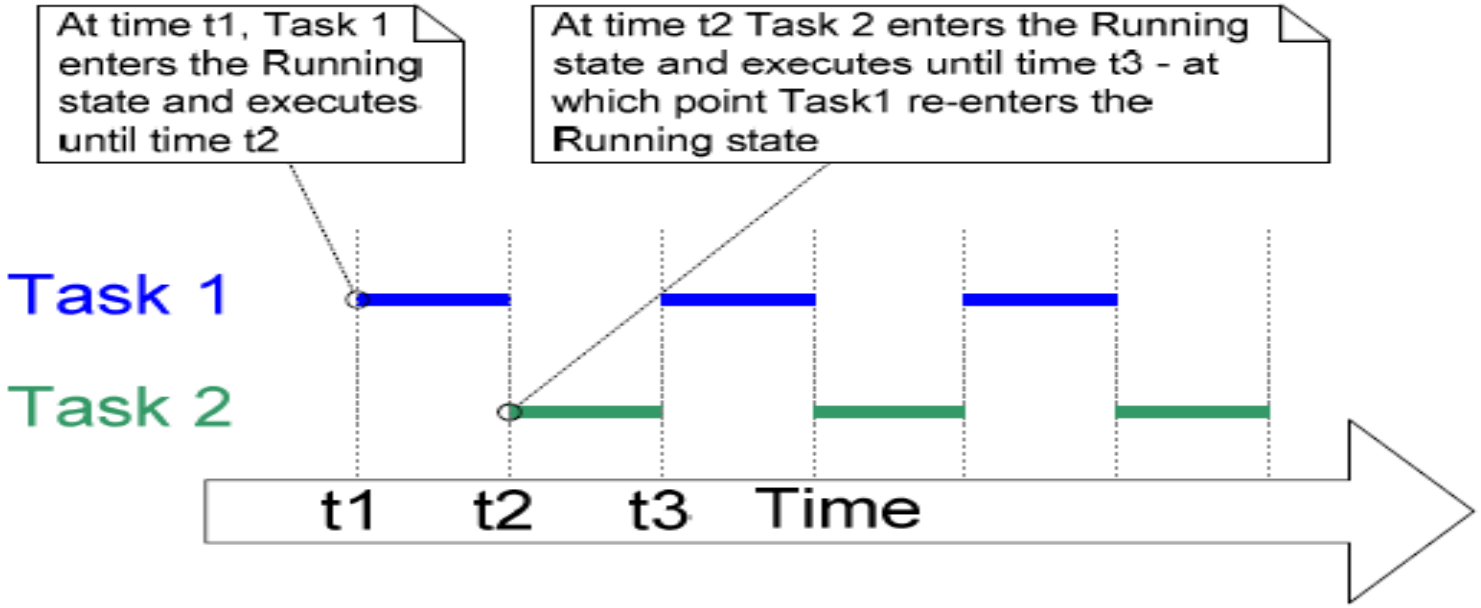



**Figure 3. The execution pattern of the two Example 1 tasks**

# Tick Interrupt

- 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.

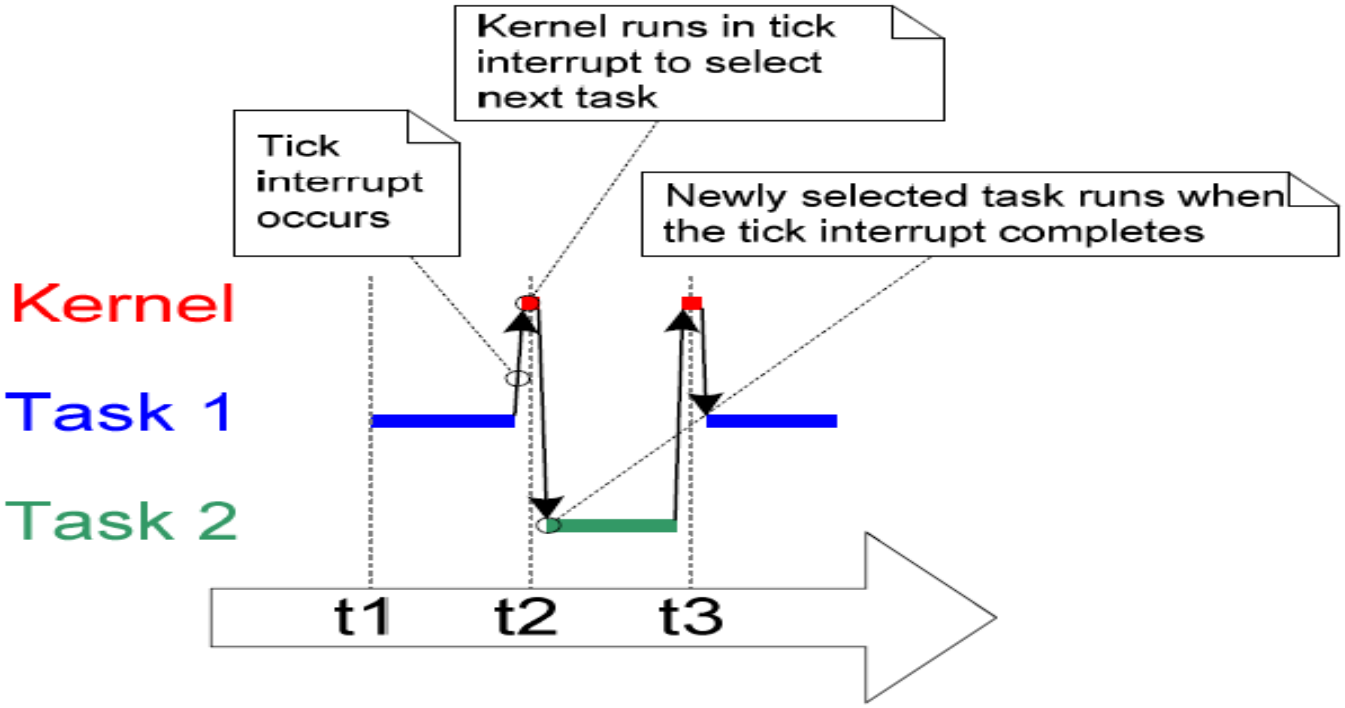



Figure 4. The execution sequence expanded to show the tick interrupt executing

# Task Creation After Schedule Started
## (From Within Another Task)

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

    /* If this task code is executing then the scheduler must already have
    been started.  Create the other task before we enter the infinite loop. */
    xTaskCreate( vTask2, "Task 2", 240, NULL, 1, NULL );

    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. */
        }
    }
}
```

# Example 2: Single Task Function "Instantiated Twice" (Two Task Instants)

```
/* 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 one of the two tasks. */
    xTaskCreate(    vTaskFunction,          /* 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 */
                    (void*)pcTextForTask1,  /* Pass the text to be printed into the
                                               task using the task parameter. */
                    1,                      /* This task will run at priority 1. */
                    NULL );                 /* We are not using the task handle. */

    /* Create the other task in exactly the same way.  Note this time that multiple
    tasks are being created from the SAME task implementation (vTaskFunction).  Only
    the value passed in the parameter is different.  Two instances of the same
    task are being created. */
    xTaskCreate( vTaskFunction, "Task 2", 240, (void*)pcTextForTask2, 1, NULL );

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

    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks.  If main() does reach here then it is likely that
    there was insufficient heap memory available for the idle task to be created.
    Chapter 5 provides more information on memory management. */
    for( ;; );
}
```


# Example 2: Single Task Function
## "Instantiated Twice" (Two Task Instants)

```
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 Priorities

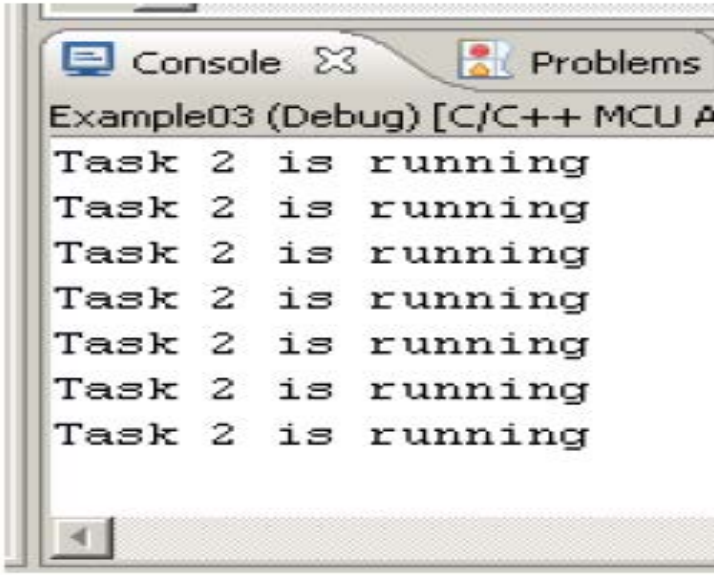
- The priority can be changed after the scheduler has been started by using the vTaskPrioritySet() API function.
- The maximum number of priorities available is set by the application-defined configMAX_PRIORITIES compile time configuration constant within FreeRTOSConfig.h.
- The higher the configMAX_PRIORITIES value the more RAM the kernel will consume, “keep it minimum”.
- Low numeric priority values denote low-priority tasks, with priority 0 being the lowest priority possible.
- 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.
- 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.

# Task Priorities; Example 3

```
/* 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( ;; );
}
```

# Task Priorities: Example 3; Starvation

- Both Tasks are made periodic by the "dummy" loop
- Both Tasks only needs CPU for short execution time!
- Task 2 (High Priority) takes CPU all the time
- Task 1 suffers starvation
- Wastes power and cycles!
- Is there another smarter way?



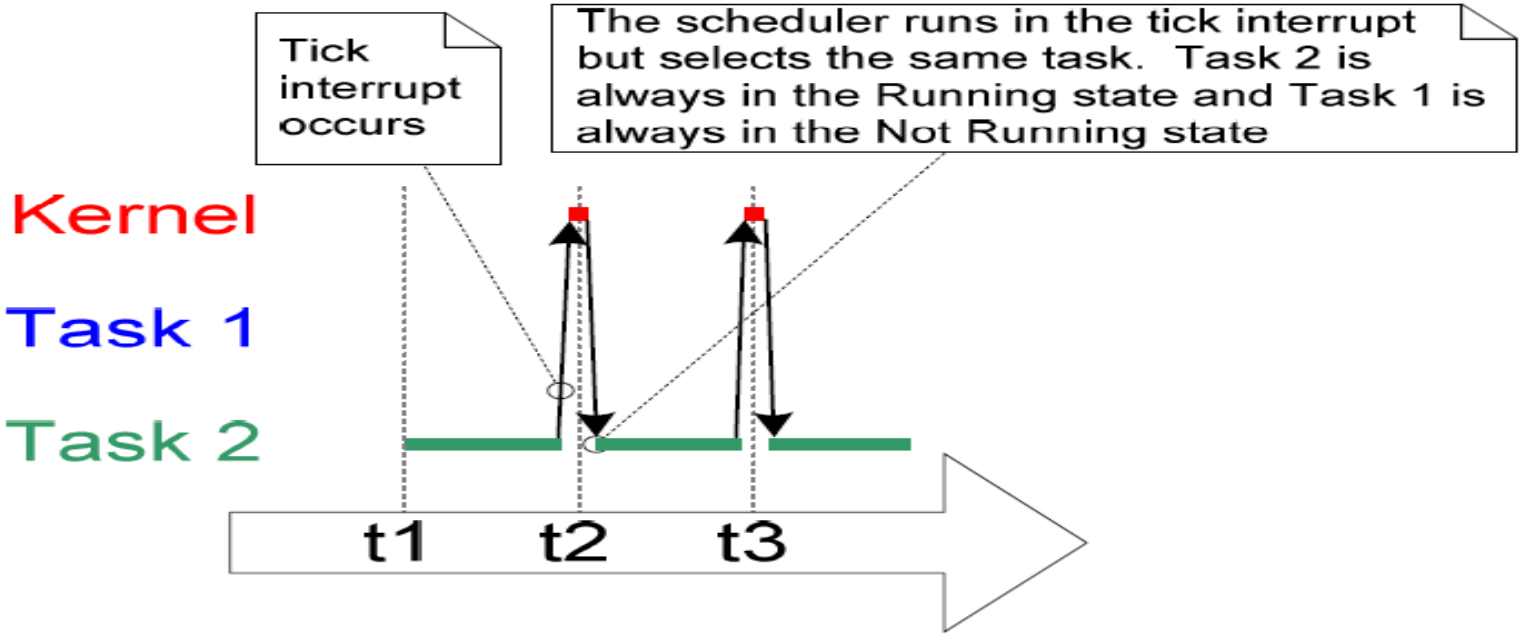



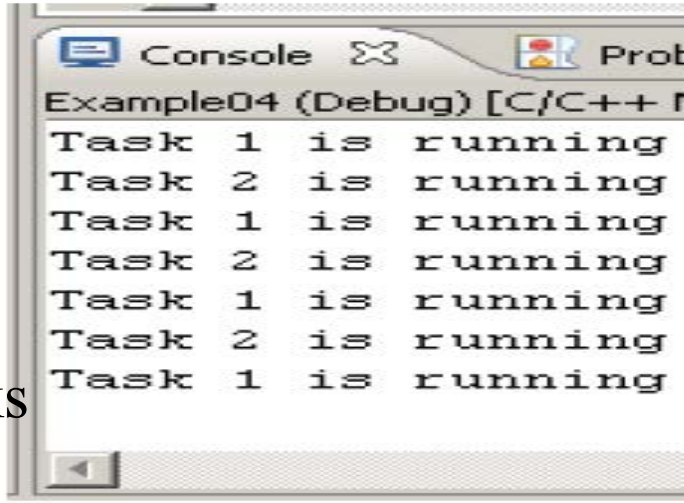

Figure 6. The execution pattern when one task has a higher priority than the other

# Using the Blocked state to create a delay

- 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
- vTaskDelay() API function is available only when. INCLUDE_vTaskDelay is set to 1 in FreeRTOSConfig.h
- The constant portTICK_RATE_MS can be used to convert milliseconds into ticks.
- Portmacro.h: #define portTICK_RATE_MS ( ( portTickType ) 1000 / configTICK_RATE_HZ )
- FreeRTOSConfig.h: #define configTICK_RATE_HZ ( ( portTickType ) 1000 )

```
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 );
    }
```

# Task Blocking: Example 4




- Each time the tasks leave the Blocked state they execute for a fraction of a tick period before re-entering the Blocked state.
- Most of the time there are no application tasks that are able to run; The idle task will run.
- Idle task time is a measure of the spare processing capacity in the system.

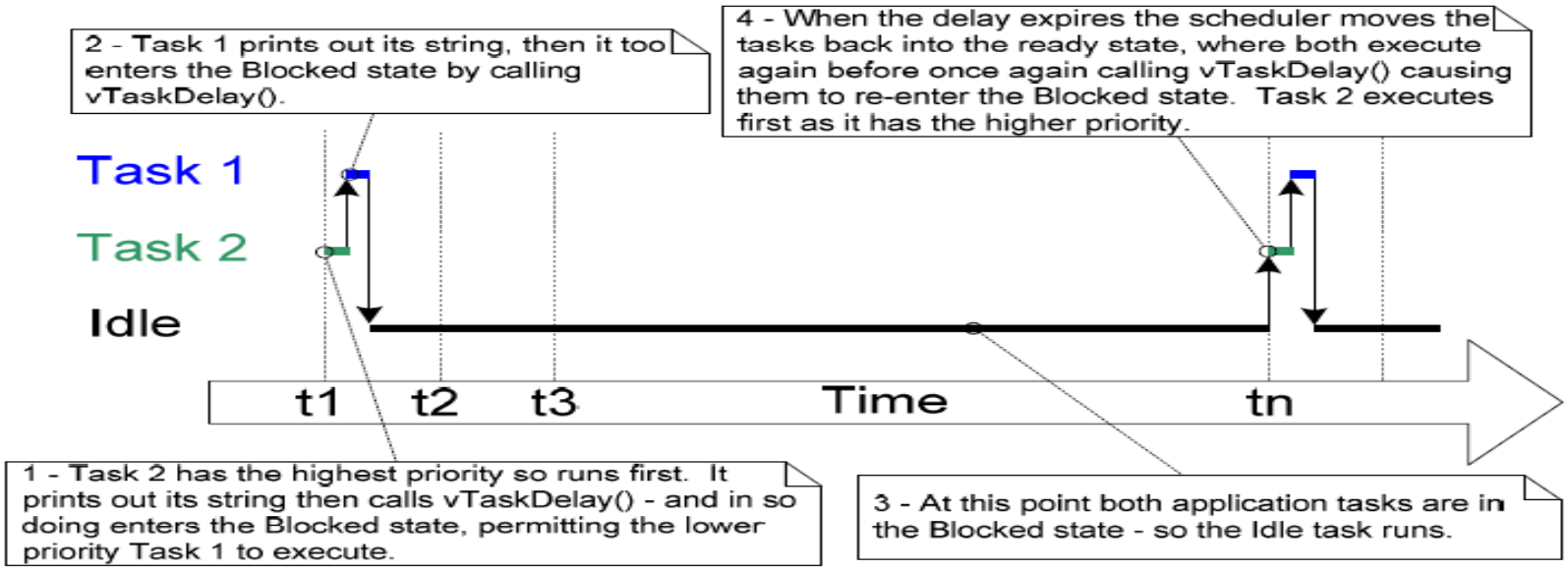
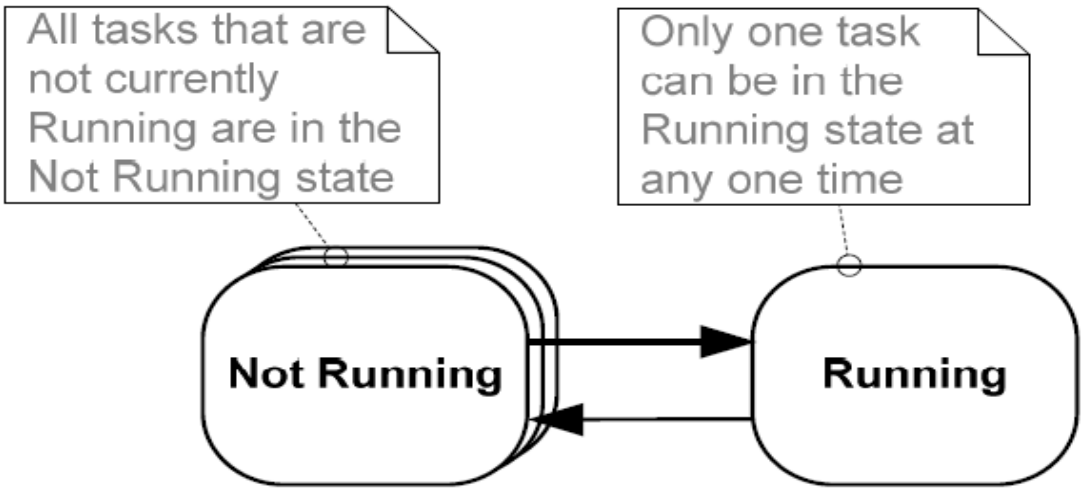



Figure 9. The execution sequence when the tasks use vTaskDelay() in place of the NULL loop

# Expanding the 'Not Running' State




## The Blocked State

- A task that is waiting for an event is said to be in the 'Blocked' state, which is a sub-state of the Not Running state.
- Tasks can enter the Blocked state to wait for two different types of event:
  - Temporal (time-related) events—the event being either a delay period expiring, or an absolute time being reached. For example, a task may enter the Blocked state to wait for 10 milliseconds to pass.
  - Synchronization events—where the events originate from another task or interrupt. For example, a task may enter the Blocked state to wait for data to arrive on a queue. Synchronization events cover a broad range of event types.

# Expanding the 'Not Running' State

## The Suspended State

- 'Suspended' is also a sub-state of Not Running.
- Tasks in the Suspended state are not available to the scheduler.
- The only way into the Suspended state is through a call to the vTaskSuspend() API function
- the only way out being through a call to the vTaskResume() or xTaskResumeFromISR() API functions.
- Most applications do not use the Suspended state.

## The Ready State

- Tasks that are in the Not Running state but are not Blocked or Suspended are said to be in the Ready state.
- They are able to run, and therefore 'ready' to run, but their priorities are not qualifying to be in the Running state.

