RTOS
Real Time Operating System

Session 1 : Introduction to RTOS

- ❑ design patterns.
- ❑ foreground/background systems.
- ❑ Real time systems
- ❑ types of real time systems.
- ❑ multi-tasking.
- ❑ scheduling algorithms.

Session Benchmarking

Session Type : Theory & Practical
Labs : 3

Session 2 : FreeRTOS

- ❑ downloading FreeRTOS.
- ❑ Porting FreeRTOS to Tiva C and ATMEGA32.
- ❑ Tasks creation
- ❑ Task states
- ❑ Task control
- ❑ Task utils

Session Benchmarking

Session Type : Theory & Practical
Labs : 5

Session 3 : Task communication

- ❑ shared resource problem.
- ❑ race condition.
- ❑ reentrancy.
- ❑ Critical sections.
- ❑ queue management.
- ❑ using queues.

Session Benchmarking

Session Type : Theory & Practical
Labs : 5

Session 4 : Task synchronization

- ❑ semaphores.
- ❑ binary semaphores
- ❑ counting semaphores.
- ❑ priority inversion and deadlocks.
- ❑ mutex.
- ❑ priority inheritance.

Session Benchmarking

Session Type : Theory & Practical
Labs : 3

AMIT
Beyond Education

Before asking Why and What?!!!

LET'S CODE IT!

Toggle the RED Led on your Tiva C every 200 ms

Easy ha!

AMIT
Beyond Education

Before asking Why and What?!!!

LET'S CODE IT!

Toggle the RED Led on your Tiva C every 200 ms and the green led every 350 ms

Still easy?

AMIT
Beyond Education

Before asking Why and What?!!!

LET'S CODE IT!

Toggle the RED Led on your Tiva C every 200 ms, the green led every 350 ms and blue led every 500 ms

Getting complicated!

AMIT
Beyond Education

Design patterns

- Static design

  the decision to call a particular block of code is done at compile time.

- Dynamic design

  the decision to call a particular block of code is done in run time.

Design patterns

- In every embedded system design, we should start with static first then move to dynamic.
- One of the best solution for this exercise is to divide the project into tasks. Each task is just a void function that conations functional functions that have the same time constrains.

  - Task1 (comes every 200msec) --> toggle the RED LED
  - Task2 (comes every 350msec) --> toggles the GREEN LED
  - Task3 (comes every 500msec) --> toggles the BLUE LED

Design patterns

- To calculate the time for each task, we can use only one timer for the whole system. This timer actives each task's flag to be ready to work by counting how much time passed now. Counting process is done every system tick, which is the value of “biggest common divider” for all tasks. It's 50 msec here in our example. Here is how it should be implemented:

Design patterns

```
ISR (/*timer OV every 50 msec*/)
            {
            //T1_counter++;   T2_counter++;   T3_counter++;
            //if   (T1_counter = = 4)   --->   T1_flag =1; T1_counter = 0
            //if   (T2_counter = = 7)   --->   T2_flag =1; T2_counter = 0
            //if   (T3_counter = = 10) --->   T3_flag =1; T3_counter = 0
            }
 int main()
      {
    // loop
      {
      //  if (T1_flag) --> call Task1; T1_flag = 0;
        //if (T2_flag) --> call Task2; T2_flag = 0;
        //if (T3_flag) --> call Task3; T3_flag = 0;
     }
}
```

AMIT
Beyond Education

Design patterns

- This method is called a **scheduler**. It's one of the solutions that is used to solve the time constrains problem. it's almost a generic solution. And used easily in this example without any problem even if we add more LEDs.

- If we have much more time constrains this can be called real time system.