# Task Blocking: Example 4

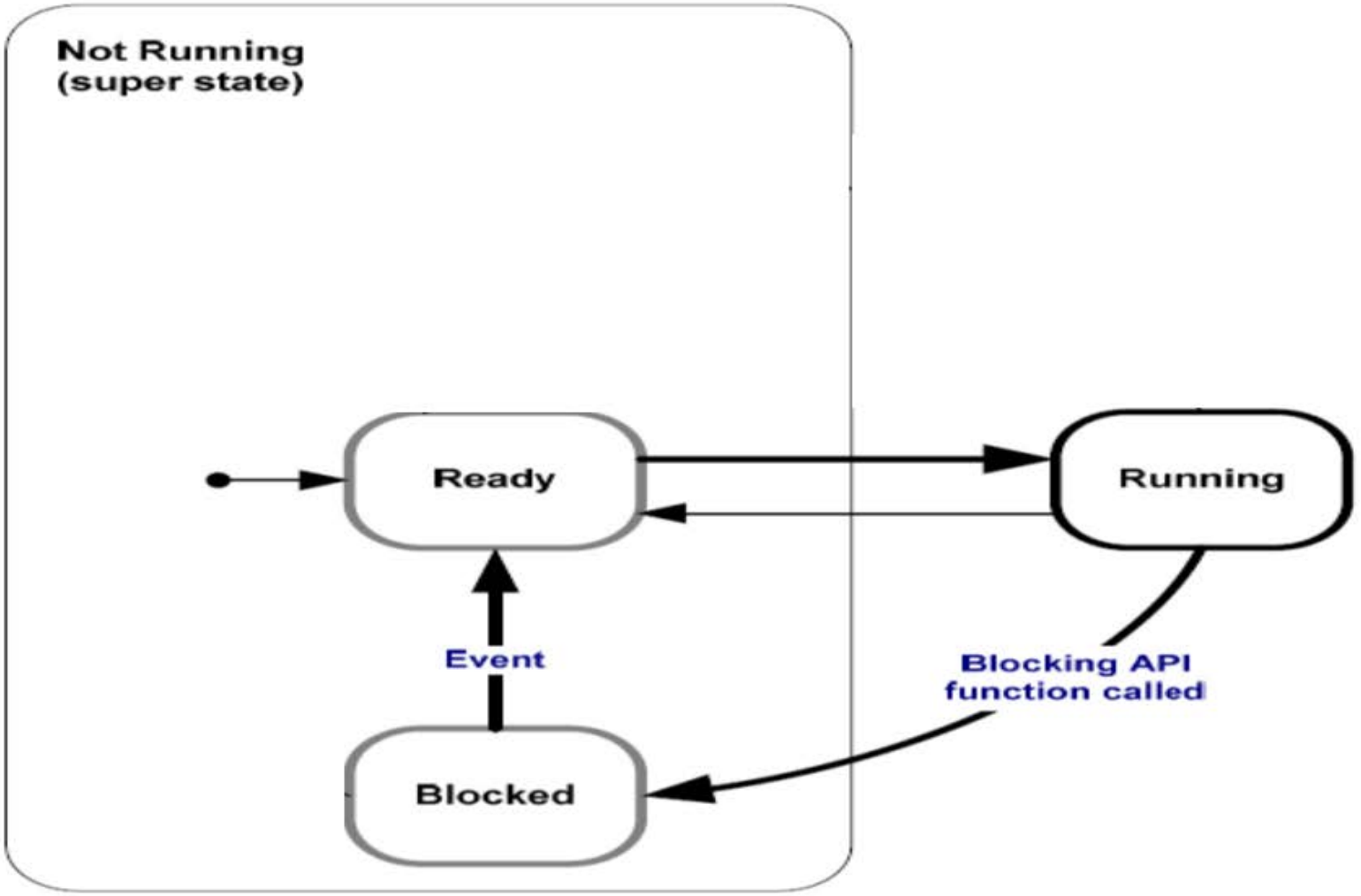



Figure 10. Bold lines indicate the state transitions performed by the tasks in Example 4

# FreeRTOS Task SM

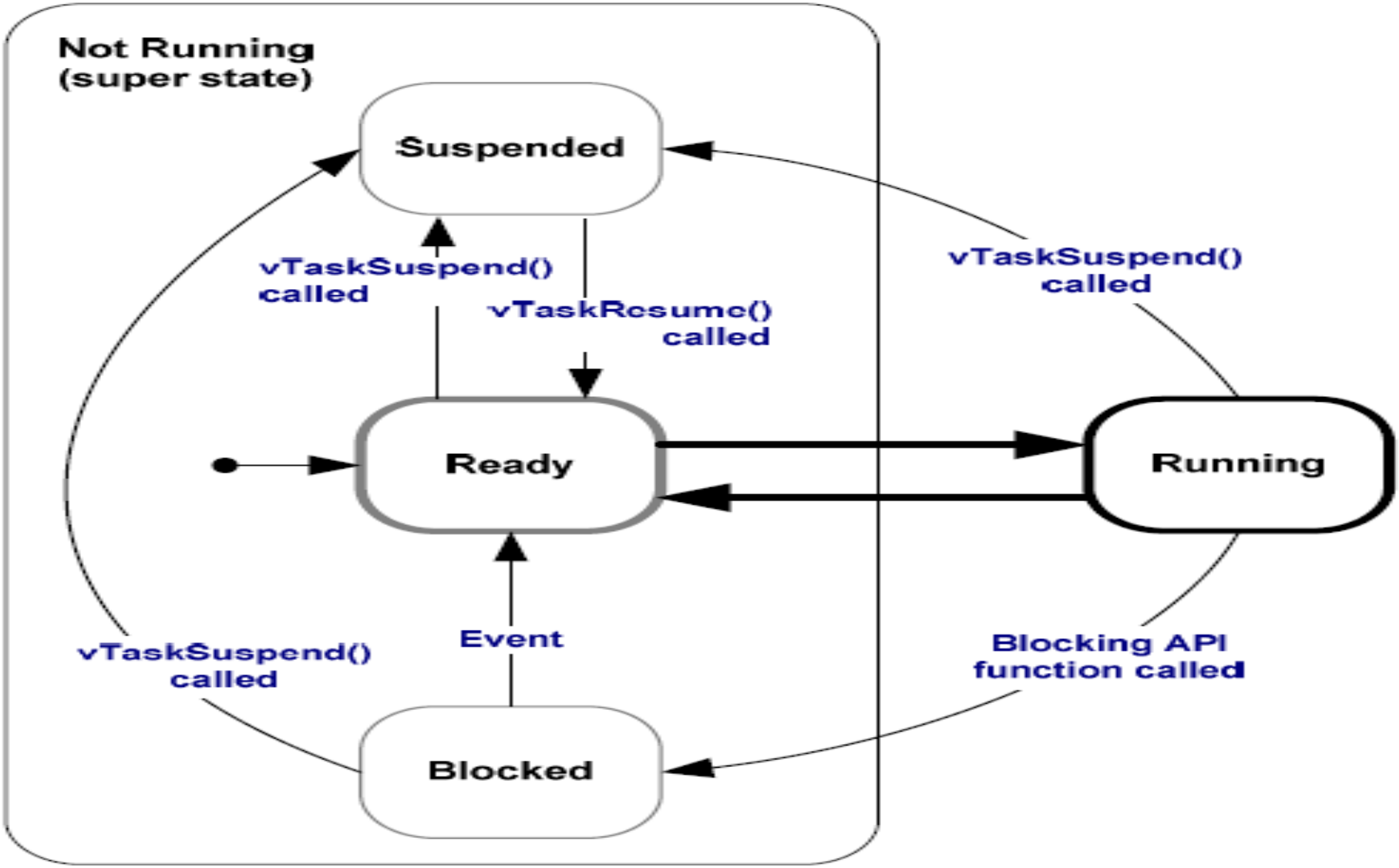



Figure 7. Full task state machine


# Real Time Operating System
# Task Priorities & Deletion

*Sherif Hammad*

# *Agenda*

- **Accurate task periods**
- **Continuous & Periodic Tasks**
- **Changing Tasks priorities**
- **Deleting tasks**

# Accurate Tasks Periods

- vTaskDelayUntil() is similar to vTaskDelay() “but”
- vTaskDelay() parameter specifies the number of tick interrupts that should occur between a task calling vTaskDelay() and the same task once again transitioning out of the Blocked state.
- The length of time the task remains in the blocked state is specified by the vTaskDelay() parameter, but the actual time at which the task leaves the blocked state is relative to the time at which vTaskDelay() was called.
- The parameters to vTaskDelayUntil() specify, instead, the exact tick count value at which the calling task should be moved from the Blocked state into the Ready state.
- vTaskDelayUntil() is the API function that should be used when a fixed execution period is required (where you want your task to execute periodically with a fixed frequency), as the time at which the calling task is unblocked is absolute, rather than relative to when the function was called (as is the case with vTaskDelay()).
- vTaskDelayUntil() API function is available only when INCLUDE_vTaskDelayUntil is set to 1 in FreeRTOSConfig.h.

# Accurate Tasks Periods (Example 5)

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

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

    /* The xLastWakeTime variable needs to be initialized with the current tick
    count.  Note that this is the only time the variable is written to explicitly.
    After this xLastWakeTime is updated automatically internally within
    vTaskDelayUntil(). */
    xLastWakeTime = xTaskGetTickCount();

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

        /* This task should execute exactly every 250 milliseconds.  As per
        the vTaskDelay() function, time is measured in ticks, and the
        portTICK_RATE_MS constant is used to convert milliseconds into ticks.
        xLastWakeTime is automatically updated within vTaskDelayUntil() so is not
        explicitly updated by the task. */
        vTaskDelayUntil( &xLastWakeTime, ( 250 / portTICK_RATE_MS ) );
    }
}
```

# Example 6: Continuous & Periodic Tasks

```
void vContinuousProcessingTask( 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.  This task just does this repeatedly
        without ever blocking or delaying. */
        vPrintString( pcTaskName );
    }
}
```

---

```
void vPeriodicTask( void *pvParameters )
{
portTickType xLastWakeTime;

    /* The xLastWakeTime variable needs to be initialized with the current tick
    count.  Note that this is the only time the variable is explicitly written to.
    After this xLastWakeTime is managed automatically by the vTaskDelayUntil()
    API function. */
    xLastWakeTime = xTaskGetTickCount();

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

        /* The task should execute every 10 milliseconds exactly. */
        vTaskDelayUntil( &xLastWakeTime, ( 10 / portTICK_RATE_MS ) );
    }
}
```

---

Listing 16. The periodic task used in Example 6

# Example 6: Continuous & Periodic Tasks

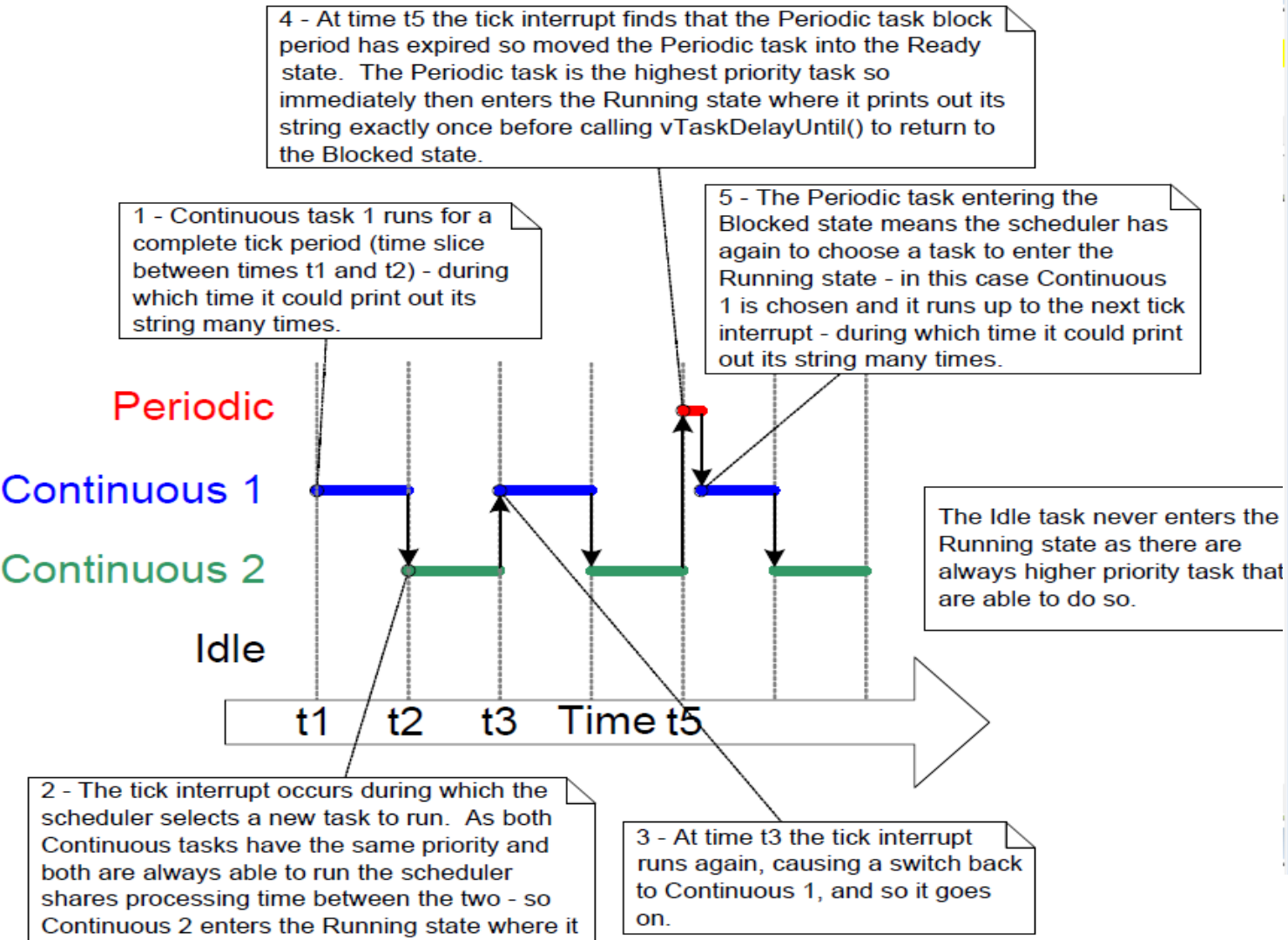



**Figure 12. The execution pattern of Example 6**

Debug (printf) Viewer

```
Periodic task is running...
Continuous task 1 running
Continuous task 1 running
Continuous task 1 running
Continuous task 2 running
Continuous task 2 running
Continuous task 2 running
Continuous task 1 running
Continuous task 1 running
Continuous task 2 running
Continuous task 2 running
Continuous task 1 running
Continuous task 1 running
Continuous task 1 running
Continuous task 2 running
Continuous task 2 running
Continuous task 2 running
Continuous task 1 running
Continuous task 1 running
Continuous task 2 running
Continuous task 2 running
Continuous task 2 running
Continuous task 1 running
Continuous task 1 running
Continuous task 1 running
Continuous task 2 running
Continuous task 2 running
Periodic task is running...
Continuous task 1 running
Continuous task 1 running
```

# Deleting a Task; Example 9

```
int main( void )
{
    /* Create the first task at priority 1.  The task parameter is not used
    so is set to NULL.  The task handle is also not used so likewise is set
    to NULL. */
    xTaskCreate( vTask1, "Task 1", 240, NULL, 1, NULL );
    /* The task is created at priority 1 ______^. */

    /* Start the scheduler so the task starts executing. */
    vTaskStartScheduler();

    /* main() should never reach here as the scheduler has been started. */
    for( ;; );
}
```

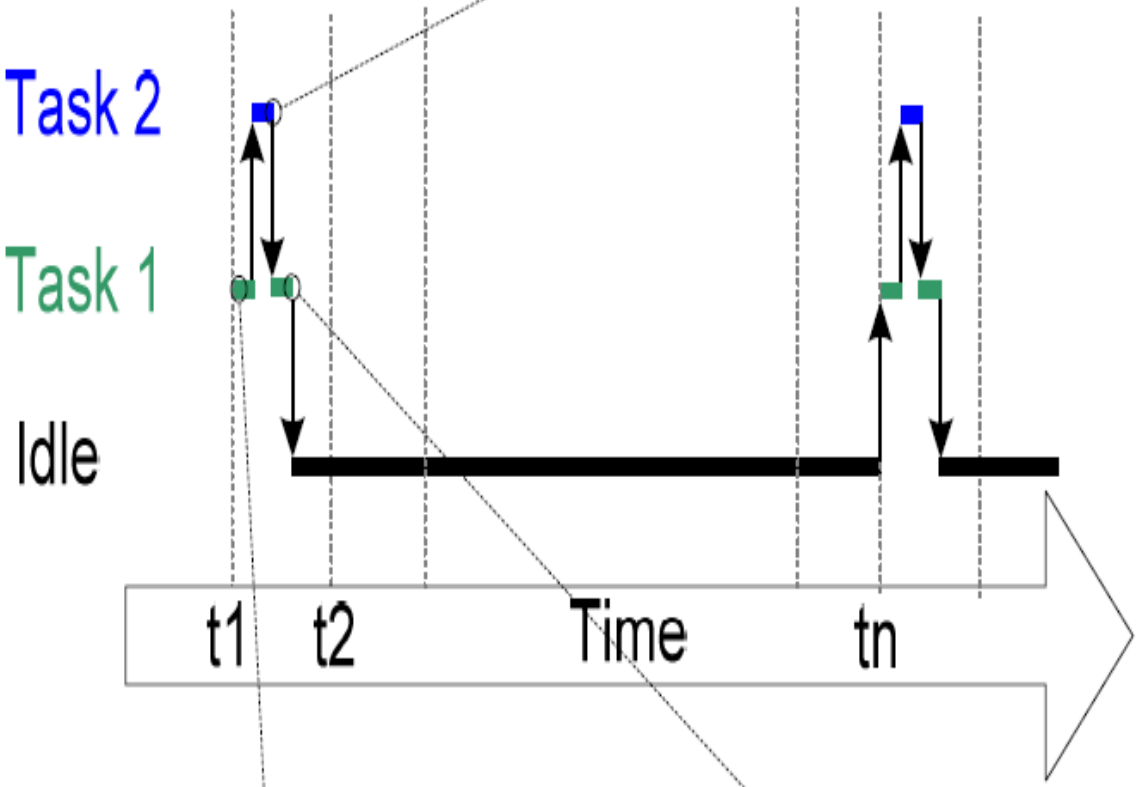
**Listing 26. The implementation of main() for Example 9**

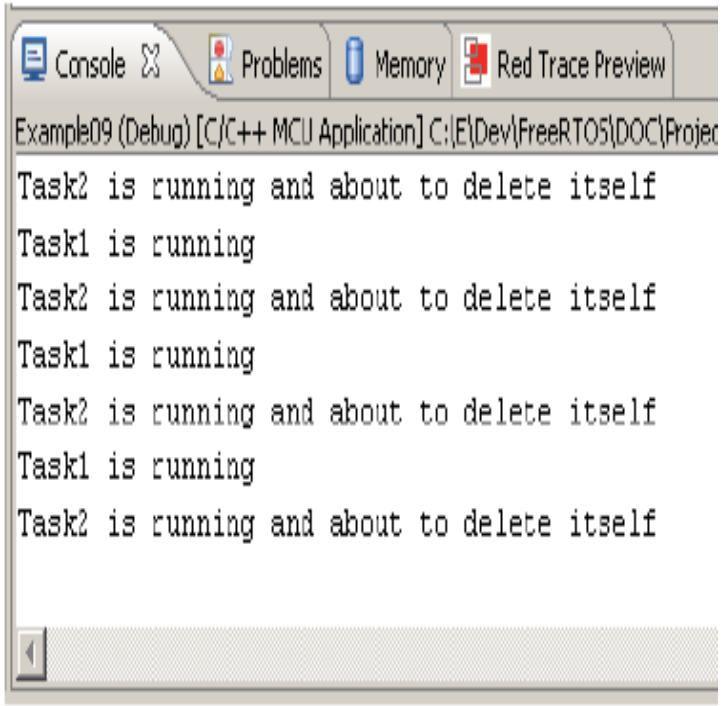
```
void vTask1( void *pvParameters )
{
const portTickType xDelay100ms = 100 / portTICK_RATE_MS;

    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( "Task 1 is running\n" );

        /* Create task 2 at a higher priority.  Again the task parameter is not
        used so is set to NULL - BUT this time the task handle is required so
        the address of xTask2Handle is passed as the last parameter. */
        xTaskCreate( vTask2, "Task 2", 240, NULL, 2, &xTask2Handle );
        /* The task handle is the last parameter _____^^^^^^^^^^^^^^ */

        /* Task 2 has/had the higher priority, so for Task 1 to reach here Task 2
        must have already executed and deleted itself.  Delay for 100
        milliseconds. */
        vTaskDelay( xDelay100ms );
    }
}
```

```
void vTask2( void *pvParameters )
{
    /* Task 2 does nothing but delete itself.  To do this it could call vTaskDelete()
    using NULL as the parameter, but instead and purely for demonstration purposes it
    instead calls vTaskDelete() passing its own task handle. */
    vPrintString( "Task2 is running and about to delete itself\n" );
    vTaskDelete( xTask2Handle );
}
```

Task 2 Preempts 1

2 - Task 2 does nothing other than delete itself, allowing execution to return to Task 1.




1 - Task 1 runs and creates Task 2. Task 2 starts to run immediately as it has the higher priority.

3 - Task 1 calls vTaskDelay(), allowing the idle task to run until the delay time expires, and the whole sequence repeats.

Task 1 Resumes

**Figure 17. The execution sequence for Example 9**

# *Priorities & Preemption*

- **1_three_tasks_same_priority**
- **2_three_tasks_different_priority**
- **3_three_tasks_preemption_delete_task (Example 9)**


# Real Time Operating System
# Idle Task

*Sherif Hammad*

# *Agenda*

- **Idle Task Call Back**
- **Changing Tasks priorities**
- **Scheduling Wrap up**

# *The Idle Task and the Idle Task Hook (Idle Task CallBack)*

- An Idle task is automatically created by the scheduler when vTaskStartScheduler() is called.
- The idle task does very little more than sit in a loop
- The idle task has the lowest possible priority (priority zero), to ensure it never prevents a higher priority application task from entering the Running state
- Idle task is always pre-empted by higher priority tasks
- idle hook (or idle callback) function—a function that is called automatically by the idle task once per iteration of the idle task loop
- Common uses for the Idle task hook include:
  - Executing low priority, background, or continuous processing.
  - Measuring the amount of spare processing capacity.
  - Placing the processor into a low power mode
- configUSE_IDLE_HOOK must be set to 1 within FreeRTOSConfig.h for the idle hook function to get called.

# Example 7: Idle Task CallBack

```
/* Declare a variable that will be incremented by the hook function. */
unsigned long ulIdleCycleCount = 0UL;

/* Idle hook functions MUST be called vApplicationIdleHook(), take no parameters,
and return void. */
void vApplicationIdleHook( void )
{
    /* This hook function does nothing but increment a counter. */
    ulIdleCycleCount++;
}
```

**Listing 18. A very simple Idle hook function**

```
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 AND the number of times ulIdleCycleCount
        has been incremented. */
        vPrintStringAndNumber( pcTaskName, ulIdleCycleCount );

        /* Delay for a period of 250 milliseconds. */
        vTaskDelay( 250 / portTICK_RATE_MS );
    }
}
```

**Listing 19. The source code for the example task prints out the ulIdleCycleCount**

# Example 7: Idle Task CallBack

## How many times IdelTask is called between tasks calls?

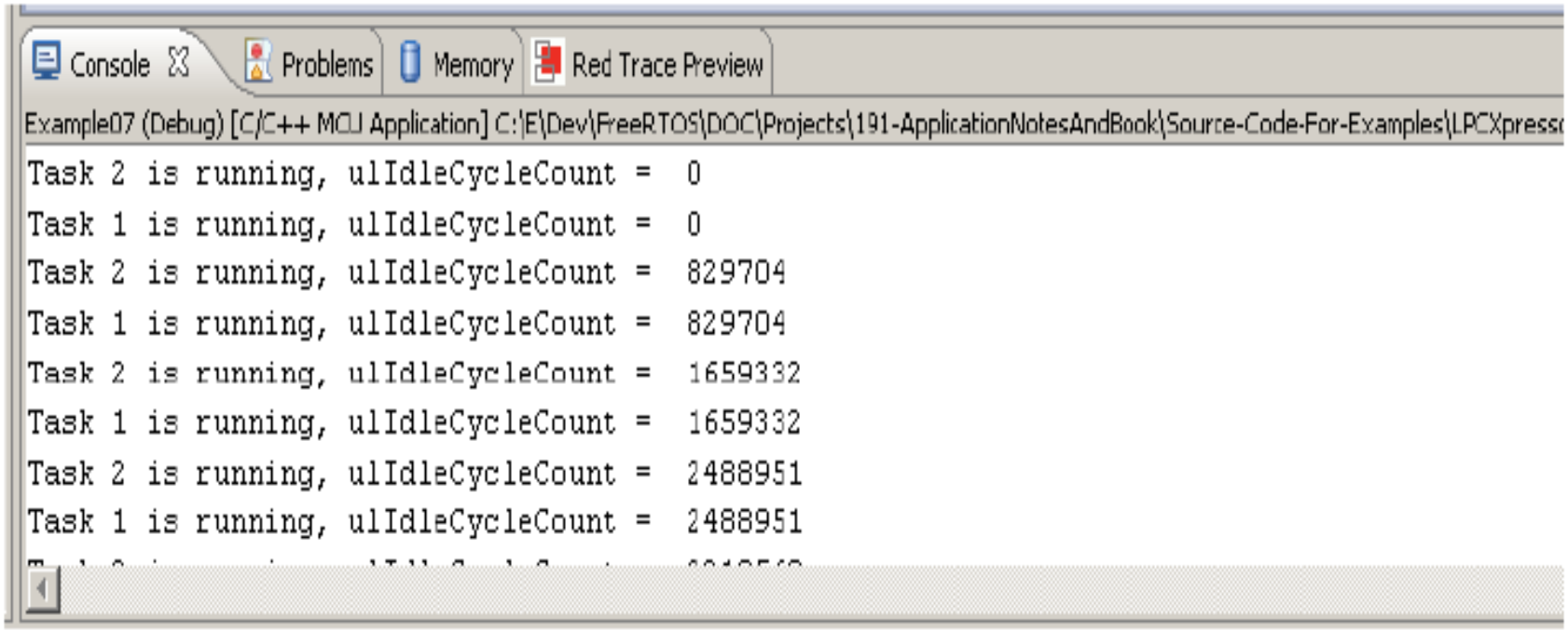



Figure 13. The output produced when Example 7 is executed

# Changing task priorities; Example 8

```
/* Declare a variable that is used to hold the handle of Task 2. */
xTaskHandle xTask2Handle;

int main( void )
{
    /* Create the first task at priority 2.  The task parameter is not used
    and set to NULL.  The task handle is also not used so is also set to NULL. */
    xTaskCreate( vTask1, "Task 1", 240, NULL, 2, NULL );
    /* The task is created at priority 2 _____^. */

    /* Create the second task at priority 1 - which is lower than the priority
    given to Task 1.  Again the task parameter is not used so is set to NULL -
    BUT this time the task handle is required so the address of xTask2Handle
    is passed in the last parameter. */
    xTaskCreate( vTask2, "Task 2", 240, NULL, 1, &xTask2Handle );
    /* The task handle is the last parameter ____^^^^^^^^^^^^^ */

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

    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks.  If main() does reach here then it is likely that
    there was insufficient heap memory available for the idle task to be created.
    Chapter 5 provides more information on memory management. */
    for( ;; );
}
```

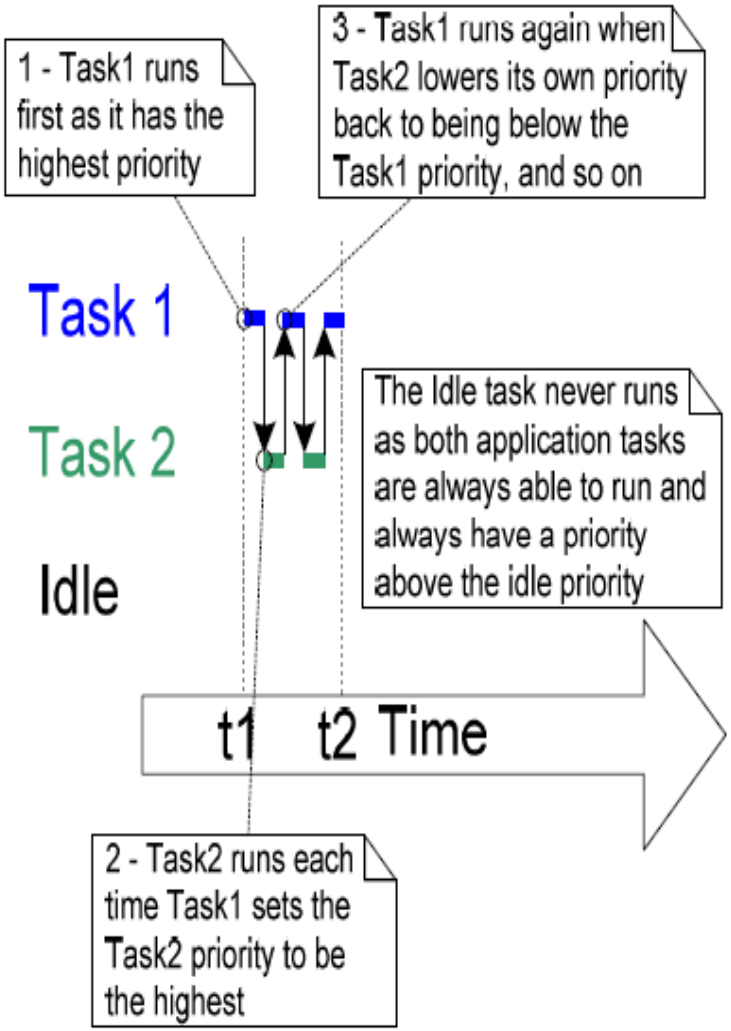
# Changing task priorities; Example 8

```
void vTask1( void *pvParameters )
{
unsigned portBASE_TYPE uxPriority;

    /* This task will always run before Task 2 as it is created with the higher
    priority.  Neither Task 1 nor Task 2 ever block so both will always be in either
    the Running or the Ready state.

    Query the priority at which this task is running - passing in NULL means
    "return my priority". */
    uxPriority = uxTaskPriorityGet( NULL );

    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( "Task 1 is running\n" );

        /* Setting the Task 2 priority above the Task 1 priority will cause
        Task 2 to immediately start running (as then Task 2 will have the higher
        priority of the two created tasks).  Note the use of the handle to task
        2 (xTask2Handle) in the call to vTaskPrioritySet().  Listing 24 shows how
        the handle was obtained. */
        vPrintString( "About to raise the Task 2 priority\n" );
        vTaskPrioritySet( xTask2Handle, ( uxPriority + 1 ) );

        /* Task 1 will only run when it has a priority higher than Task 2.
        Therefore, for this task to reach this point Task 2 must already have
        executed and set its priority back down to below the priority of this
        task. */
    }
}
```

# Changing task priorities; Example 8

```
void vTask2( void *pvParameters )
{
unsigned portBASE_TYPE uxPriority;

    /* Task 1 will always run before this task as Task 1 is created with the
    higher priority.  Neither Task 1 nor Task 2 ever block so will always be
    in either the Running or the Ready state.

    Query the priority at which this task is running - passing in NULL means
    "return my priority". */
    uxPriority = uxTaskPriorityGet( NULL );

    for( ;; )
    {
        /* For this task to reach this point Task 1 must have already run and
        set the priority of this task higher than its own.

        Print out the name of this task. */
        vPrintString( "Task2 is running\n" );

        /* Set our priority back down to its original value.  Passing in NULL
        as the task handle means "change my priority".  Setting the
        priority below that of Task 1 will cause Task 1 to immediately start
        running again - pre-empting this task. */
        vPrintString( "About to lower the Task 2 priority\n" );
        vTaskPrioritySet( NULL, ( uxPriority - 2 ) );
    }
}
```

# Changing task priorities; Example 8



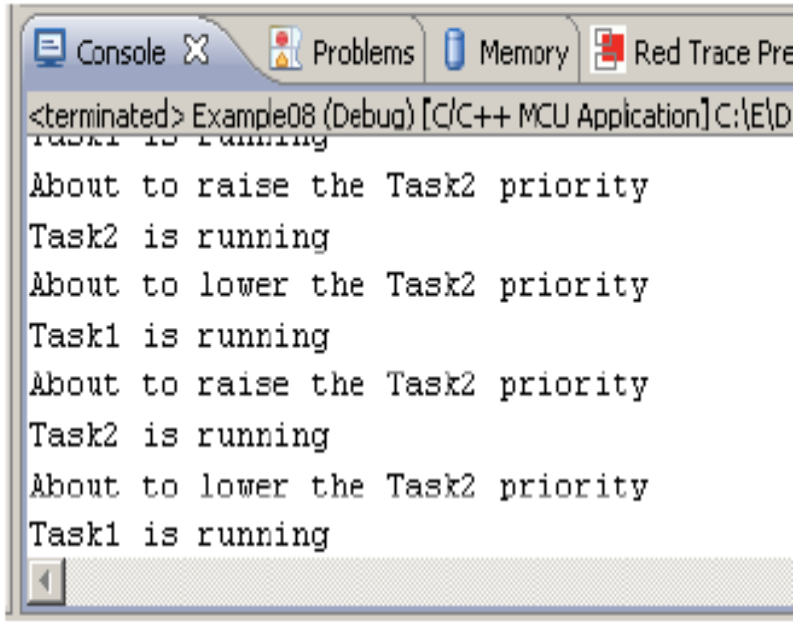




**Figure 14. The sequence of task execution when running Example 8**

## *The Scheduling Algorithm—A Summary*

- **Fixed Prioritized Pre-emptive Scheduling**
  - **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.**

- **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.**

# The Scheduling Algorithm—A Summary

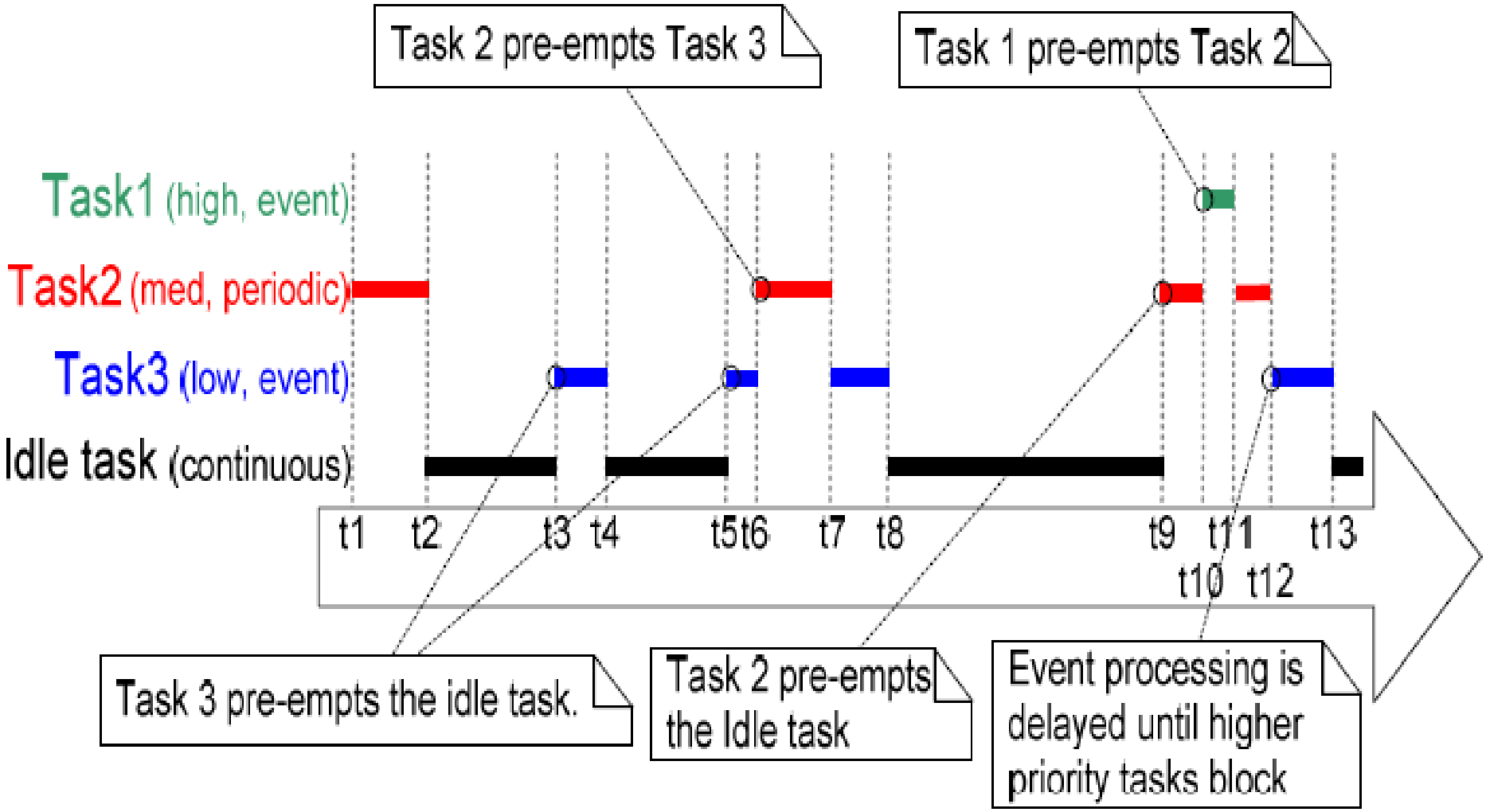



**Figure 18. Execution pattern with pre-emption points highlighted**


# Real Time Operating System
# "FreeRTOS"
# Change Task Priority

***Sherif Hammad***

Using the FreeRTOS Real Time Kernel - a Practical Guide - Cortex M3 Edition (FreeRTOS Tutorial Books)

by Richard Barry

# Changing task priorities; Example 8

```
/* Declare a variable that is used to hold the handle of Task 2. */
xTaskHandle xTask2Handle;

int main( void )
{
    /* Create the first task at priority 2.  The task parameter is not used
    and set to NULL.  The task handle is also not used so is also set to NULL. */
    xTaskCreate( vTask1, "Task 1", 240, NULL, 2, NULL );
    /* The task is created at priority 2 _____^. */

    /* Create the second task at priority 1 - which is lower than the priority
    given to Task 1.  Again the task parameter is not used so is set to NULL -
    BUT this time the task handle is required so the address of xTask2Handle
    is passed in the last parameter. */
    xTaskCreate( vTask2, "Task 2", 240, NULL, 1, &xTask2Handle );
    /* The task handle is the last parameter ____^^^^^^^^^^^^^ */

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

    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks.  If main() does reach here then it is likely that
    there was insufficient heap memory available for the idle task to be created.
    Chapter 5 provides more information on memory management. */
    for( ;; );
}
```

Listing 24. The implementation of main() for Example 8

# Changing task priorities; Example 8

```
void vTask1( void *pvParameters )
{
unsigned portBASE_TYPE uxPriority;

    /* This task will always run before Task 2 as it is created with the higher
    priority.  Neither Task 1 nor Task 2 ever block so both will always be in either
    the Running or the Ready state.

    Query the priority at which this task is running - passing in NULL means
    "return my priority". */
    uxPriority = uxTaskPriorityGet( NULL );

    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( "Task 1 is running\n" );

        /* Setting the Task 2 priority above the Task 1 priority will cause
        Task 2 to immediately start running (as then Task 2 will have the higher
        priority of the two created tasks).  Note the use of the handle to task
        2 (xTask2Handle) in the call to vTaskPrioritySet().  Listing 24 shows how
        the handle was obtained. */
        vPrintString( "About to raise the Task 2 priority\n" );
        vTaskPrioritySet( xTask2Handle, ( uxPriority + 1 ) );

        /* Task 1 will only run when it has a priority higher than Task 2.
        Therefore, for this task to reach this point Task 2 must already have
        executed and set its priority back down to below the priority of this
        task. */
    }
}
```

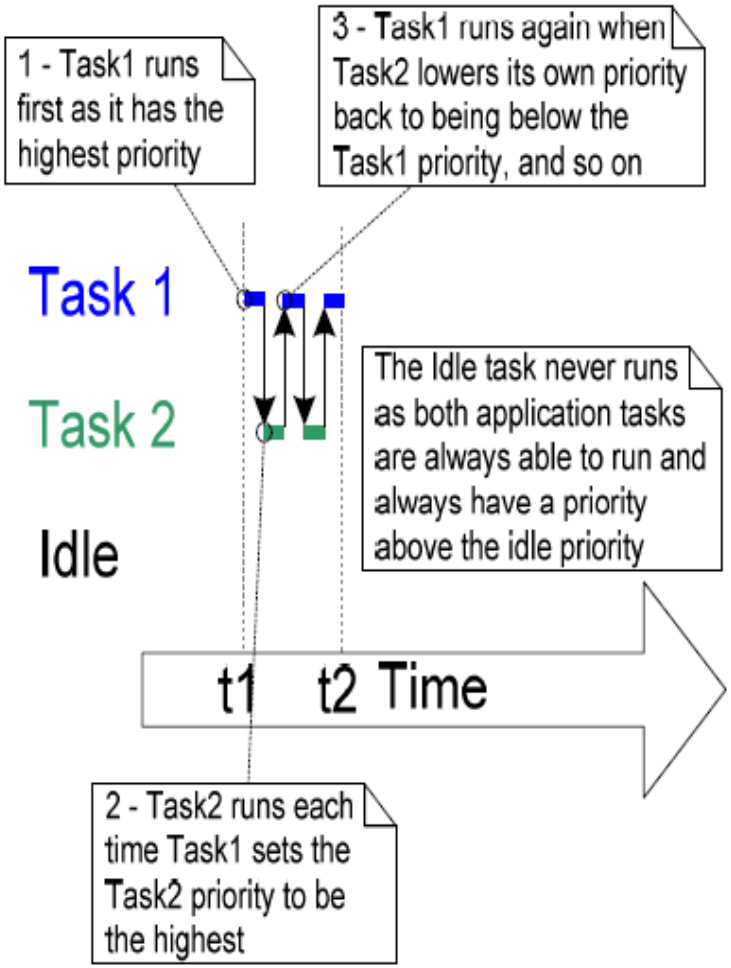
Listing 22. The implementation of Task 1 in Example 8

# Changing task priorities; Example 8

```
void vTask2( void *pvParameters )
{
unsigned portBASE_TYPE uxPriority;

    /* Task 1 will always run before this task as Task 1 is created with the
    higher priority.  Neither Task 1 nor Task 2 ever block so will always be
    in either the Running or the Ready state.

    Query the priority at which this task is running - passing in NULL means
    "return my priority". */
    uxPriority = uxTaskPriorityGet( NULL );

    for( ;; )
    {
        /* For this task to reach this point Task 1 must have already run and
        set the priority of this task higher than its own.

        Print out the name of this task. */
        vPrintString( "Task2 is running\n" );