Foreground/Background Systems

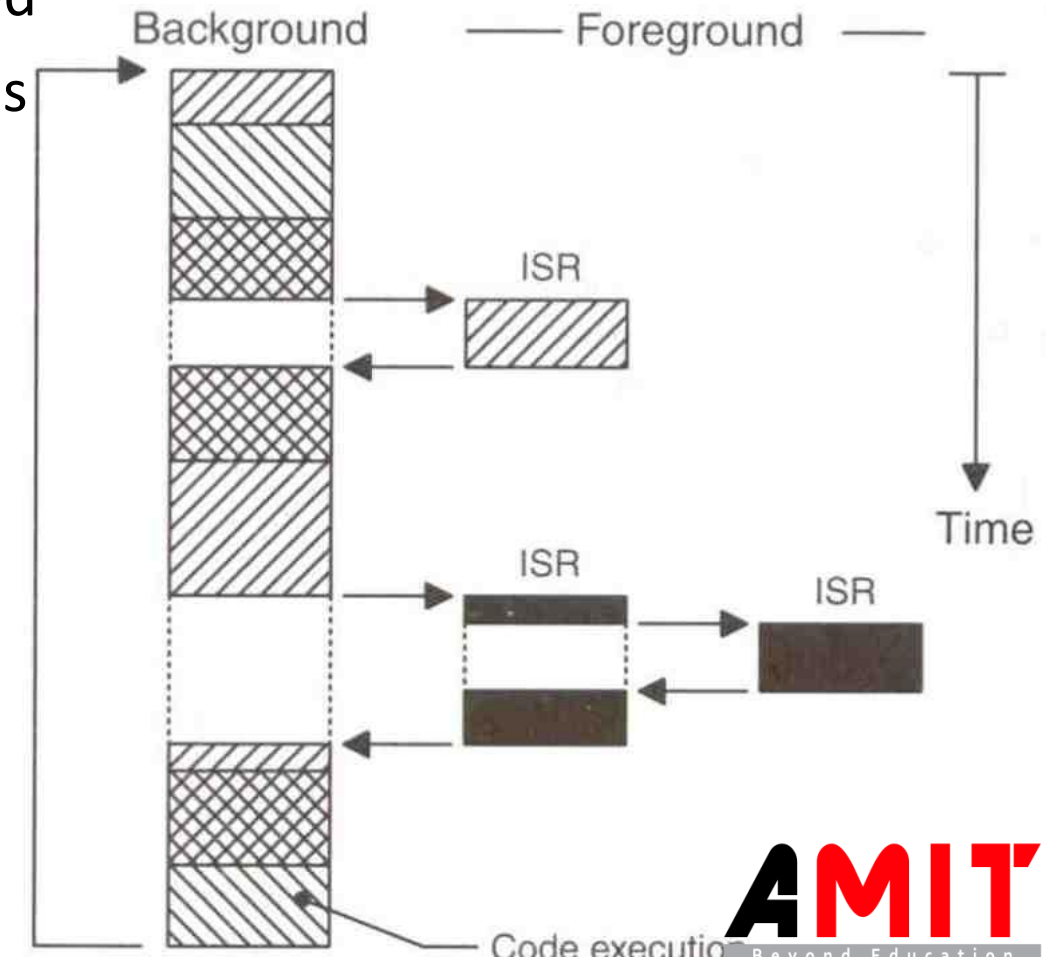
- Small systems of low complexity
- These systems are also called “super-loops”
- An application consists of an infinite loop of desired operations (background)
- Interrupt service routines (ISRs) handle asynchronous events (foreground)
- Critical operations must be performed by the ISRs to ensure the timing correctness
- Thus, ISRs tend to take longer than they should
- Task-Level Response
  - Information for a background module is not processed until the module gets its turn

Foreground/Background Systems

- The execution time of typical code is not constant
- If a code is modified, the timing of the loop is affected
- Most high-volume microcontroller-based applications