        /* Set our priority back down to its original value.  Passing in NULL
        as the task handle means "change my priority".  Setting the
        priority below that of Task 1 will cause Task 1 to immediately start
        running again - pre-empting this task. */
        vPrintString( "About to lower the Task 2 priority\n" );
        vTaskPrioritySet( NULL, ( uxPriority - 2 ) );
    }
}
```

Listing 23. The implementation of Task 2 in Example 8

# Changing task priorities; Example 8



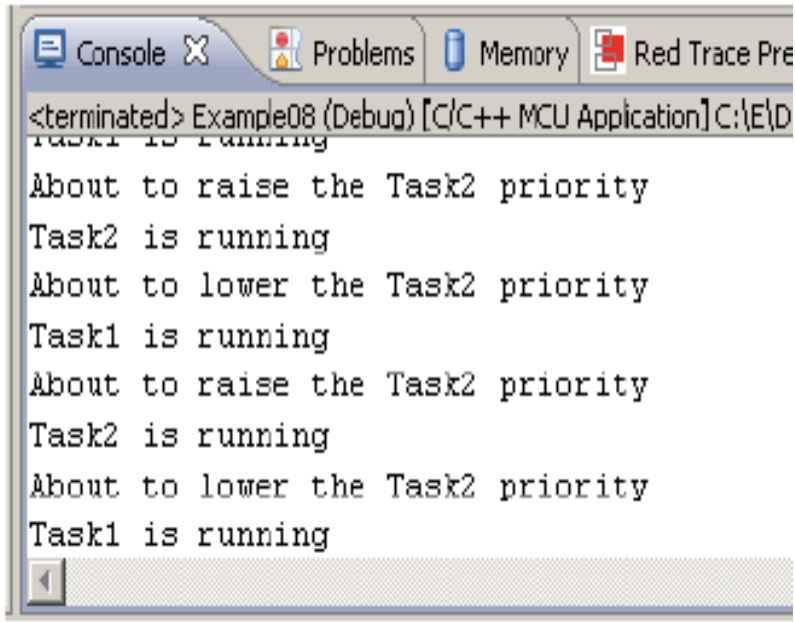




**Figure 14. The sequence of task execution when running Example 8**

## *The Scheduling Algorithm—A Summary*

- **Fixed Prioritized Pre-emptive Scheduling**
  - **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.**

- **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.**

# The Scheduling Algorithm—A Summary

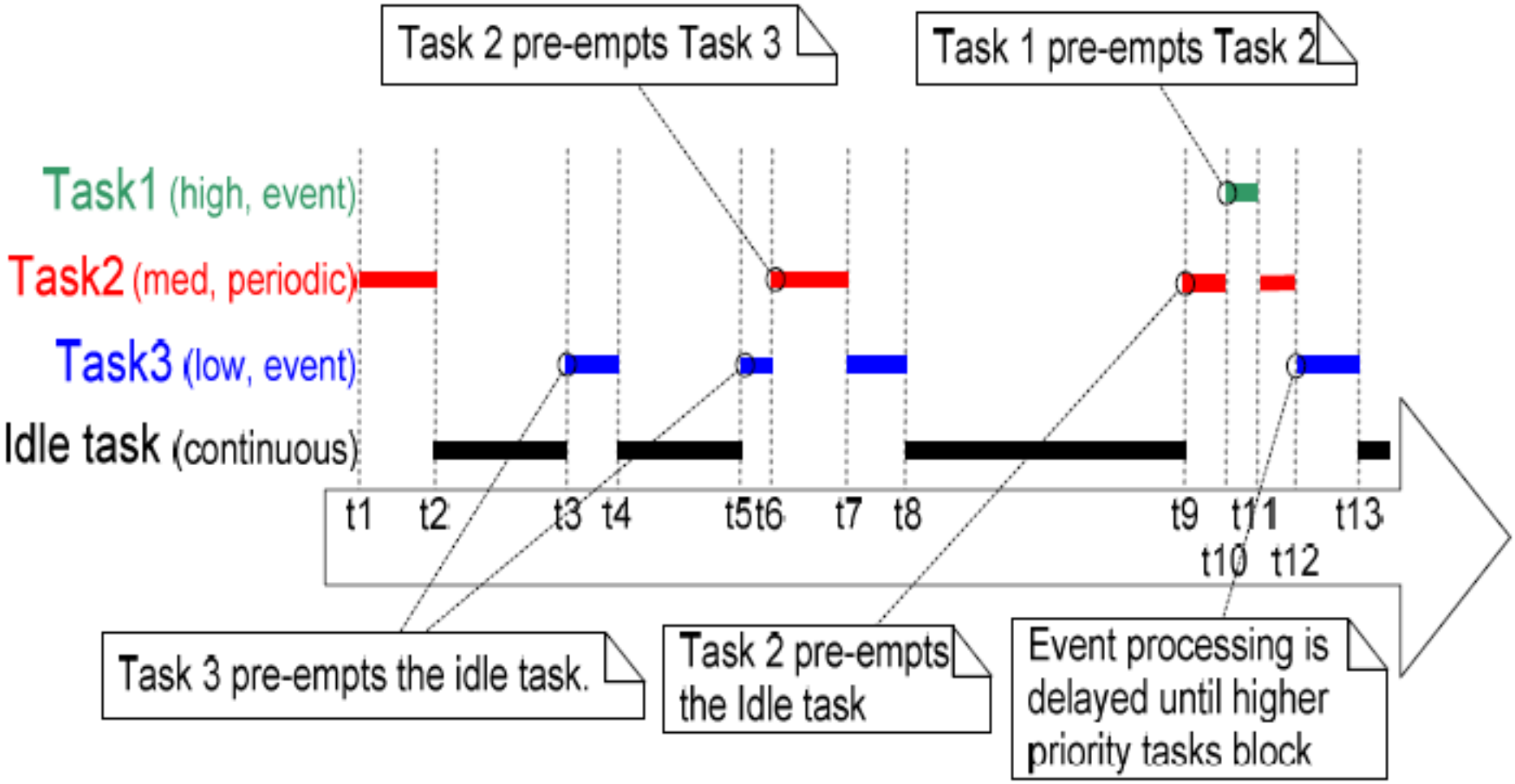



**Figure 18. Execution pattern with pre-emption points highlighted**


# Real Time Operating System "FreeRTOS" Idle Task Hook

Sherif Hammad

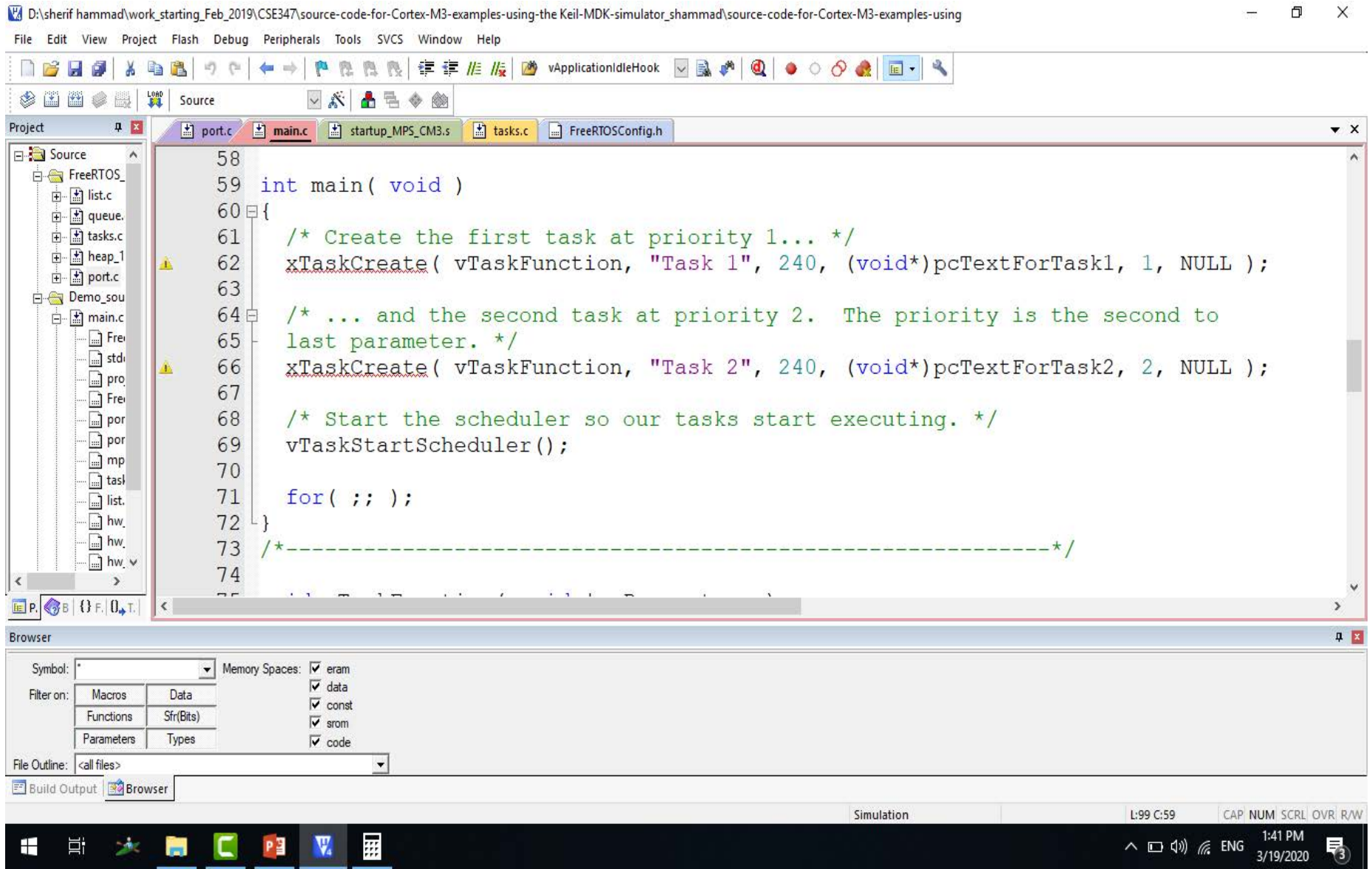
**Using the FreeRTOS Real Time Kernel - a Practical Guide - Cortex M3 Edition (FreeRTOS Tutorial Books)** Paperback – 2010

by Richard Barry

# *The Idle Task and the Idle Task Hook (Idle Task CallBack)*

- **An Idle task is automatically created by the scheduler when vTaskStartScheduler() is called.**
- **The idle task does very little more than sit in a loop**
- **The idle task has the lowest possible priority (priority zero), to ensure it never prevents a higher priority application task from entering the Running state**
- **Idle task is always pre-empted by higher priority tasks**
- **idle hook (or idle callback) function—a function that is called automatically by the idle task once per iteration of the idle task loop**
- **Common uses for the Idle task hook include:**
  - **Executing low priority, background, or continuous processing.**
  - **Measuring the amount of spare processing capacity.**
  - **Placing the processor into a low power mode**
- **configUSE_IDLE_HOOK must be set to 1 within FreeRTOSConfig.h for the idle hook function to get called.**

# Example 7: Idle Task CallBack



# Example 7: Idle Task CallBack

```
/* Declare a variable that will be incremented by the hook function. */
unsigned long ulIdleCycleCount = 0UL;

/* Idle hook functions MUST be called vApplicationIdleHook(), take no parameters,
and return void. */
void vApplicationIdleHook( void )
{
    /* This hook function does nothing but increment a counter. */
    ulIdleCycleCount++;
}
```

**Listing 18. A very simple Idle hook function**

```
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 AND the number of times ulIdleCycleCount
        has been incremented. */
        vPrintStringAndNumber( pcTaskName, ulIdleCycleCount );

        /* Delay for a period of 250 milliseconds. */
        vTaskDelay( 250 / portTICK_RATE_MS );
    }
}
```

**Listing 19. The source code for the example task prints out the ulIdleCycleCount value**

***How many times IdelTask is called between tasks calls?***
***For how long the CPU is into Idle Task?***

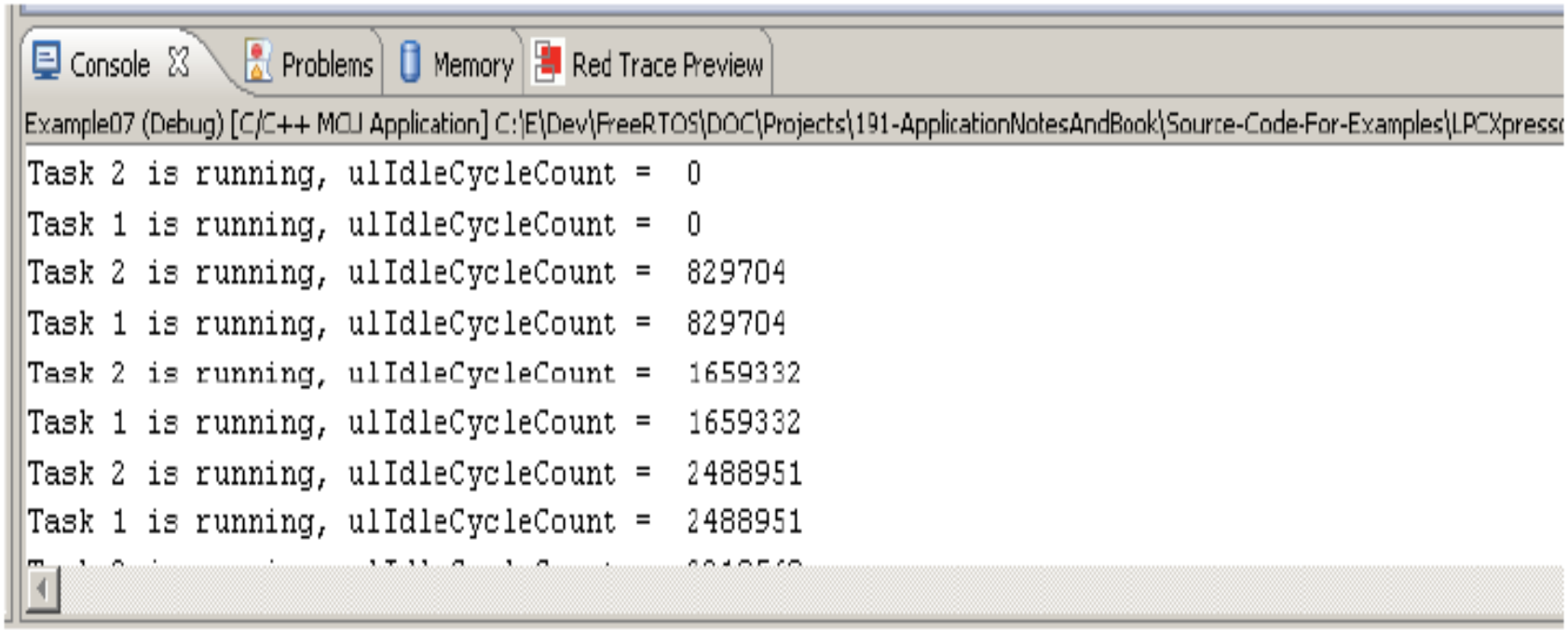



**Figure 13. The output produced when Example 7 is executed**


# Real Time Operating System
# "FreeRTOS"
# QUEUE; Task to Task Communication

***Sherif Hammad***

Using the FreeRTOS Real Time Kernel - a Practical Guide - Cortex M3 Edition (FreeRTOS Tutorial Books) by Richard Barry

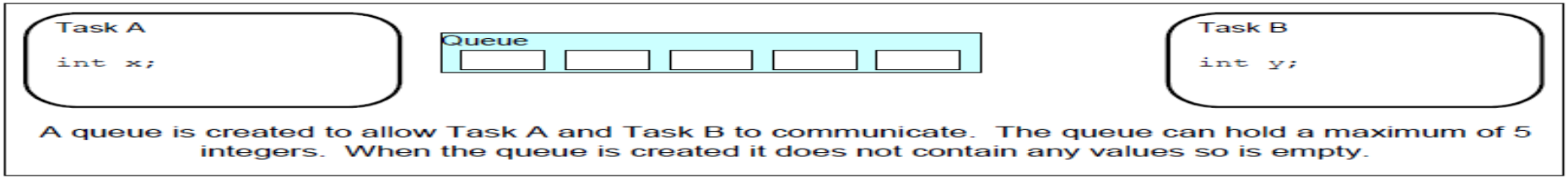

# *Agenda*

- What is a queue?
- How to create a queue.
- How a queue manages the data it contains.
- How to send data to a queue.
- How to receive data from a queue.
- What it means to block on a queue.
- The effect of task priorities when writing to and reading from a queue.

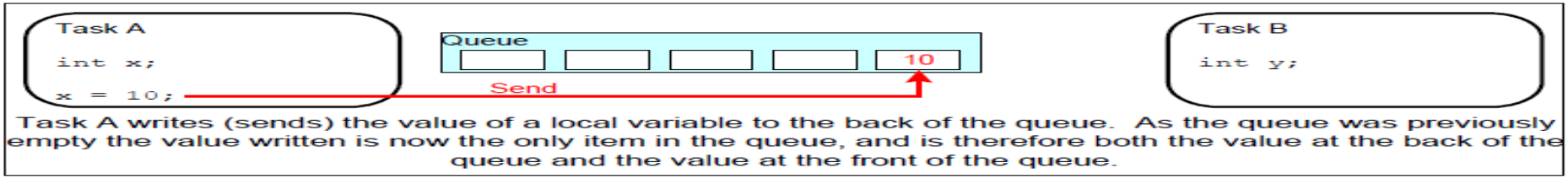
# What is Queue?

- A queue can hold a finite number of fixed size data items.
- The maximum number of items a queue can hold is called its 'length'.
- Both the length and the size of each data item are set when the queue is created.
- Normally, queues are used as First In First Out (FIFO) buffers where data is written to the end (tail) of the queue and removed from the front (head) of the queue.
- It is also possible to write to the front of a queue.
- Writing to a queue is a "physical append" to end of the queue.
- Reading from a queue is a "logical remove" from the top of the queue.

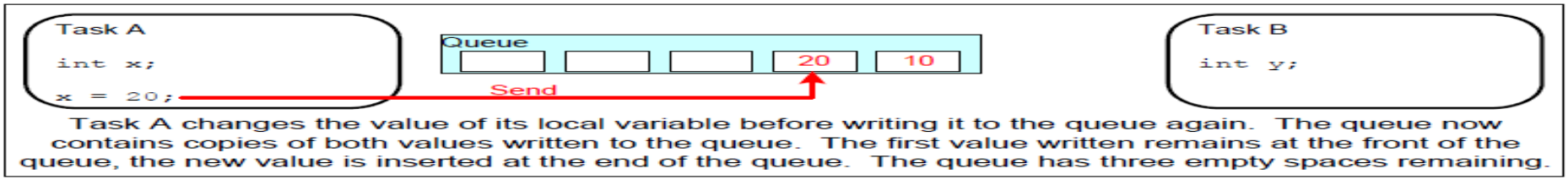
# What is Queue?



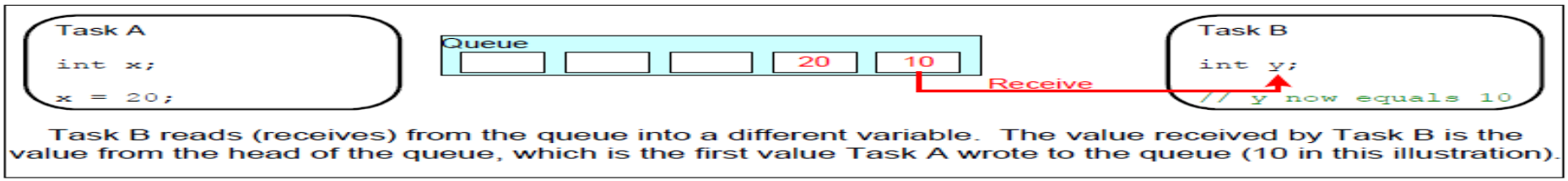

A queue is created to allow Task A and Task B to communicate. The queue can hold a maximum of 5 integers. When the queue is created it does not contain any values so is empty.



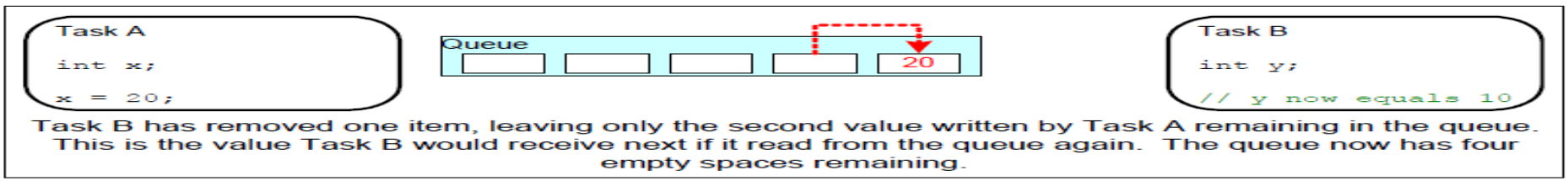

Task A writes (sends) the value of a local variable to the back of the queue. As the queue was previously empty the value written is now the only item in the queue, and is therefore both the value at the back of the queue and the value at the front of the queue.




Task A changes the value of its local variable before writing it to the queue again. The queue now contains copies of both values written to the queue. The first value written remains at the front of the queue, the new value is inserted at the end of the queue. The queue has three empty spaces remaining.




Task B reads (receives) from the queue into a different variable. The value received by Task B is the value from the head of the queue, which is the first value Task A wrote to the queue (10 in this illustration).




Task B has removed one item, leaving only the second value written by Task A remaining in the queue. This is the value Task B would receive next if it read from the queue again. The queue now has four empty spaces remaining.

**Figure 19. An example sequence of writes and reads to and from a queue**

# Blocking on Queue Reads

- When a task attempts to read from a queue it can optionally specify a 'block' time.
- 'block' time is the time the task should be kept in the Blocked state to wait for data to be available from the queue should the queue already be empty.
- A task that is in the Blocked state, waiting for data to become available from a queue, is automatically moved to the Ready state when another task or interrupt places data into the queue.
- The task will also be moved automatically from the Blocked state to the Ready state if the specified block time expires before data becomes available.
- Queues can have multiple readers so it is possible for a single queue to have more than one task blocked on it waiting for data.
- Only one task will be unblocked when data becomes available. The task that is unblocked will always be the highest priority task that is waiting for data.
- If the blocked tasks have equal priority, then the task that has been waiting for data the longest will be unblocked.

# Blocking on Queue Writes

- A task can optionally specify a block time when writing to a queue.
- The block time is the maximum time the task should be held in the Blocked state to wait for space to become available on the queue, should the queue already be full.
- Queues can have multiple writers, so it is possible for a full queue to have more than one task blocked on it waiting to complete a send operation.
- Only one task will be unblocked when space on the queue becomes available.
- The task that is unblocked will always be the highest priority task that is waiting for space.
- If the blocked tasks have equal priority, then the task that has been waiting for space the longest will be unblocked.

# Example 10. Blocking when receiving from a queue

```
/* Declare a variable of type xQueueHandle.  This is used to store the handle
to the queue that is accessed by all three tasks. */
xQueueHandle xQueue;


int main( void )
{
    /* The queue is created to hold a maximum of 5 values, each of which is
    large enough to hold a variable of type long. */
    xQueue = xQueueCreate( 5, sizeof( long ) );

    if( xQueue != NULL )
    {
        /* Create two instances of the task that will send to the queue.  The task
        parameter is used to pass the value that the task will write to the queue,
        so one task will continuously write 100 to the queue while the other task
        will continuously write 200 to the queue.  Both tasks are created at
        priority 1. */
        xTaskCreate( vSenderTask, "Sender1", 240, ( void * ) 100, 1, NULL );
        xTaskCreate( vSenderTask, "Sender2", 240, ( void * ) 200, 1, NULL );

        /* Create the task that will read from the queue.  The task is created with
        priority 2, so above the priority of the sender tasks. */
        xTaskCreate( vReceiverTask, "Receiver", 240, NULL, 2, NULL );

        /* Start the scheduler so the created tasks start executing. */
        vTaskStartScheduler();
    }
    else
    {
        /* The queue could not be created. */
    }

    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks.  If main() does reach here then it is likely that
    there was insufficient heap memory available for the idle task to be created.
    Chapter 5 provides more information on memory management. */
```

# Example 10. Blocking when receiving from a queue

```
static void vSenderTask( void *pvParameters )
{
long lValueToSend;
portBASE_TYPE xStatus;

    /* Two instances of this task are created so the value that is sent to the
    queue is passed in via the task parameter - this way each instance can use
    a different value.  The queue was created to hold values of type long,
    so cast the parameter to the required type. */
    lValueToSend = ( long ) pvParameters;

    /* As per most tasks, this task is implemented within an infinite loop. */
    for( ;; )
    {
        /* Send the value to the queue.

        The first parameter is the queue to which data is being sent.  The
        queue was created before the scheduler was started, so before this task
        started to execute.

        The second parameter is the address of the data to be sent, in this case
        the address of lValueToSend.

        The third parameter is the Block time – the time the task should be kept
        in the Blocked state to wait for space to become available on the queue
        should the queue already be full.  In this case a block time is not
        specified because the queue should never contain more than one item and
        therefore never be full. */
        xStatus = xQueueSendToBack( xQueue, &lValueToSend, 0 );

        if( xStatus != pdPASS )
        {
            /* The send operation could not complete because the queue was full -
            this must be an error as the queue should never contain more than
            one item! */
            vPrintString( "Could not send to the queue.\n" );
        }

        /* Allow the other sender task to execute.  taskYIELD() informs the
        scheduler that a switch to another task should occur now rather than
        keeping this task in the Running state until the end of the current time
        slice. */
        taskYIELD();
    }
}
```

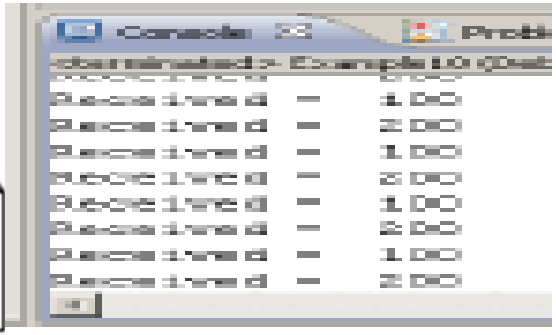
Listing 35. Implementation of the sending task used in Example 10

# Example 10. Blocking when receiving from a queue

```
static void vReceiverTask( void *pvParameters )
{
/* Declare the variable that will hold the values received from the queue. */
long lReceivedValue;
portBASE_TYPE xStatus;
const portTickType xTicksToWait = 100 / portTICK_RATE_MS;

      /* This task is also defined within an infinite loop. */
      for( ;; )
      {
            /* This call should always find the queue empty because this task will
            immediately remove any data that is written to the queue. */
            if( uxQueueMessagesWaiting( xQueue ) != 0 )
            {
                  vPrintString( "Queue should have been empty!\n" );
            }

            /* Receive data from the queue.