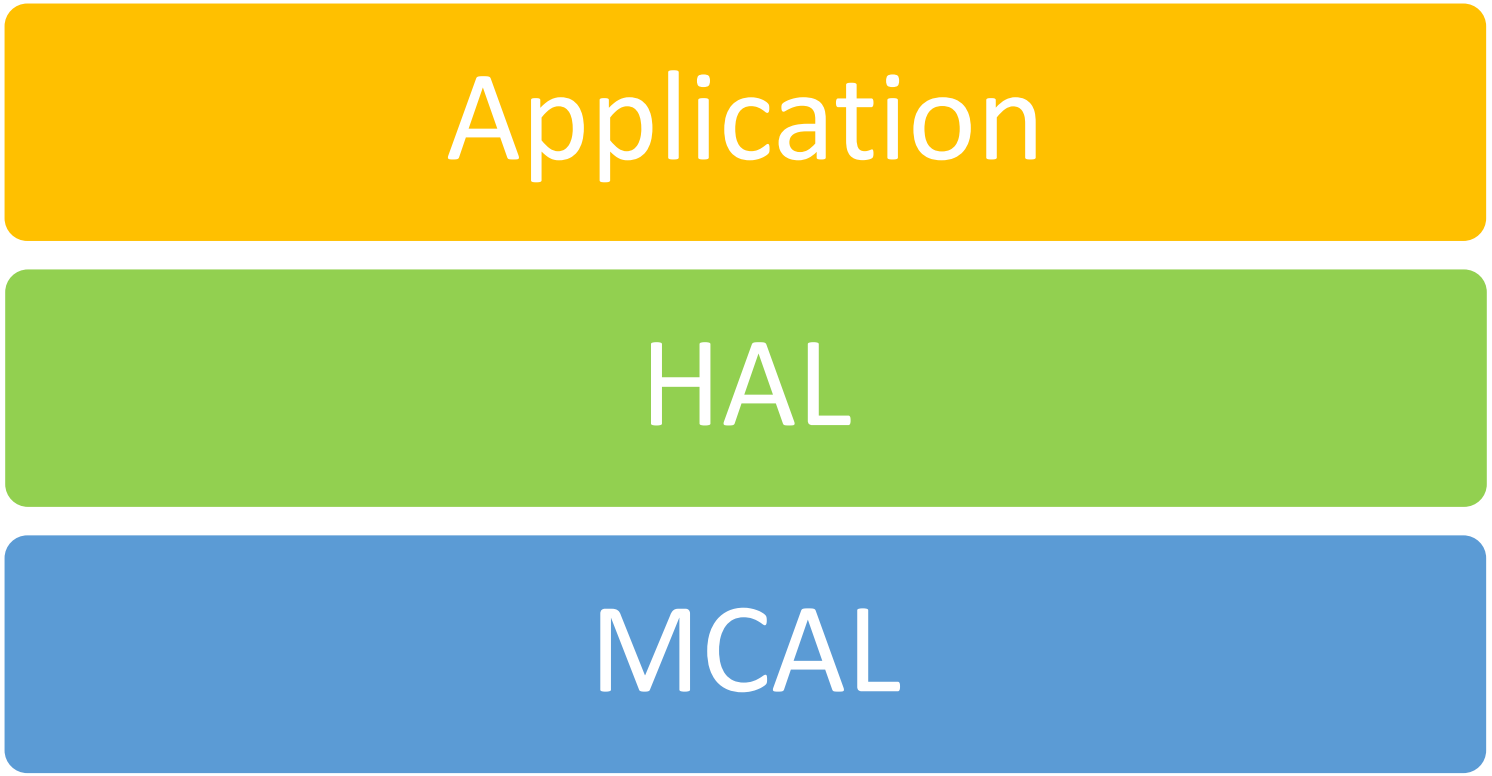
are F/B systems

  - Microwave ovens
  - Telephones
  - Toys
- From a power consumption point of view, it might be better to halt and perform all processing in ISRs



Foreground/Background Systems

A simple foreground/background system architecture would be like:



Real-Time Systems

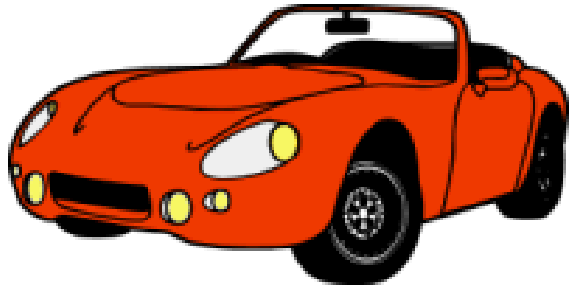
What are the Real-Time Systems?

Simple definition:

- Real-time systems are defined as those systems in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced.

Types Of Real Time Systems

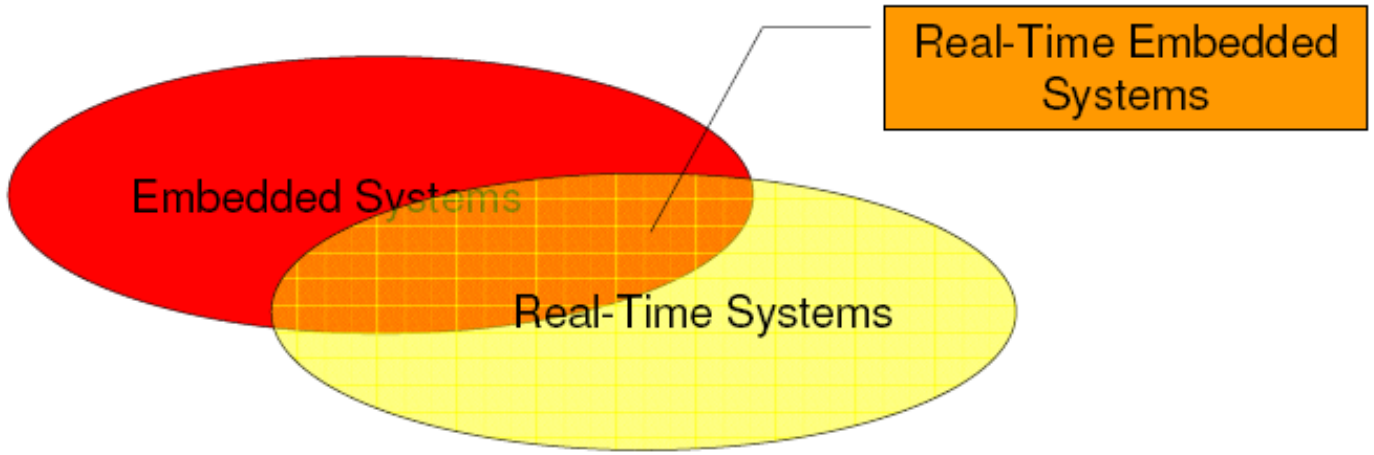
- **Hard real-time system:**
- Also known as an immediate real-time system.
  - The system that must operate within the confines of a stringent deadline. It is considered to have failed if it does not complete its function within the allowed time span.
  - Examples:
    - Car Air bag.
    - anti-lock Braking Systems(ABS).
    - aircraft control systems.



Types Of Real Time Systems (Cont.)

- **Soft real-time system:**
  - The system that can operate with the presence of latency at deadline.
  - It is not considered to have failed if it does not complete its function at deadline but that affects the quality of the output.
  - Examples:
    - Video Processing
    - Audio processing
    - Digital cameras
    - Mobile phones

Embedded Systems
Vs.
Real Time Systems



Misconceptions about Real Time Systems

- Real-time computing is equivalent to fast computing. X

  Truth=> Real-time computing is equivalent to **PREDICTABLE** computing.

- Real-time programming is assembly coding. X

  Truth=> It is better to **automate** (as much possible) real-time system design, instead relying on a clever hand-crafted code.

- "Real time" means performance engineering. X

  Truth => In real-time computing, timeliness is always more important than performance.

- Real-time systems function in a static environment. X

  Truth => Real-time systems consider systems in which the operating mode may change **dynamically**. On the other hand everything should be deterministic!