            The first parameter is the queue from which data is to be received.  The
            queue is created before the scheduler is started, and therefore before this
            task runs for the first time.

            The second parameter is the buffer into which the received data will be
            placed.  In this case the buffer is simply the address of a variable that
            has the required size to hold the received data.

            The last parameter is the block time – the maximum amount of time that the
            task should remain in the Blocked state to wait for data to be available
            should the queue already be empty.  In this case the constant
            portTICK_RATE_MS is used to convert 100 milliseconds to a time specified in
            ticks. */
            xStatus = xQueueReceive( xQueue, &lReceivedValue, xTicksToWait );

            if( xStatus == pdPASS )
            {
                  /* Data was successfully received from the queue, print out the received
                  value. */
                  vPrintStringAndNumber( "Received = ", lReceivedValue );
            }
            else
            {
                  /* Data was not received from the queue even after waiting for 100ms.
                  This must be an error as the sending tasks are free running and will be
                  continuously writing to the queue. */
                  vPrintString( "Could not receive from the queue.\n" );
            }
      }
}
```

Listing 36. Implementation of the receiver task for Example 10

# Example 10. Blocking when receiving from a queue



1 - The Receiver task runs first because it has the highest priority. It attempts to read from the queue. The queue is empty so the Receiver enters the Blocked state to wait for data to become available. Once the Receiver is blocked Sender 2 can run.

3 - The Receiver task empties the queue then enters the Blocked state again, allowing Sender 2 to execute once more. Sender 2 immediately Yields to Sender 1.

5 - The Receiver task empties the queue then enters the Blocked State. CPU gets back to preempted tasks

Receiver

Sender 2

Sender 1

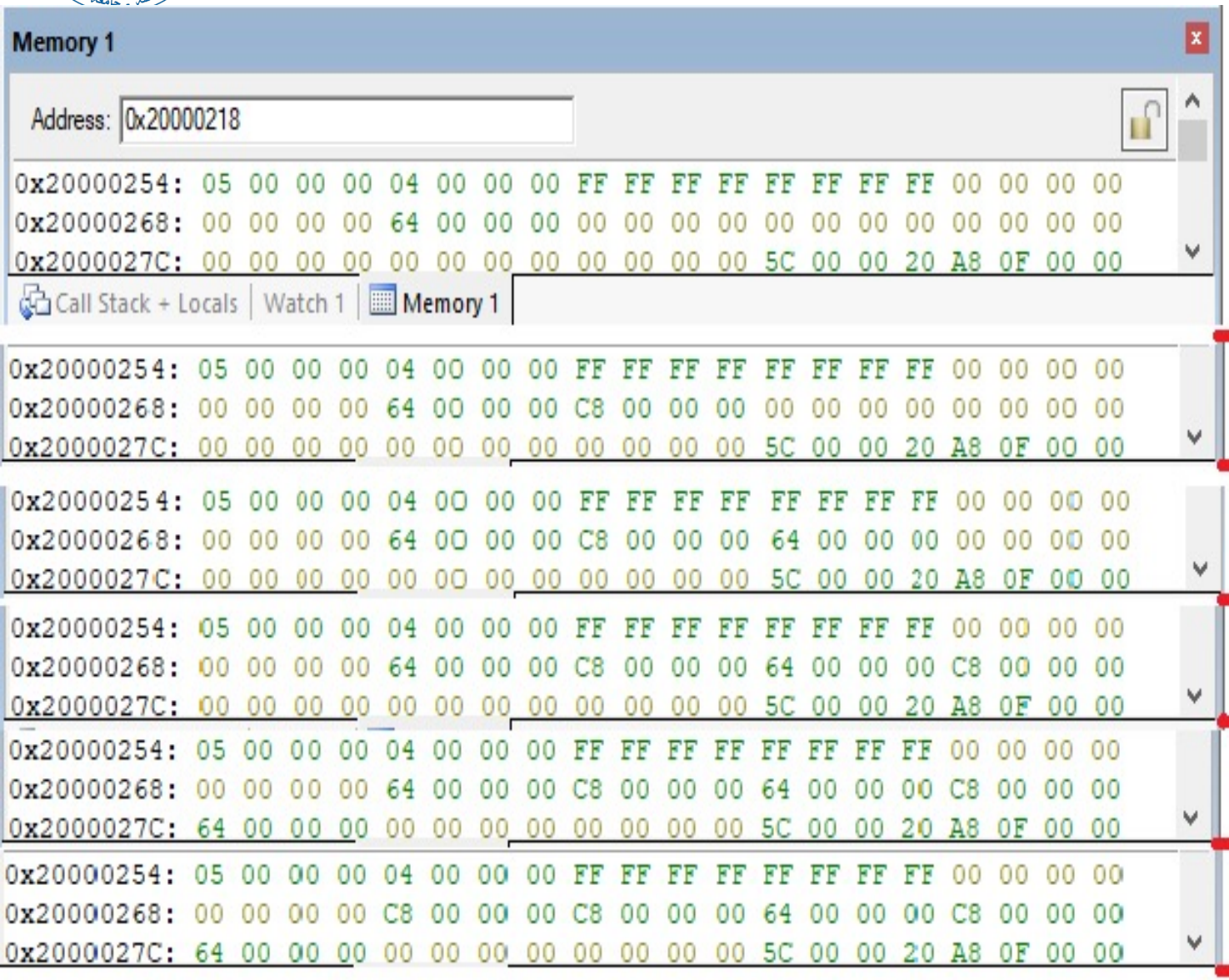
t1 Time

6&7- Premepted Sender 2 completes and Yields and so Sender 1. Receiever is Blocked. So Sender 2 writes to the empty queue before preempted again

2 - Sender two writes to the queue, causing the Receiver to exit the Blocked state. The Receiver has the highest priority so pre-empts Sender 2.

4 - Sender 1 writes to the queue, causing the Receiver to exit the Blocked state and pre-empt Sender 1 - and so it goes on .....

# Example 10. Blocking when receiving from a queue




Sender 1: Write No. 1

Receiver reads 64 just after

Sender 2: Write No. 2

Receiver reads C8 just after

Sender 1: Write No. 3

Receiver reads 64 just after

Sender 2: Write No. 4

Receiver reads C8 just after

Sender 1: Write No. 5

Receiver reads 64 just after

Sender 2: Writes No. 6 Circularly

Receiver reads C8 just after


# Real Time Operating System "FreeRTOS" Interrupt Management

***Sherif Hammad***

Using the FreeRTOS Real Time Kernel - a Practical Guide - Cortex M3 Edition (FreeRTOS Tutorial Books)

by Richard Barry

# *Agenda*

- How Embedded real-time systems have to take actions in response to events that originate from the environment (Interrupt Event Based)
- Which FreeRTOS API functions can be used from within an ISR.
- How a deferred interrupt scheme can be implemented.
- How to create and use binary semaphores and counting semaphores.

# *Deferred Interrupt Processing*

## *Binary Semaphores Used for Synchronization*

- A Binary Semaphore can be used to unblock a task each time a particular interrupt occurs
- Binary Semaphore synchronizes the task with the interrupt.
- This allows the majority of the interrupt event processing to be implemented within the synchronized task
- Only a very fast and short portion remaining directly in the ISR.
- The interrupt processing is said to have been ‘deferred’ to a ‘handler’ task.

# *Deferred Interrupt Processing*

## *Binary Semaphores Used for Synchronization*

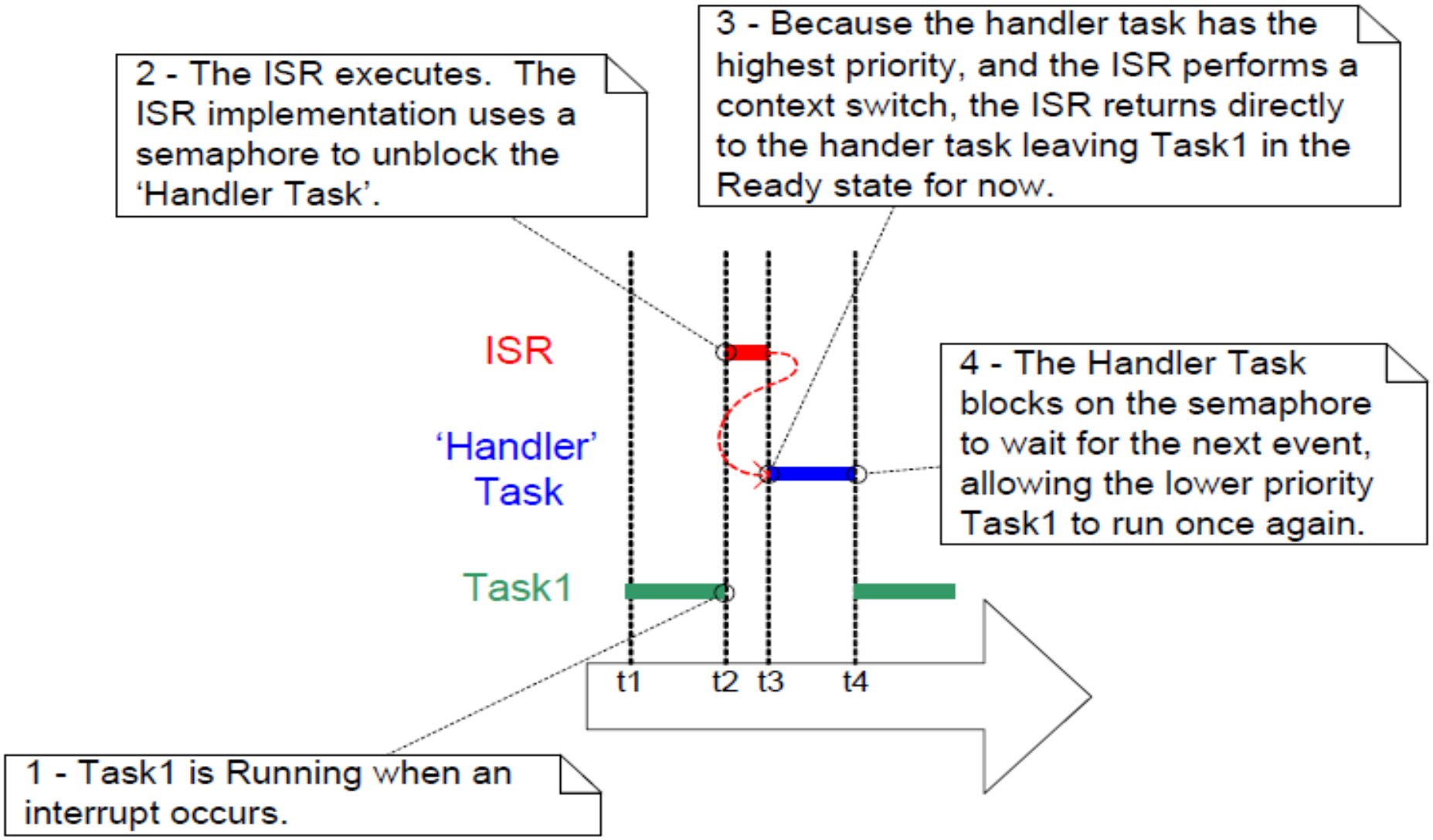
- If the interrupt processing is particularly time critical, then the handler task priority can be set to ensure that the handler task always pre-empts the other tasks in the system.

- The ISR can then be implemented to include a context switch to ensure that the ISR returns directly to the handler task when the ISR itself has completed executing.

- This has the effect of ensuring that the entire event processing executes contiguously in time, just as if it had all been implemented within the ISR itself.

# Deferred Interrupt Processing

## Binary Semaphores Used for Synchronization

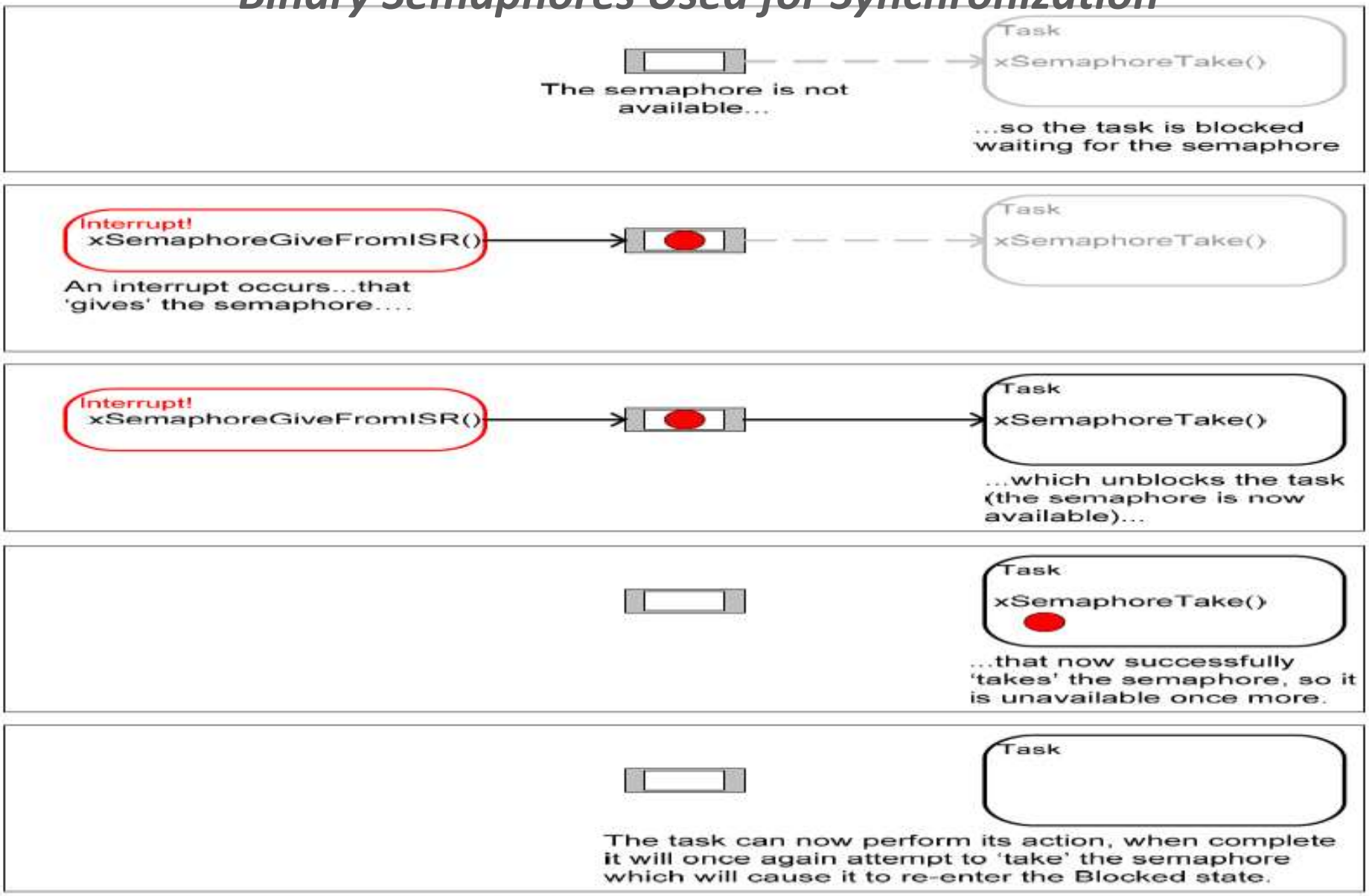


# Deferred Interrupt Processing
## Binary Semaphores Used for Synchronization

- The handler task uses a blocking 'take' call to a semaphore as a means of entering the Blocked state to wait for the event to occur.
- When the event occurs, the ISR uses a 'give' operation on the same semaphore to unblock the task so that the required event processing can proceed.
- In this interrupt synchronization scenario, the binary semaphore can be considered conceptually as a queue with a length of one. The queue can contain a maximum of one item at any time, so is always either empty or full (hence, binary). By calling xSemaphoreTake()
- The handler task effectively attempts to read from the queue with a block time, causing the task to enter the Blocked state if the queue is empty.
- When the event occurs, the ISR uses the xSemaphoreGiveFromISR() function to place a token (the semaphore) into the queue, making the queue full. This causes the handler task to exit the Blocked state and remove the token, leaving the queue empty once more.
- When the handler task has completed its processing, it once more attempts to read from the queue and, finding the queue empty, re-enters the Blocked state to wait for the next event.

# Deferred Interrupt Processing

## Binary Semaphores Used for Synchronization



## *Example 12. Using a binary semaphore to synchronize a task with an interrupt*

- **This example uses a binary semaphore to unblock a task from within an interrupt service routine—effectively synchronizing the task with the interrupt.**
- **A simple periodic task is used to generate an interrupt every 500 milliseconds.**
- **A software generated interrupt is used because it allows the time at which the interrupt occurs to be controlled, which in turn allows the sequence of execution to be observed more easily.**
- **The interrupt service routine, which is simply a standard C function.**
- **It does very little other than clear the interrupt and 'give' the semaphore to unblock the handler task.**

# *Example 12. Using a binary semaphore to synchronize a task with an interrupt*

- The macro portEND_SWITCHING_ISR() is part of the FreeRTOS Cortex-M3 port and is the ISR safe equivalent of taskYIELD().
- It will force a context switch only if its parameter is not zero (not equal to pdFALSE).
- Note how xHigherPriorityTaskWoken is used. It is initialized to pdFALSE before being passed by reference into xSemaphoreGiveFromISR()
- it will get set to pdTRUE only if xSemaphoreGiveFromISR() causes a task of equal or higher priority than the currently executing task to leave the blocked state. portEND_SWITCHING_ISR() then performs a context switch only if xHigherPriorityTaskWoken equals pdTRUE.

# Example 12. Using a binary semaphore to synchronize a task with an interrupt

```
static void vPeriodicTask( void *pvParameters )
{
    /* As per most tasks, this task is implemented within an infinite loop. */
    for( ;; )
    {
        /* This task is just used to 'simulate' an interrupt.  This is done by
        periodically generating a software interrupt. */
        vTaskDelay( 500 / portTICK_RATE_MS );

        /* Generate the interrupt, printing a message both before hand and
        afterwards so the sequence of execution is evident from the output. */
        vPrintString( "Periodic task - About to generate an interrupt.\n" );
        mainTRIGGER_INTERRUPT();
        vPrintString( "Periodic task - Interrupt generated.\n\n" );
    }
}
```

Listing 45. Implementation of the task that periodically generates a software interrupt in Example 12

# Example 12. Using a binary semaphore to synchronize a task with an interrupt

```
static void vHandlerTask( void *pvParameters )
{
    /* As per most tasks, this task is implemented within an infinite loop.

    Take the semaphore once to start with so the semaphore is empty before the
    infinite loop is entered.  The semaphore was created before the scheduler
    was started so before this task ran for the first time.*/
    xSemaphoreTake( xBinarySemaphore, 0 );

    for( ;; )
    {
        /* Use the semaphore to wait for the event.  The task blocks
        indefinitely meaning this function call will only return once the
        semaphore has been successfully obtained - so there is no need to check
        the returned value. */
        xSemaphoreTake( xBinarySemaphore, portMAX_DELAY );

        /* To get here the event must have occurred.  Process the event (in this
        case we just print out a message). */
        vPrintString( "Handler task - Processing event.\n" );
    }
}
```

Listing 46. The implementation of the handler task (the task that synchronizes with the interrupt) in Example 12

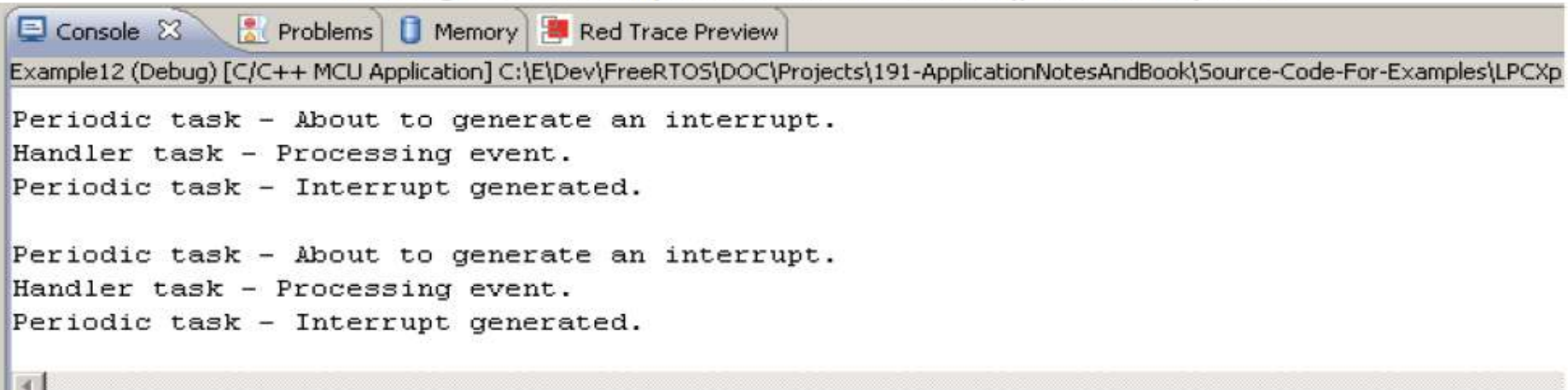
# Example 12. Using a binary semaphore to synchronize a task with an interrupt

```
void vSoftwareInterruptHandler( void )
{
portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;

    /* 'Give' the semaphore to unblock the task. */
    xSemaphoreGiveFromISR( xBinarySemaphore, &xHigherPriorityTaskWoken );

    /* Clear the software interrupt bit using the interrupt controllers
    Clear Pending register. */
    mainCLEAR_INTERRUPT();

    /* Giving the semaphore may have unblocked a task - if it did and the
    unblocked task has a priority equal to or above the currently executing
    task then xHigherPriorityTaskWoken will have been set to pdTRUE and
    portEND_SWITCHING_ISR() will force a context switch to the newly unblocked
    higher priority task.