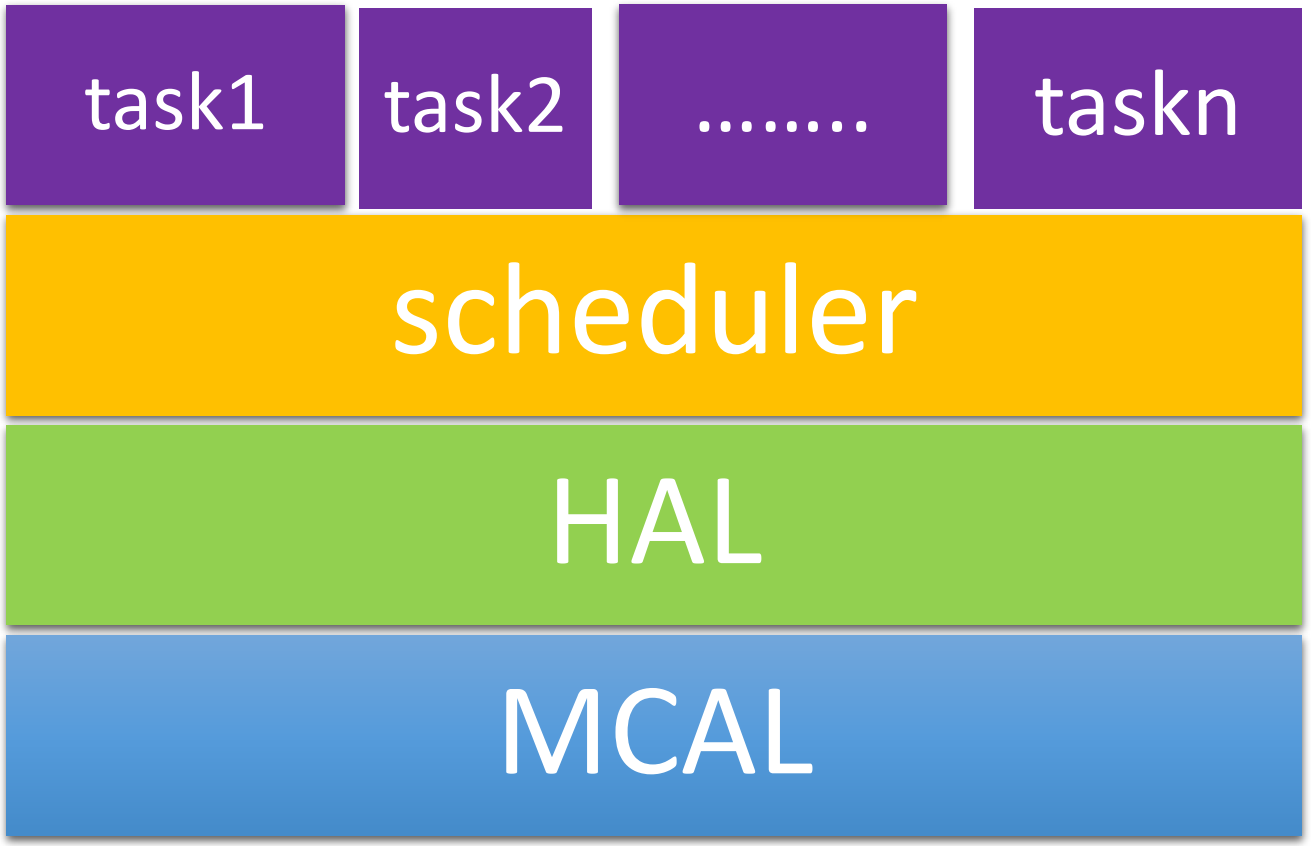
Task

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

```
void task(void)
{
        /*Some Initialization Code*/
        for(;;)
        {
                /*Task Code*/
        }
}
```

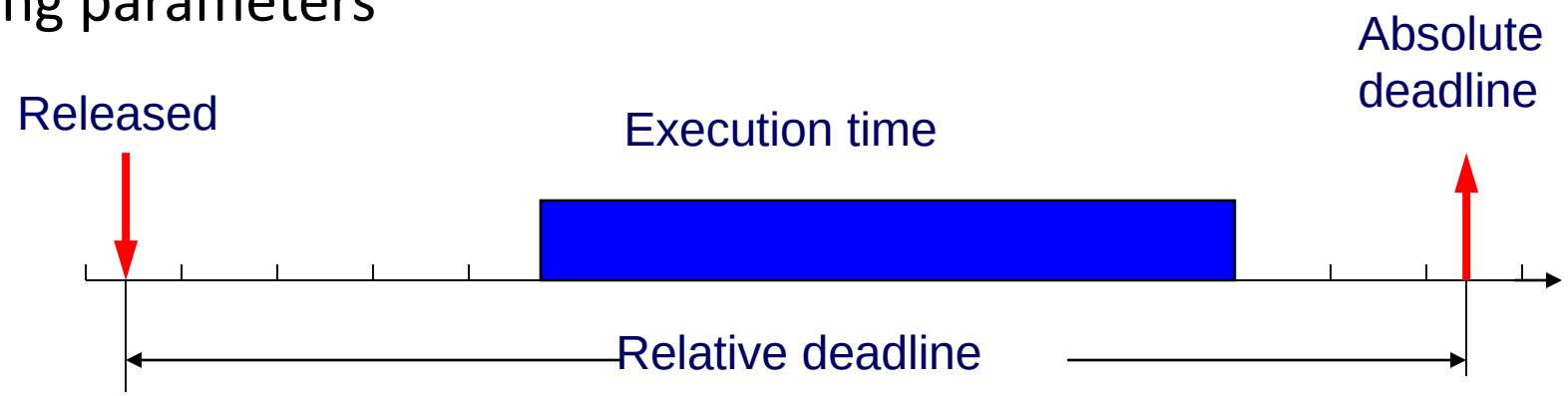
Multi-Tasking

After separating the application into tasks another layer has to be added to manage which task to be executed when based on the scheduling algorithm



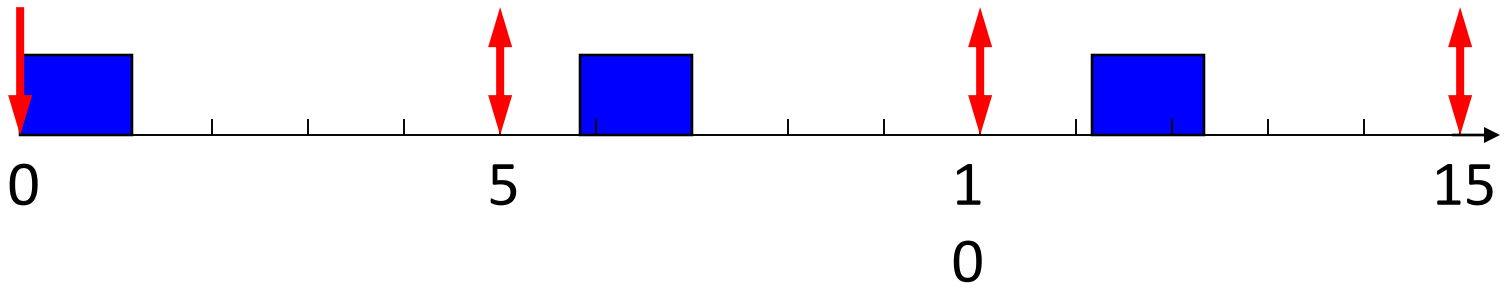
Real-Time Workload

- Job (unit of work)
  - a computation, a file read, a message transmission, etc
- Attributes
  - Resources required to make progress
  - Timing parameters



Real-Time Task

- Task : a sequence of similar jobs
  - Periodic task (p,e)
    - Its jobs repeat regularly
    - Period p = inter-release time ($0 < p$)
    - Execution time e = maximum execution time ($0 < e < p$)
    - Utilization U = e/p



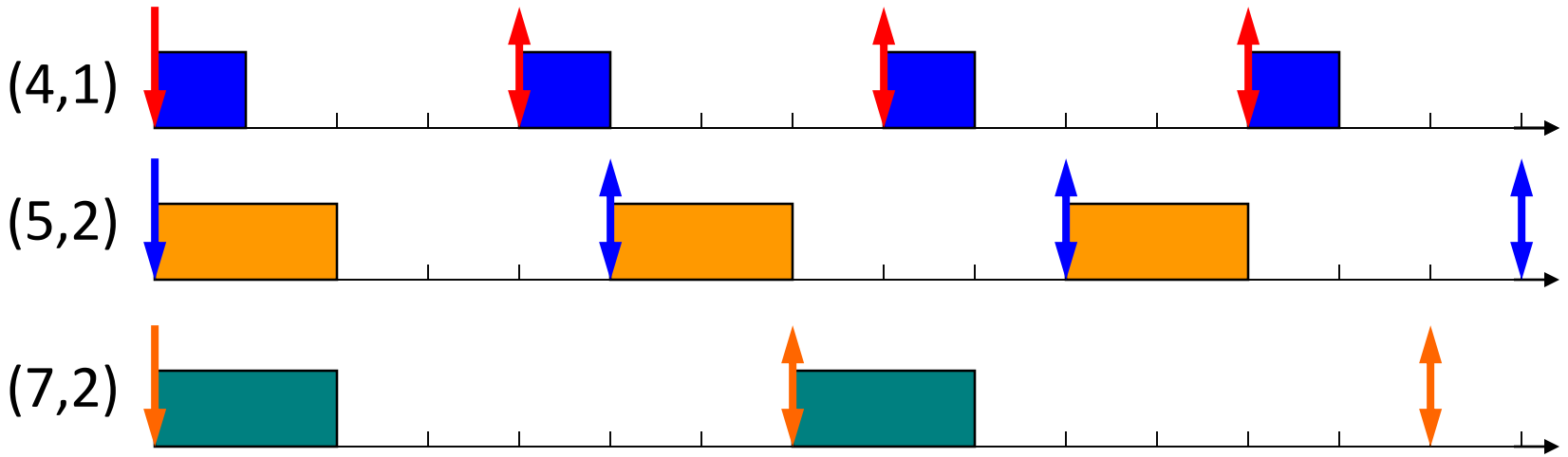
Back to Deadlines

- Hard deadline
  - Disastrous or very serious consequences may occur if the deadline is missed
  - Validation is essential : can all the deadlines be met, even under worst-case scenario?
  - Deterministic guarantees