    NOTE: The syntax for forcing a context switch within an ISR varies between
    FreeRTOS ports.  The portEND_SWITCHING_ISR() macro is provided as part of
    the Corte M3 port layer for this purpose.  taskYIELD() must never be called
    from an ISR! */
    portEND_SWITCHING_ISR( xHigherPriorityTaskWoken );
}
```

# Example 12. Using a binary semaphore to synchronize a task with an interrupt

```
int main( void )
{
    /* Configure both the hardware and the debug interface. */
    vSetupEnvironment();

    /* Before a semaphore is used it must be explicitly created.  In this example
    a binary semaphore is created. */
    vSemaphoreCreateBinary( xBinarySemaphore );

    /* Check the semaphore was created successfully. */
    if( xBinarySemaphore != NULL )
    {
        /* Enable the software interrupt and set its priority. */
        prvSetupSoftwareInterrupt();

        /* Create the 'handler' task.  This is the task that will be synchronized
        with the interrupt.  The handler task is created with a high priority to
        ensure it runs immediately after the interrupt exits.  In this case a
        priority of 3 is chosen. */
        xTaskCreate( vHandlerTask, "Handler", 240, NULL, 3, NULL );

        /* Create the task that will periodically generate a software interrupt.
        This is created with a priority below the handler task to ensure it will
        get preempted each time the handler task exits the Blocked state. */
        xTaskCreate( vPeriodicTask, "Periodic", 240, NULL, 1, NULL );

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

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

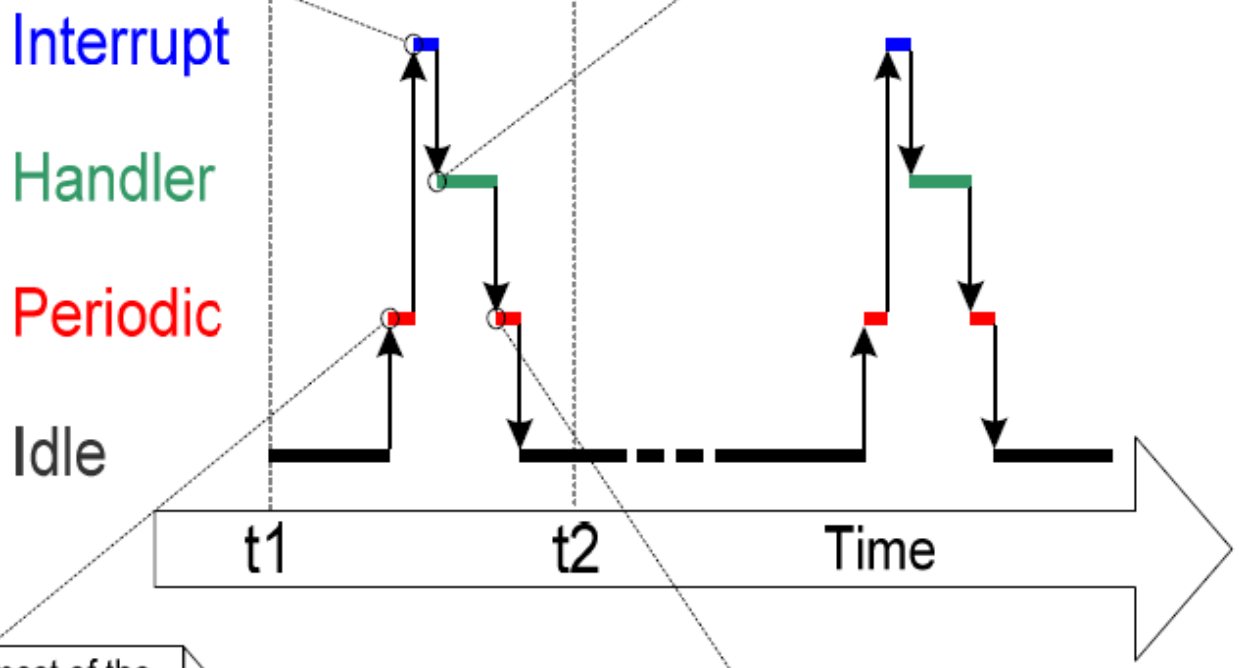
Listing 48. The implementation of main() for Example 12



# Example 12. Using a binary semaphore to synchronize a task with an interrupt

2 - The Periodic task prints its first message then forces an interrupt. The interrupt service routine executes immediately.

3 - The interrupt 'gives' the semaphore, causing the Handler task to unblock. The interrupt service routine then returns directly to the Handler task because the Handler task is the highest priority Ready state task. The handler task prints out its message before returning to the Blocked state to wait for the next interrupt.




1 - The Idle task is running most of the time. Every 500ms its gets pre-empted by the Periodic task.

4 - The Periodic task is once again the highest priority task - it prints out its second message before entering the Blocked state again to wait for the next time period. This leaves just the Idle task able to run.


# Real Time Operating System
# "FreeRTOS"
# Interrupt Management
# Counting Semaphores/Queues

***Sherif Hammad***

Using the FreeRTOS Real Time Kernel - a Practical Guide - Cortex M3 Edition (FreeRTOS Tutorial Books) by Richard Barry

# *Agenda*

- **Binary semaphore pitfall**
- **Counting semaphore handling of fast interrupts (Example 13)**
- **Sending/Receiving to Queues from ISR (Example 14)**

# Binary Semaphore Synchronization (Maximum One Pending Interrupt)

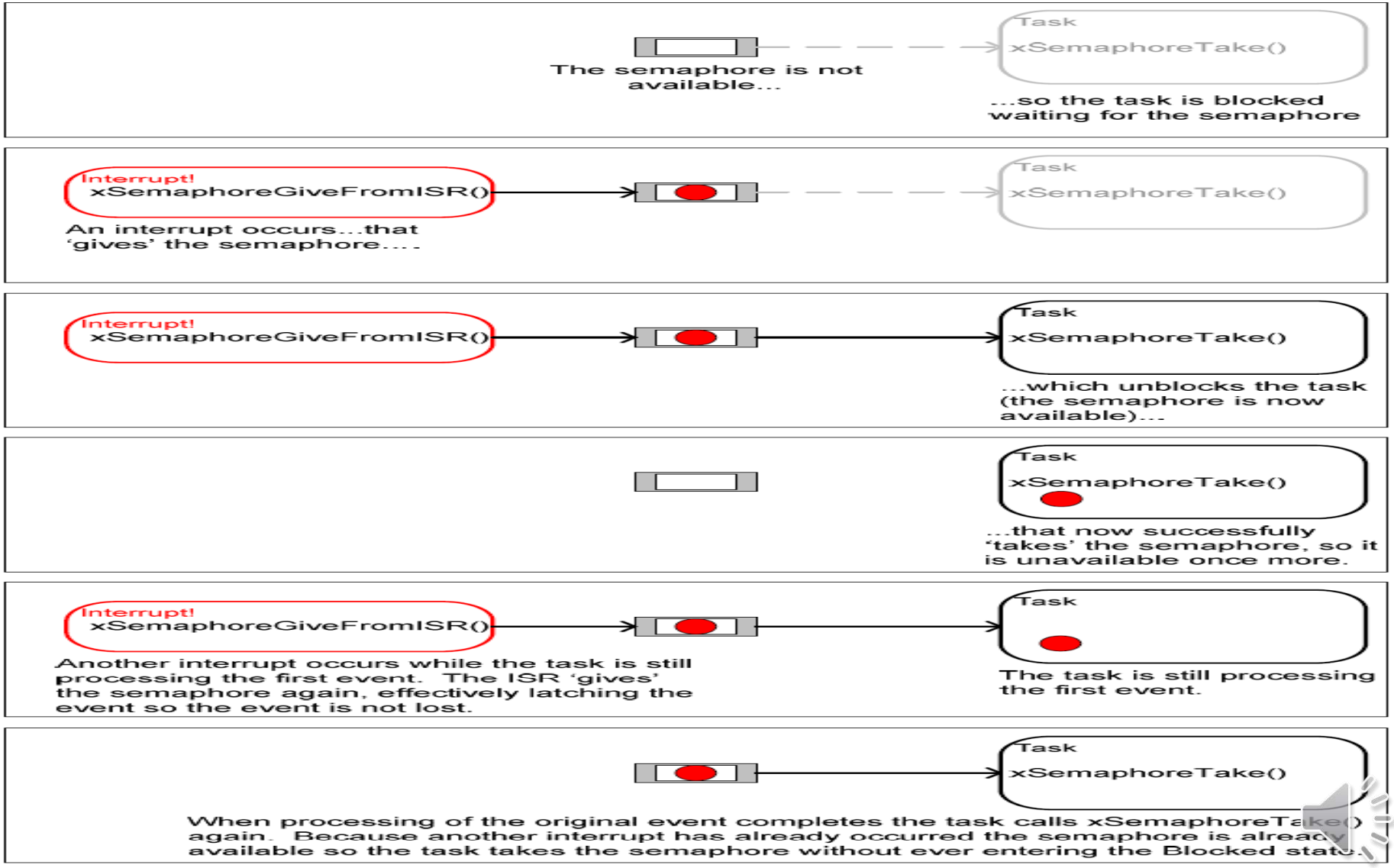



Figure 29. A binary semaphore can latch at most one event

# Counting Semaphore Synchronization


Figure 30. Using a counting semaphore to 'count' events

# Counting Semaphore; Use Cases

- **Counting events**
  - An event handler will 'give' a semaphore each time an event occurs—causing the semaphore's count value to be incremented on each 'give'.
  - A handler task will 'take' a semaphore each time it processes an event—causing the semaphore's count value to be decremented on each take.
  - The count value is the difference between the number of events that have occurred and the number that have been processed.
  - Counting semaphores that are used to count events are created with an initial count value of zero.

- **Resource management.**
  - The count value indicates the number of resources available.
  - To obtain control of a resource a task must first obtain a semaphore—decrementing the semaphore's count value.
  - When the count value reaches zero, there are no free resources.
  - When a task finishes with the resource, it 'gives' the semaphore back—incrementing the semaphore's count value.
  - Counting semaphores that are used to manage resources are created so that their initial count value equals the number of resources that are available.

# Counting Semaphore; FreeRtos API

```
xSemaphoreHandle xSemaphoreCreateCounting( unsigned portBASE_TYPE uxMaxCount,
                                           unsigned portBASE_TYPE uxInitialCount );
```

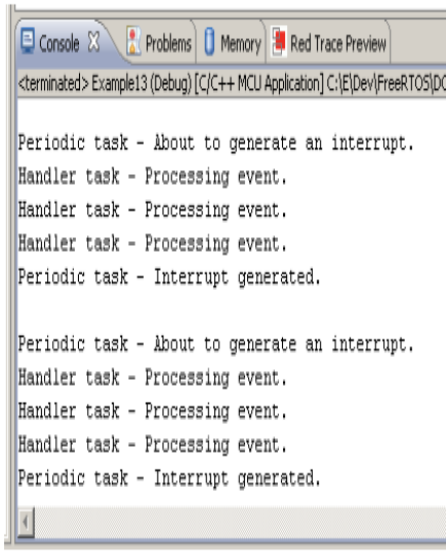
- **uxMaxCount: The maximum value the semaphore will count to.**
  - uxMaxCount **value is effectively the** length of the "queue".
  - **When the semaphore is to be used to count or** latch events, **uxMaxCount is the** maximum number of events that can be latched.
  - **When the semaphore is to be used to manage access to a collection of resources,** uxMaxCount **should be set to the** total number of resources that are available.

- **uxInitialCount: The initial count value of the semaphore after it has been created.**
  - **When the semaphore is to be used to count** or latch events, uxInitialCount **should be set** to zero—as, **presumably, when the semaphore is created, no events have yet occurred.**
  - **When the semaphore is to be used to manage access to a collection of** resources, **uxInitialCount should be set to equal** uxMaxCount—as, **presumably, when the semaphore is created, all** the resources are available.

# Counting Semaphore; Example 13

```
/* Before a semaphore is used it must be explicitly created.  In this example
a counting semaphore is created.  The semaphore is created to have a maximum
count value of 10, and an initial count value of 0. */
xCountingSemaphore = xSemaphoreCreateCounting( 10, 0 );
```

```
void vSoftwareInterruptHandler( void )
{
portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;

    /* 'Give' the semaphore multiple times.  The first will unblock the handler
    task, the following 'gives' are to demonstrate that the semaphore latches
    the events to allow the handler task to process them in turn without any
    events getting lost.  This simulates multiple interrupts being taken by the
    processor, even though in this case the events are simulated within a single
    interrupt occurrence.*/
    xSemaphoreGiveFromISR( xCountingSemaphore, &xHigherPriorityTaskWoken );
    xSemaphoreGiveFromISR( xCountingSemaphore, &xHigherPriorityTaskWoken );
    xSemaphoreGiveFromISR( xCountingSemaphore, &xHigherPriorityTaskWoken );

    /* Clear the software interrupt bit using the interrupt controllers Clear
    Pending register. */
    mainCLEAR_INTERRUPT();

    /* Giving the semaphore may have unblocked a task - if it did and the
    unblocked task has a priority equal to or above the currently executing
    task then xHigherPriorityTaskWoken will have been set to pdTRUE and
    portEND_SWITCHING_ISR() will force a context switch to the newly unblocked
    higher priority task.

    NOTE: The syntax for forcing a context switch within an ISR varies between
    FreeRTOS ports.  The portEND_SWITCHING_ISR() macro is provided as part of
    the Cortex-M3 port layer for this purpose.  taskYIELD() must never be called
    from an ISR! */
    portEND_SWITCHING_ISR( xHigherPriorityTaskWoken );
}
```




Listing 51. The implementation of the interrupt service routine used by Example 13

# Using Queues within an Interrupt Service Routine

```
portBASE_TYPE xQueueSendToBackFromISR( xQueueHandle xQueue,
                                       void *pvItemToQueue
                                       portBASE_TYPE *pxHigherPriorityTaskWoken
                                     );
```

Listing 53. The xQueueSendToBackFromISR() API function prototype

- pxHigherPriorityTaskWoken It is possible that a single queue will have one or more tasks blocked on it waiting for data to become available.
- Calling xQueueSendToFrontFromISR() or xQueueSendToBackFromISR() can make data available, and so cause such a task to leave the Blocked state.
- If calling the API function causes a task to leave the Blocked state, and the unblocked task has a priority equal to or higher than the currently executing task (the task that was interrupted), then, internally, the API function will set *pxHigherPriorityTaskWoken to pdTRUE.

# Using Queues within an Interrupt Service Routine; Example 14

```
int main( void )
{
    /* Before a queue can be used it must first be created.  Create both queues
    used by this example.  One queue can hold variables of type unsigned long,
    the other queue can hold variables of type char*.  Both queues can hold a
    maximum of 10 items.  A real application should check the return values to
    ensure the queues have been successfully created. */
    xIntegerQueue = xQueueCreate( 10, sizeof( unsigned long ) );
    xStringQueue = xQueueCreate( 10, sizeof( char * ) );

    /* Enable the software interrupt and set its priority. */
    prvSetupSoftwareInterrupt();

    /* Create the task that uses a queue to pass integers to the interrupt service
    routine.  The task is created at priority 1. */
    xTaskCreate( vIntegerGenerator, "IntGen", 240, NULL, 1, NULL );

    /* Create the task that prints out the strings sent to it from the interrupt
    service routine.  This task is created at the higher priority of 2. */
    xTaskCreate( vStringPrinter, "String", 240, NULL, 2, NULL );

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

    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks.  If main() does reach here then it is likely that
    there was insufficient heap memory available for the idle task to be created.
    Chapter 5 provides more information on memory management. */
    for( ;; );
}
```

Listing 57. The main() function for Example 14

# Using Queues within an Interrupt Service Routine; Example 14

```
static void vIntegerGenerator( void *pvParameters )
{
portTickType xLastExecutionTime;
unsigned long ulValueToSend = 0;
int i;

    /* Initialize the variable used by the call to vTaskDelayUntil(). */
    xLastExecutionTime = xTaskGetTickCount();

    for( ;; )
    {
        /* This is a periodic task.  Block until it is time to run again.
        The task will execute every 200ms. */
        vTaskDelayUntil( &xLastExecutionTime, 200 / portTICK_RATE_MS );

        /* Send an incrementing number to the queue five times.  The values will
        be read from the queue by the interrupt service routine.  The interrupt
        service routine always empties the queue so this task is guaranteed to be
        able to write all five values, so a block time is not required. */
        for( i = 0; i < 5; i++ )
        {
            xQueueSendToBack( xIntegerQueue, &ulValueToSend, 0 );
            ulValueToSend++;
        }

        /* Force an interrupt so the interrupt service routine can read the
        values from the queue. */
        vPrintString( "Generator task - About to generate an interrupt.\n" );
        mainTRIGGER_INTERRUPT();
        vPrintString( "Generator task - Interrupt generated.\n\n" );
    }
}
```

Listing 54. The implementation of the task that writes to the queue in Example 14

# Using Queues within an Interrupt Service Routine; Example 14

```
void vSoftwareInterruptHandler( void )
{
portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;
static unsigned long ulReceivedNumber;

/* The strings are declared static const to ensure they are not allocated to the
interrupt service routine stack, and exist even when the interrupt service routine
is not executing. */
static const char *pcStrings[] =
{
     "String 0\n",
     "String 1\n",
     "String 2\n",
     "String 3\n"
};

     /* Loop until the queue is empty. */
     while( xQueueReceiveFromISR( xIntegerQueue,
                                  &ulReceivedNumber,
                                  &xHigherPriorityTaskWoken ) != errQUEUE_EMPTY )
     {
          /* Truncate the received value to the last two bits (values 0 to 3 inc.),
          then send the string that corresponds to the truncated value to the other
          queue. */
          ulReceivedNumber &= 0x03;
          xQueueSendToBackFromISR( xStringQueue,
                                   &pcStrings[ ulReceivedNumber ],
                                   &xHigherPriorityTaskWoken );
     }

     /* Clear the software interrupt bit using the interrupt controllers Clear
     Pending register. */
     mainCLEAR_INTERRUPT();

     /* xHigherPriorityTaskWoken was initialised to pdFALSE.  It will have then
     been set to pdTRUE only if reading from or writing to a queue caused a task
     of equal or greater priority than the currently executing task to leave the
     Blocked state.  When this is the case a context switch should be performed.
     In all other cases a context switch is not necessary.

     NOTE: The syntax for forcing a context switch within an ISR varies between
     FreeRTOS ports.  The portEND_SWITCHING_ISR() macro is provided as part of
     the Cortex-M3 port layer for this purpose.  taskYIELD() must never be called
     from an ISR! */
     portEND_SWITCHING_ISR( xHigherPriorityTaskWoken );
}
```

Listing 55. The implementation of the interrupt service routine used by Example 14

# Using Queues within an Interrupt Service Routine; Example 14

```
static void vStringPrinter( void *pvParameters )
{
char *pcString;

    for( ;; )
    {
        /* Block on the queue to wait for data to arrive. */
        xQueueReceive( xStringQueue, &pcString, portMAX_DELAY );

        /* Print out the string received. */
        vPrintString( pcString );
    }
}
```

**Listing 56. The task that prints out the strings received from the interrupt service routine in Example 14**

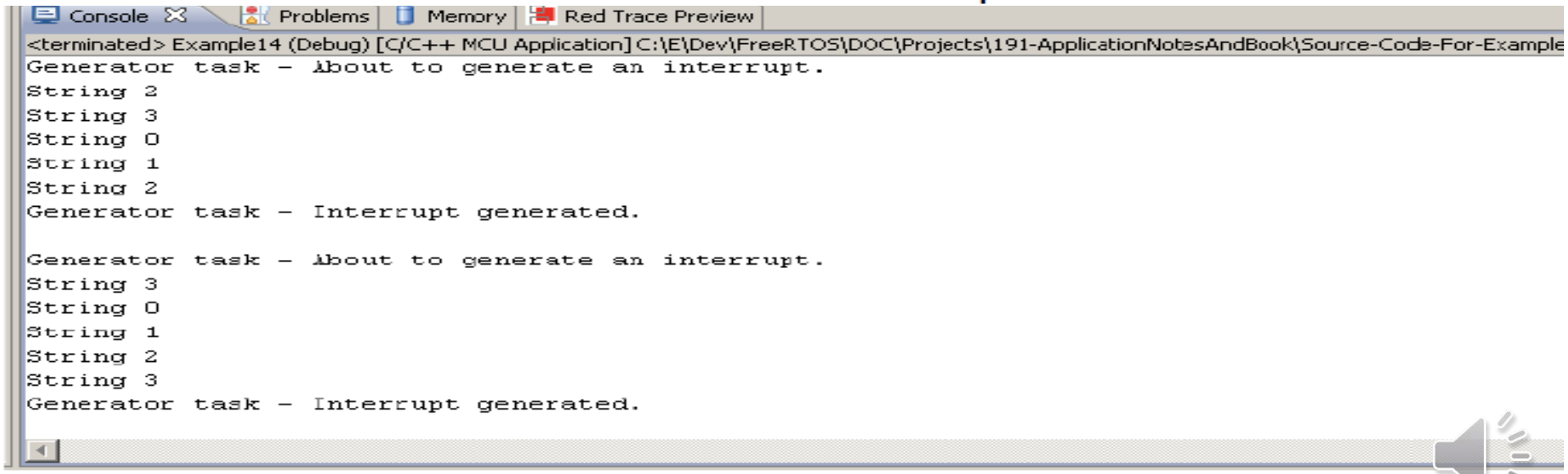



**Figure 32. The output produced when Example 14 is executed**

# Using Queues within an Interrupt Service Routine; Example 14

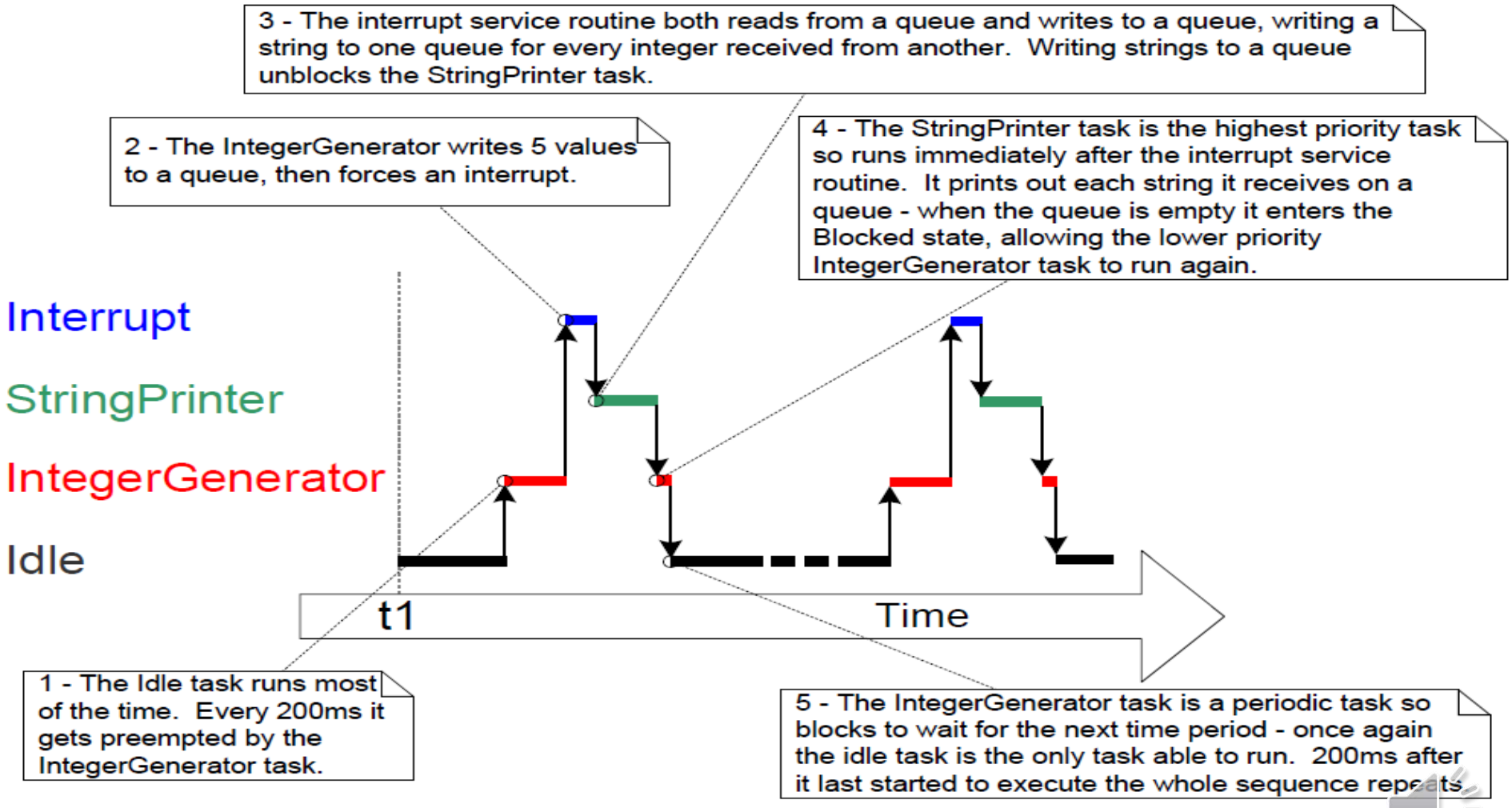



**Figure 33. The sequence of execution produced by Example 14**


# Real Time Operating System
# "FreeRTOS"
# Resource Management

***Sherif Hammad***

Using the FreeRTOS Real Time Kernel - a Practical Guide - Cortex M3 Edition (FreeRTOS Tutorial Books) by Richard Barry

# *Agenda*

- When and why resource management and control is necessary.
- What a critical section is.
- What mutual exclusion means.
- What it means to suspend the scheduler.
- How to use a mutex.
- What priority inversion is, and how priority inheritance can reduce (but not remove) its impact.
- What is DeadLock?

## *Why Resource Management*

- **In a multitasking system, there is potential for conflict if one task starts to access a resource, but does not complete its access before being transitioned out of the Running state.**

- "Resource Data" could be corrupted in the following cases:
  - **Accessing Peripherals:** Consider the following scenario where two tasks attempt to write to an LCD:
    - Task A executes and starts to write the string "Hello world" to the LCD.
    - Task A is pre-empted by Task B after outputting just the beginning of the string—"Hello w".
    - Task B writes "Abort, Retry, Fail?" to the LCD before entering the Blocked state.
    - Task A continues from the point at which it was pre-empted and completes outputting the remaining characters—"orld".
    - The LCD now displays the corrupted string "Hello wAbort, Retry, Fail?orld".

# Why Resource Management

```
/* The C code being compiled. */
GlobalVar |= 0x01;

/* The assembly code produced. */
LDR       r4,[pc,#284]
LDR       r0,[r4,#0x08] /* Load the value of GlobalVar into r0. */
ORR       r0,r0,#0x01   /* Set bit 0 of r0. */
STR       r0,[r4,#0x08] /* Write the new r0 value back to GlobalVar. */
```

Listing 59. An example read, modify, write sequence

- Read, Modify, Write Operations: This is a 'non-atomic' operation because it takes more than one instruction to complete and can be interrupted. Consider the following scenario where two tasks attempt to update a variable called GlobalVar:
  - Task A loads the value of GlobalVar into a register—the read portion of the operation.
  - Task A is pre-empted by Task B before it completes the modify and write portions of the same operation (before ORR instruction)
  - Task B updates the value of GlobalVar, then enters the Blocked state.
  - Task A continues from the point at which it was pre-empted. It modifies/corrupts the copy of the GlobalVar value calculated by Task B
- Non-atomic Access to Variables:
  - Updating multiple members of a structure
  - Updating a variable that is larger than the natural word size of the architecture (for example, updating a 64-bit variable on a 32-bit machine)

# Why Resource Management

- **Function Reentrancy**
  - A function is reentrant if it is safe to call the function from more than one task, or from both tasks and interrupts.
  - Each task maintains its own stack and its own set of core register values. If a function does not access any data other than data stored on the stack or held in a register, then the function is reentrant.

```
/* A parameter is passed into the function.  This will either be
passed on the stack or in a CPU register.  Either way is safe as
each task maintains its own stack and its own set of register
values.  */
long lAddOneHundered( long lVar1 )
{
/* This function scope variable will also be allocated to the stack
or a register, depending on the compiler and optimization level.  Each
task or interrupt that calls this function will have its own copy
of lVar2.  */
long lVar2;

    lVar2 = lVar1 + 100;

    /* Most likely the return value will be placed in a CPU register,
    although it too could be placed on the stack.  */
    return lVar2;
}
```

Listing 60. An example of a reentrant function

```
/* In this case lVar1 is a global variable so every task that calls
the function will be accessing the same single copy of the variable.  */
long lVar1;

long lNonsenseFunction( void )
{
/* This variable is static so is not allocated on the stack.  Each task
that calls the function will be accessing the same single copy of the
variable.  */
static long lState = 0;
long lReturn;

    switch( lState )
    {
        case 0  :   lReturn = lVar1 + 10;
                    lState = 1;
                    break;

        case 1  :   lReturn = lVar1 + 20;
                    lState = 0;
                    break;
    }
```

Listing 61. An example of a function that is not reentrant

# Basic Critical Sections

```
/* 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();
```

**Listing 62. 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, using a critical section as a crude method
    of mutual exclusion. */
    taskENTER_CRITICAL();
    {
        sprintf( cBuffer, "%s", pcString );
        consoleprint( cBuffer );
    }
    taskEXIT_CRITICAL();
}
```

**Listing 63. A possible implementation of vPrintString()**

# Basic Critical Sections

- Critical sections implemented in this way are a very crude method of providing **mutual exclusion**.
- They work by **disabling interrupts** up to the interrupt priority set by configMAX_SYSCALL_INTERRUPT_PRIORITY.
- Pre-emptive context switches can occur only from within an interrupt.
- As long as interrupts remain disabled, the task that called taskENTER_CRITICAL() is guaranteed to remain in the Running state until the critical section is exited.
- **Critical sections must be kept very short;** otherwise, they will adversely affect interrupt response times.
- Every call to taskENTER_CRITICAL() must be closely paired with a call to taskEXIT_CRITICAL().

# *Suspending (or Locking) the Scheduler*

- Critical sections can also be created by suspending the scheduler.
- Basic critical sections protect a region of code from access by other tasks and by interrupts.
- A critical section implemented by suspending the scheduler protects a region of code only from access by other tasks because interrupts remain enabled.
- FreeRTOS API functions should not be called while the scheduler is suspended.

# Suspending (or Locking) the Scheduler

### The vTaskSuspendAll() API Function

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

Listing 64. The vTaskSuspendAll() API function prototype

### The xTaskResumeAll() API Function

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

Listing 65. The xTaskResumeAll() API function prototype

```
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();
}
```

Listing 66. The implementation of vPrintString()

Table 19. xTaskResumeAll() return value

| Returned Value | Description |
|---|---|
| Returned value | Context switches that are requested while the scheduler is suspended are held pending and performed only as the scheduler is being resumed. A previously pending context switch being performed before xTaskResumeAll() returns results in the function returning pdTRUE. In all other cases, xTaskResumeAll() returns pdFALSE. |

# *Mutexes (and Binary Semaphores)*

- In a mutual exclusion scenario: the mutex can be thought of as a token associated with the shared resource.
- For a task to access the resource legitimately, it must first successfully 'take' the token (be the token holder).
- When the token holder has finished with the resource, it must 'give' the token back.
- Only when the token has been returned can another task successfully take the token and then safely access the same shared resource.
- A task is not permitted to access the shared resource unless it holds the token.
- A semaphore that is used for mutual exclusion must always be returned.
- A semaphore that is used for synchronization is normally discarded and not returned.
- There is no reason why a task cannot access the resource at any time, but each task 'agrees' not to do so, unless it is able to become the mutex holder.

# Mutexes (and Binary Semaphores)

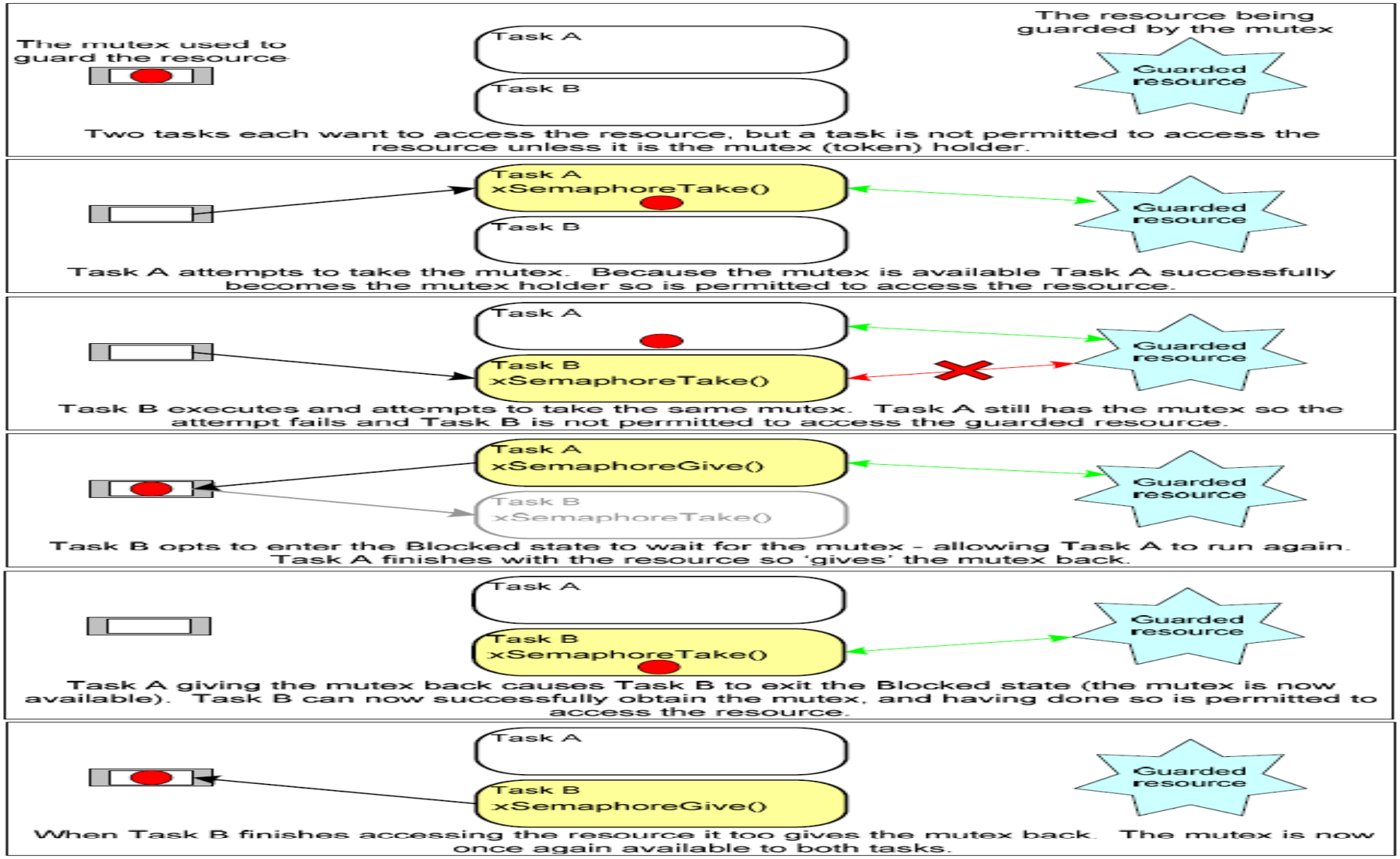



Figure 35. Mutual exclusion implemented using a mutex

# Mutexes (and Binary Semaphores)

```
int main( void )
{
    /* Before a semaphore is used it must be explicitly created.  In this example
    a mutex type semaphore is created. */
    xMutex = xSemaphoreCreateMutex();

    /* The tasks are going to use a pseudo random delay, seed the random number
    generator. */
    srand( 567 );

    /* Only create the tasks if the semaphore was created successfully. */
    if( xMutex != NULL )
    {
        /* Create two instances of the tasks that write to stdout.  The string
        they write is passed in as the task parameter.  The tasks are created
        at different priorities so some pre-emption will occur. */
        xTaskCreate( prvPrintTask, "Print1", 240,
                     "Task 1 ****************************************\n", 1, NULL );

        xTaskCreate( prvPrintTask, "Print2", 240,
                     "Task 2 ----------------------------------------\n", 2, NULL );

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

    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks.  If main() does reach here then it is likely that
    there was insufficient heap memory available for the idle task to be created.
    Chapter 5 provides more information on memory management. */
    for( ;; );
}
```


# Mutexes (and Binary Semaphores)

```
static void prvNewPrintString( const char *pcString )
{
static char cBuffer[ mainMAX_MSG_LEN ];

    /* The mutex is created before the scheduler is started so already
    exists by the time this task first executes.

    Attempt to take the mutex, blocking indefinitely to wait for the mutex if
    it is not available straight away.  The call to xSemaphoreTake() will only
    return when the mutex has been successfully obtained so there is no need to
    check the function return value.  If any other delay period was used then
    the code must check that xSemaphoreTake() returns pdTRUE before accessing
    the shared resource (which in this case is standard out). */
    xSemaphoreTake( xMutex, portMAX_DELAY );
    {
        /* The following line will only execute once the mutex has been
        successfully obtained.  Standard out can be accessed freely now as
        only one task can have the mutex at any one time. */
        sprintf( cBuffer, "%s", pcString );
        consoleprint( cBuffer );

        /* The mutex MUST be given back! */
    }
    xSemaphoreGive( xMutex );
}
```

Listing 68. The implementation of prvNewPrintString()

```
static void prvPrintTask( void *pvParameters )
{
char *pcStringToPrint;

    /* Two instances of this task are created so the string the task will send
    to prvNewPrintString() is passed into the task using the task parameter.
    Cast this to the required type. */
    pcStringToPrint = ( char * ) pvParameters;

    for( ;; )
    {
        /* Print out the string using the newly defined function. */
        prvNewPrintString( pcStringToPrint );

        /* Wait a pseudo random time.  Note that rand() is not necessarily
        reentrant, but in this case it does not really matter as the code does
        not care what value is returned.  In a more secure application a version
        of rand() that is known to be reentrant should be used - or calls to
        rand() should be protected using a critical section. */
        vTaskDelay( ( rand() & 0x1FF ) );
    }
}
```

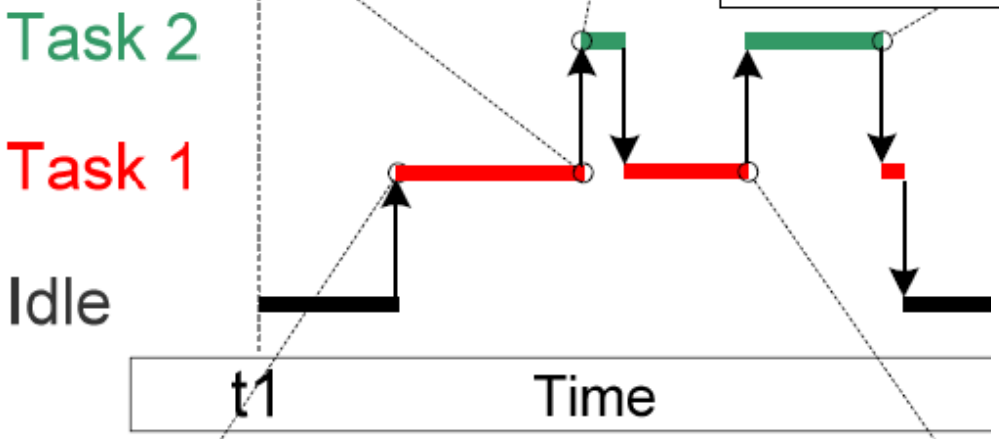
Listing 69. The implementation of prvPrintTask() for Example 15

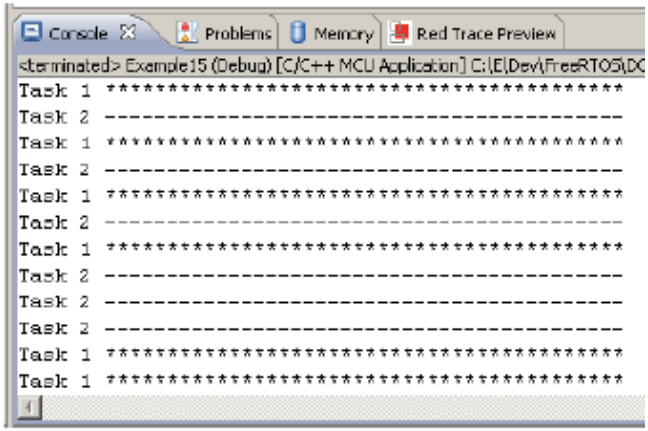
# Mutexes (and Binary Semaphores)

3 - Task 2 attempts to take the mutex, but the mutex is still held by Task 1 so Task 2 enters the Blocked state, allowing Task 1 to execute again.

2 - Task 1 takes the mutex and starts to write out its string. Before the entire string has been output Task 1 is preempted by the higher priority Task 2.

5 - Task 2 writes out its string, gives back the semaphore, then enters the Blocked state to wait for the next execution time. This allows Task 1 to run again - Task 1 also enters the Blocked state to wait for its next execution time leaving only the Idle task to run.




1 - The delay period for Task 1 expires so Task 1 pre-empts the idle task.

4 - Task 1 completes writing out its string, and gives back the mutex - causing Task 2 to exit the Blocked state. Task 2 preempts Task 1 again.

**Figure 37. A possible sequence of execution for Example 15**

## *Mutexes (and Binary Semaphores)*
## *Priority Inversion*

- The higher priority Task 2 having to wait for the lower priority Task 1 to give up control of the mutex.
- A higher priority task being delayed by a lower priority task in this manner is called ‘priority inversion’.
- This undesirable behavior would be exaggerated further if a medium priority task started to execute while the high priority task was waiting for the semaphore—the result would be a high priority task waiting for a low priority task without the low priority task even being able to execute.
- Priority inversion can be a significant problem, but in small embedded systems it can often be avoided at system design time, by considering how resources are accessed.

# Mutexes (and Binary Semaphores)

## Priority Inversion

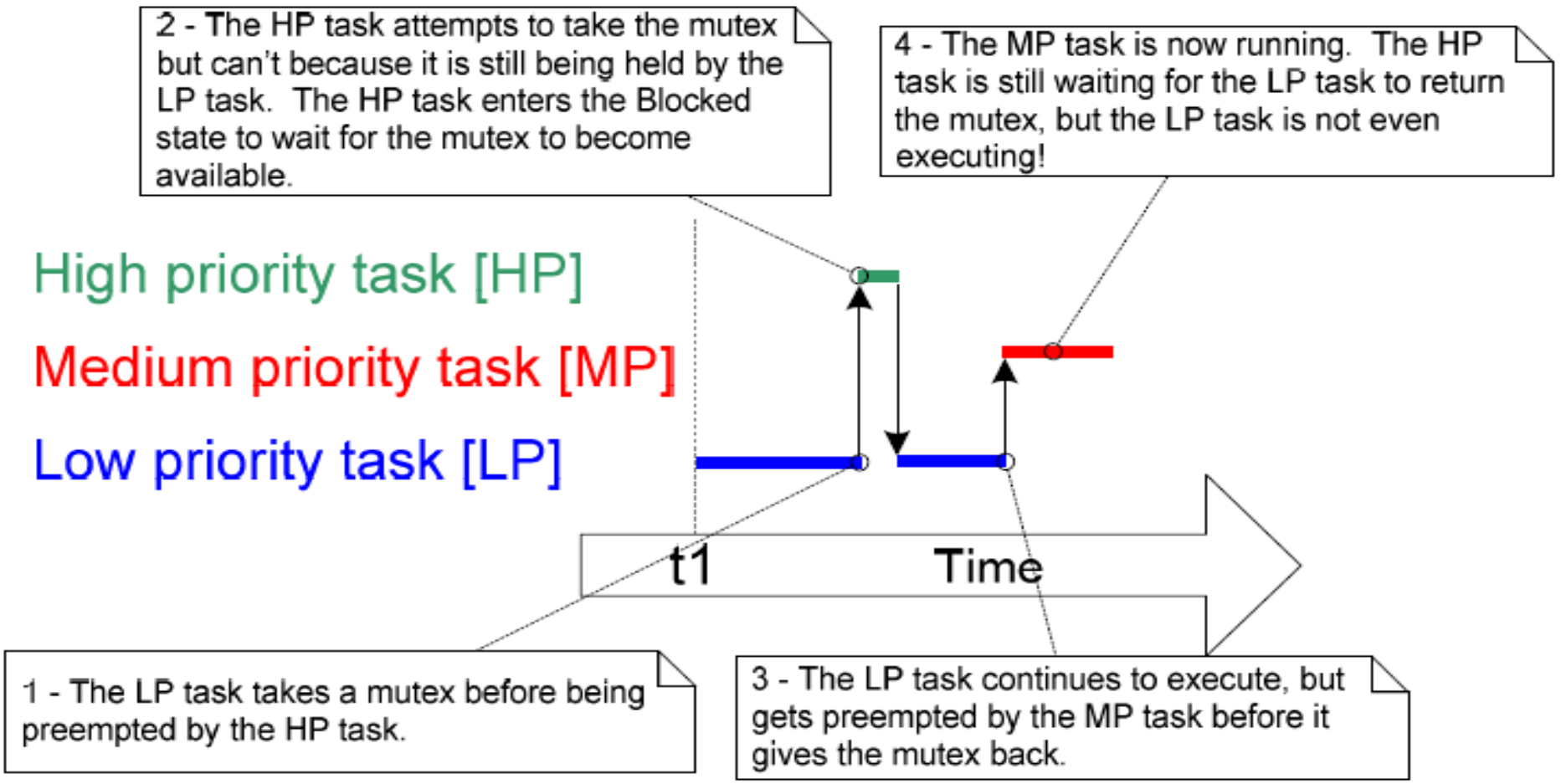



**Figure 38. A worst case priority inversion scenario**

# Mutexes (and Binary Semaphores)
## Priority Inheritance

- FreeRTOS mutexes and binary semaphores are very similar “BUT”
- Mutexes include a basic ‘priority inheritance’ mechanism
- Binary semaphores do not.
- Priority inheritance is a scheme that minimizes the negative effects of priority inversion but does not ‘fix’ priority inversion
- Priority inheritance works by temporarily raising the priority of the mutex holder to that of the highest priority task that is attempting to obtain the same mutex.
- The low priority task that holds the mutex ‘inherits’ the priority of the task waiting for the mutex.
- The priority of the mutex holder is reset automatically to its original value when it gives the mutex back.

# Mutexes (and Binary Semaphores)
## Priority Inheritance

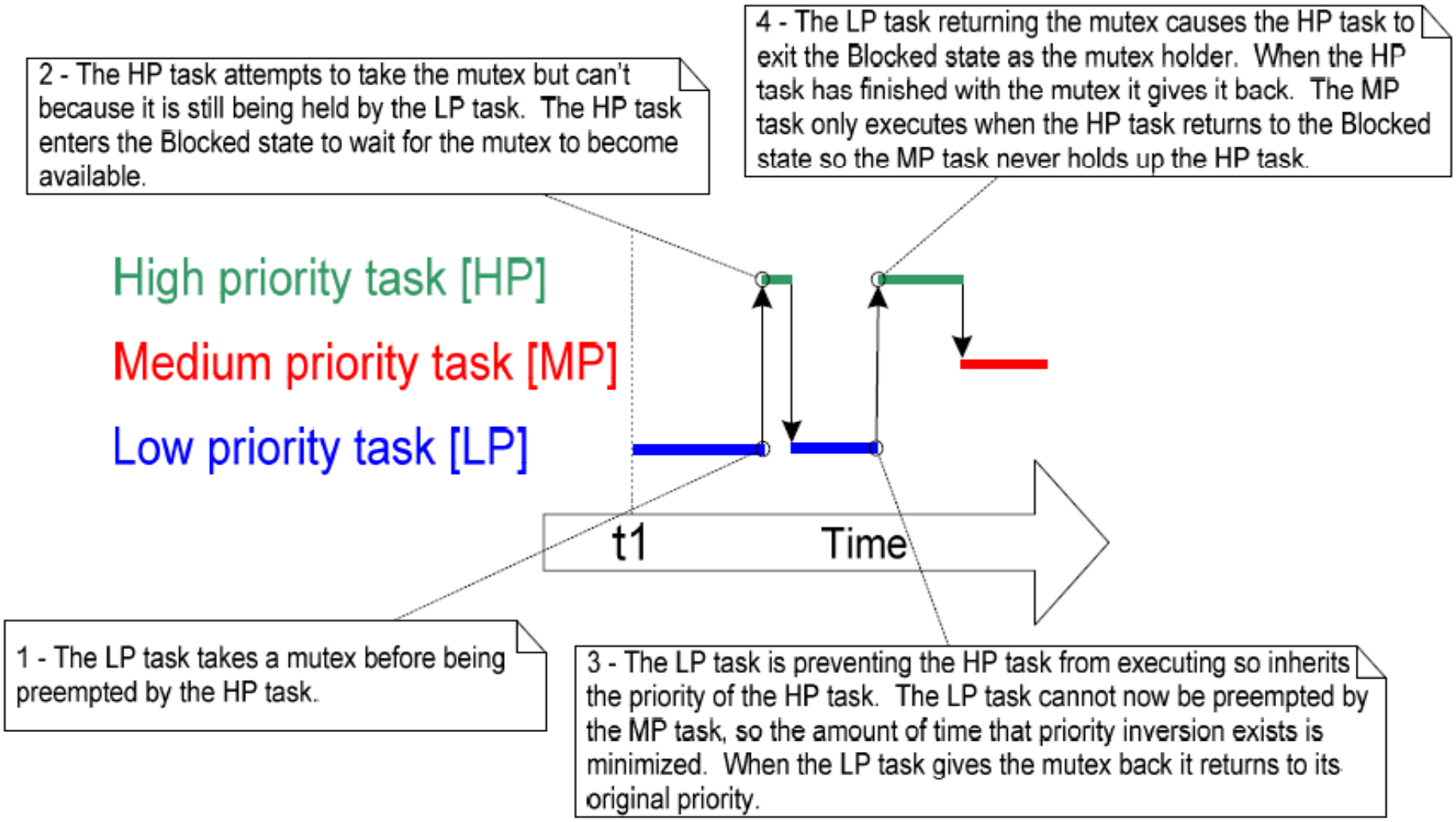



**Figure 39. Priority inheritance minimizing the effect of priority inversion**

# Mutexes (and Binary Semaphores) Deadlock (or Deadly Embrace)

- 'Deadlock' ('deadly embrace') is another potential pitfall that can occur when using mutexes for mutual exclusion.

- Deadlock occurs when two tasks cannot proceed because they are both waiting for a resource that is held by the other.

- Consider the following scenario where Task A and Task B both need to acquire mutex X and mutex Y in order to perform an action:
  1. Task A executes and successfully takes mutex X.
  2. Task A is pre-empted by Task B.
  3. Task B successfully takes mutex Y before attempting to also take mutex X—but mutex X is held by Task A, so is not available to Task B. Task B opts to enter the Blocked state to wait for mutex X to be released.
  4. Task A continues executing. It attempts to take mutex Y—but mutex Y is held by Task B, so is not available to Task A. Task A opts to enter the Blocked state to wait for mutex Y to be released. At the end of this scenario, Task A is waiting for a mutex held by Task B, and Task B is waiting for a mutex held by Task A.


# Real Time Operating System
# "FreeRTOS"
# Memory Management

***Sherif Hammad***

Using the FreeRTOS Real Time Kernel - a Practical Guide - Cortex M3 Edition (FreeRTOS Tutorial Books) by Richard Barry


# *Agenda*

- **Memory Management problems**
- **Memory Management solutions**
- **FreeRTOS 3 implementations of memory management algorithms**


## *Problem & FreeRTOS Solutions*

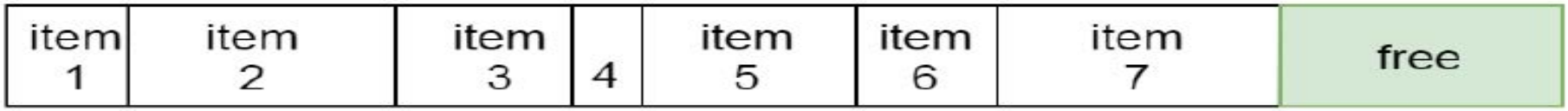
**The kernel has to allocate RAM dynamically each time a task, queue, or semaphore is created. The standard malloc() and free() library functions can be used, but they may not be suitable or appropriate for one or more of the following reasons:**

- **They are not always available on small embedded systems.**
- **Their implementation can be relatively large, taking up valuable code space.**
- **They are rarely thread-safe.**
- **They are not deterministic; the amount of time taken to execute the functions will differ from call to call.**
- **They can suffer from memory fragmentation.**
- **They can complicate the linker configuration.**

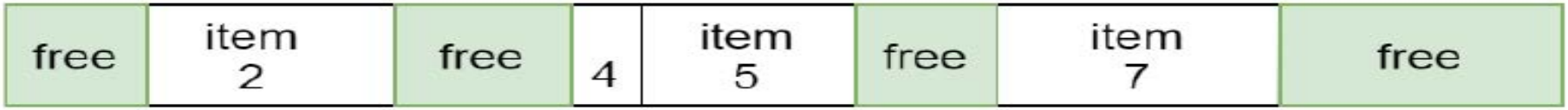
# Heap Fragmentation

empty Heap

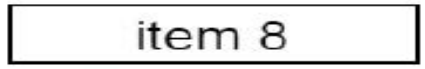

after allocations

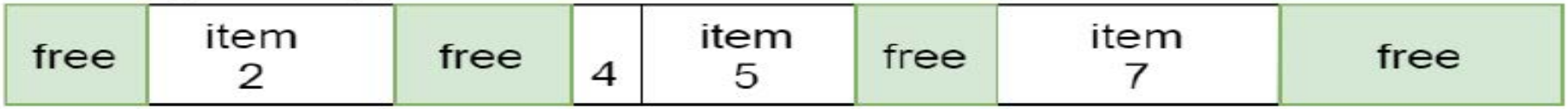


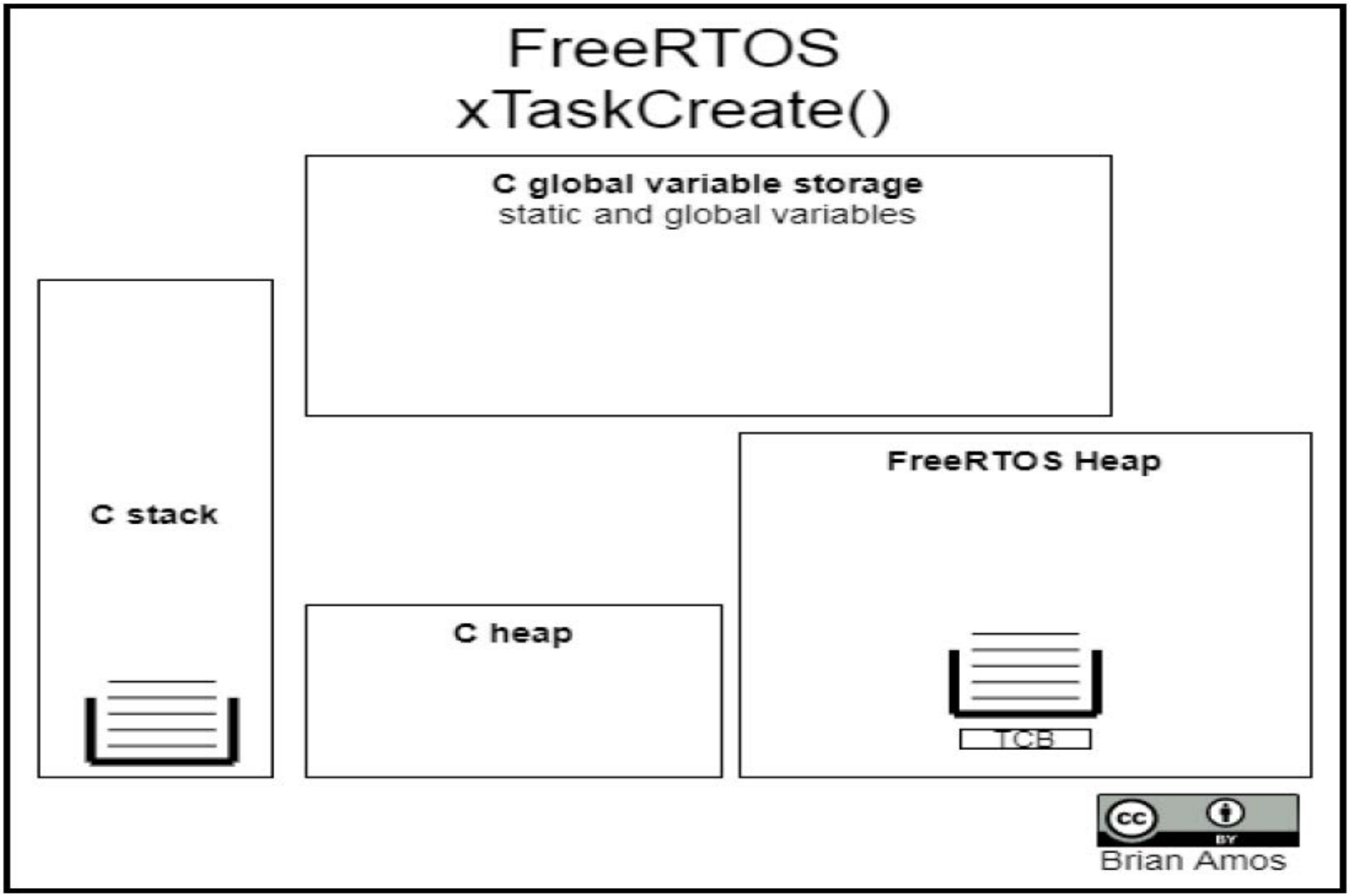

after free




allocation attempt (fails)
no contiguous space available for item 8




FreeRTOS
xTaskCreate()
C global variable storage
static and global variables
C stack
C heap
FreeRTOS Heap
TCB
Brian Amos


# Dynamic Allocation Problem Solution

**Where/why is FreeRtos Solution code?**

- Different embedded systems have varying RAM allocation and timing requirements
- Single RAM allocation algorithm does not fit all applications.
- FreeRTOS treats memory allocation as part of the portable layer (as opposed to part of the core code base).
- This enables individual applications to provide their own specific implementations, when appropriate.

**What are FreeRTOS APIs?**

- When the kernel requires RAM, instead of calling **malloc()** directly it calls **pvPortMalloc()** with same prototype
- When RAM is being freed, instead of calling **free()** directly, the kernel calls **vPortFree()** with same prototype

**What are the three example implementations of pvPortMalloc() and vPortFree()?**

- heap_1.c, heap_2.c, and heap_3
- located in the FreeRTOS\Source\portable\MemMang directory

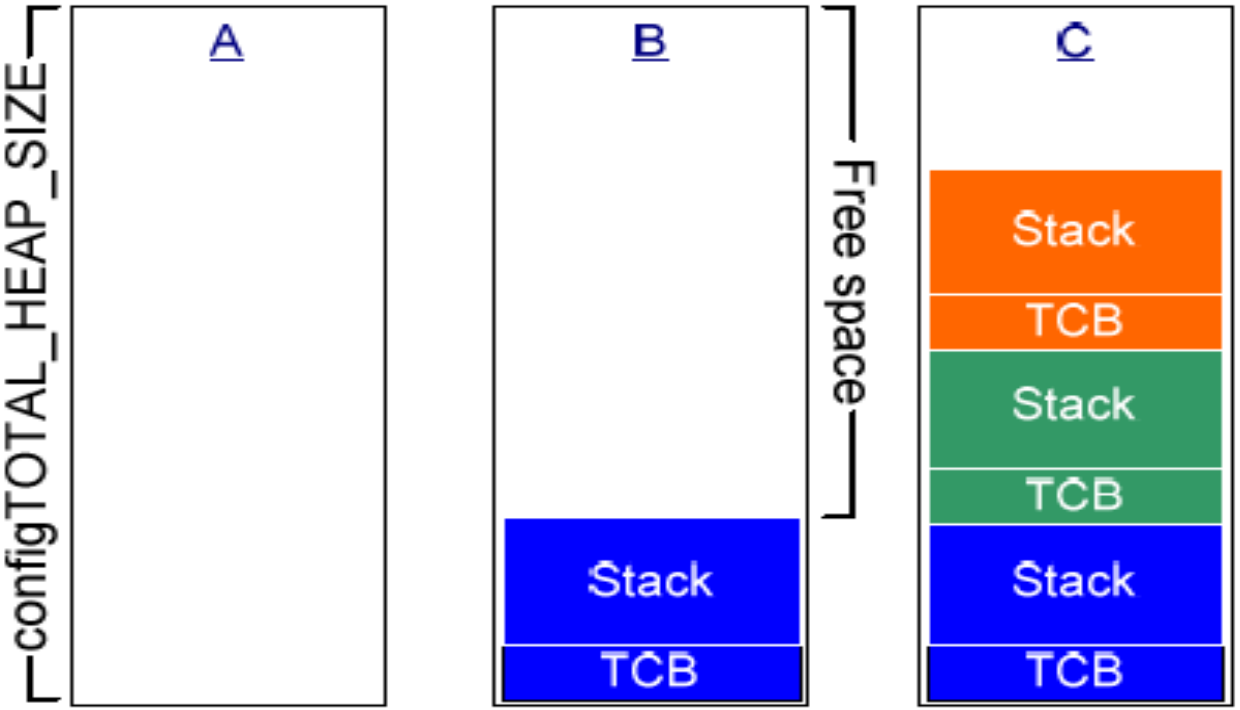
## System Designer Simple Use Case

- It is common for small embedded systems only to create tasks, queues, and semaphores before the scheduler has been started.
- Memory only gets dynamically allocated by the kernel before the application starts
- Memory remains allocated for the lifetime of the application.
- Designer when choosing allocation scheme does not have to consider any of the more complex issues such as determinism and fragmentation
- He/She can instead consider only attributes such as code size and simplicity.

# Heap_1.c

- Heap_1.c implements a very basic version of pvPortMalloc() and does not implement vPortFree().
- Applications that never delete a task, queue, or semaphore have the potential to use heap_1.
- Heap_1 is always deterministic.
- The allocation scheme subdivides a simple array into smaller blocks as calls to pvPortMalloc() are made.
- The array is the FreeRTOS heap.
- The total size (in bytes) of the array is set by the definition configTOTAL_HEAP_SIZE within FreeRTOSConfig.h.
- Defining a large array in this manner can make the application appear to consume a lot of RAM—even before any of the array has been assigned.
- Each created task requires a task control block (TCB) and a stack to be allocated from the heap.

# Heap_1.c




RAM being allocated within the array each time a task is created

- A shows the array before any tasks have been created—the entire array is free.
- B shows the array after one task has been created.
- C shows the array after three tasks have been created.

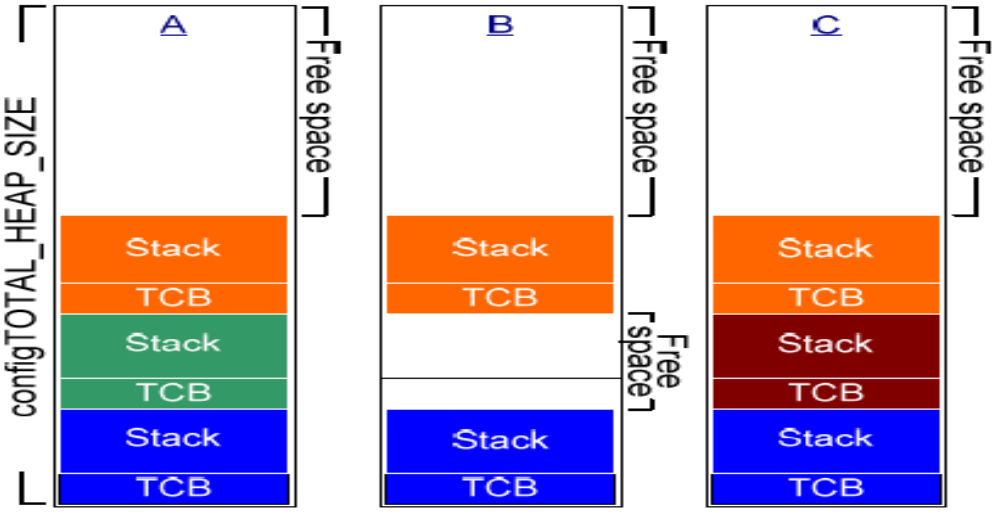
# Heap_2.c

- Heap_2.c also uses a static simple array dimensioned by configTOTAL_HEAP_SIZE
- It uses a best fit algorithm to allocate memory
- Unlike heap_1, it does allow memory to be freed.
- The best fit algorithm ensures that pvPortMalloc() uses the free block of memory that is closest in size to the number of bytes requested.

**For example, consider the scenario where:**

- The heap contains three blocks of free memory that are 5 bytes, 25 bytes, and 100 bytes, respectively. pvPortMalloc() is called to request 20 bytes of RAM.
- The smallest free block of RAM into which the requested number of bytes will fit is the 25-byte block, so pvPortMalloc() splits the 25-byte block into one block of 20 bytes and one block of 5 bytes
- Before returning a pointer to the 20-byte block. The new 5-byte block remains available to future calls to pvPortMalloc().
- Heap_2.c does not combine adjacent free blocks into a single larger block, so it can suffer from fragmentation.
- However, fragmentation is not an issue if the blocks being allocated and subsequently freed are always the same size. (coming example)
- Heap_2.c is suitable for an application that creates and deletes tasks repeatedly, provided the size of the stack allocated to the created tasks does not change.

# Heap_2.c




RAM being allocated from the array as tasks are created and deleted

- A shows the array after three tasks have been created. A large free block remains at the top of the array.
- B shows the array after one of the tasks has been deleted. The large free block at the top of the array remains. There are now also two smaller free blocks that were previously allocated to the TCB and stack of the deleted task.
- C shows the situation after another task has been created. Creating the task has resulted in two calls to pvPortMalloc(), one to allocate a new TCB and one to allocate the task stack. (The calls to pvPortMalloc() occur internally within the xTaskCreate() API function.)
- Every TCB is exactly the same size, so the best fit algorithm ensures that the block of RAM previously allocated to the TCB of the deleted task is reused to allocate the TCB of the new task.
- The size of the stack allocated to the newly created task is identical to that allocated to the previously deleted task, so the best fit algorithm ensures that the block of RAM previously allocated to the stack of the deleted task is reused to allocate the stack of the new task.
- The larger unallocated block at the top of the array remains untouched.
- Heap_2.c is not deterministic but is more efficient than most standard library implementations of malloc() and free().


# Heap_1.c & Heap_2.c

## The xPortGetFreeHeapSize() API Function

xPortGetFreeHeapSize() is available only when heap_1.c or heap_2.c is being used. It provides a simple method of optimizing the heap size by returning the current number of unallocated bytes. For example, if xPortGetFreeHeapSize() returns 2000 after all the required tasks, queues, and semaphores have been created, then configTOTAL_HEAP_SIZE can be reduced by 2000.

```
size_t xPortGetFreeHeapSize( void );
```

Listing 77. The xPortGetFreeHeapSize() API function prototype

# Heap_3.c

- Heap_3.c uses the standard library malloc() and free() function but makes the calls thread safe by temporarily suspending the scheduler.
- The size of the heap is not affected by configTOTAL_HEAP_SIZE; instead, it is defined by the linker configuration.

```
void *pvPortMalloc( size_t xWantedSize )
{
void *pvReturn;

    vTaskSuspendAll();
    {
        pvReturn = malloc( xWantedSize );
    }
    xTaskResumeAll();

    return pvReturn;
}

void vPortFree( void *pv )
{
    if( pv != NULL )
    {
        vTaskSuspendAll();
        {
            free( pv );
        }
        xTaskResumeAll();
    }
}
```

Listing 76. The heap_3.c implementation