- Soft deadline
  - Ideally, the deadline should be met for maximum performance. The performance degrades in case of deadline misses.
  - Best effort approaches / statistical guarantees

Schedulability

- Property indicating whether a real-time system (a set of real-time tasks) can meet their deadlines

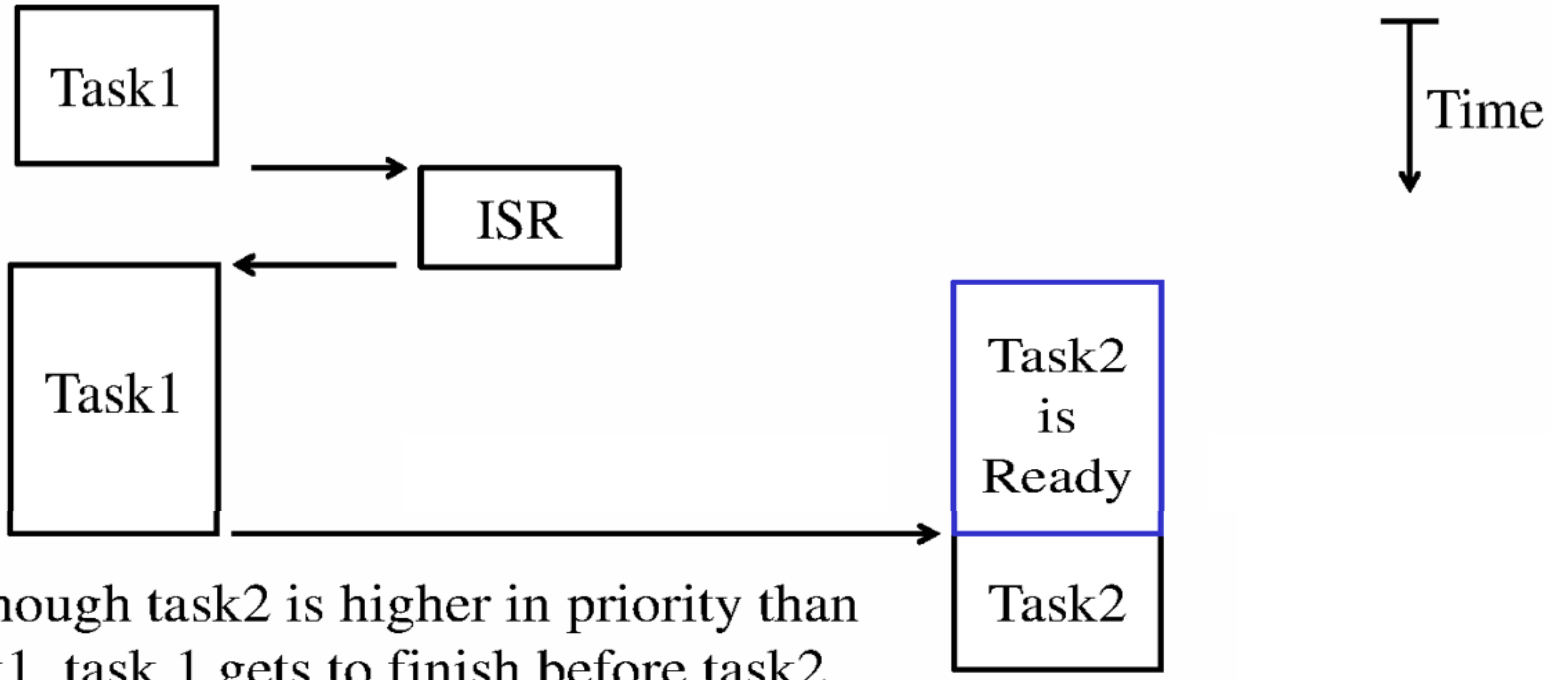


Scheduling Algorithms

- What is a scheduler?
  - It is part of the kernel which decides which task can run when.
  - There are many algorithms for scheduling & they can be categorized into two main categories.

Scheduling Algorithms

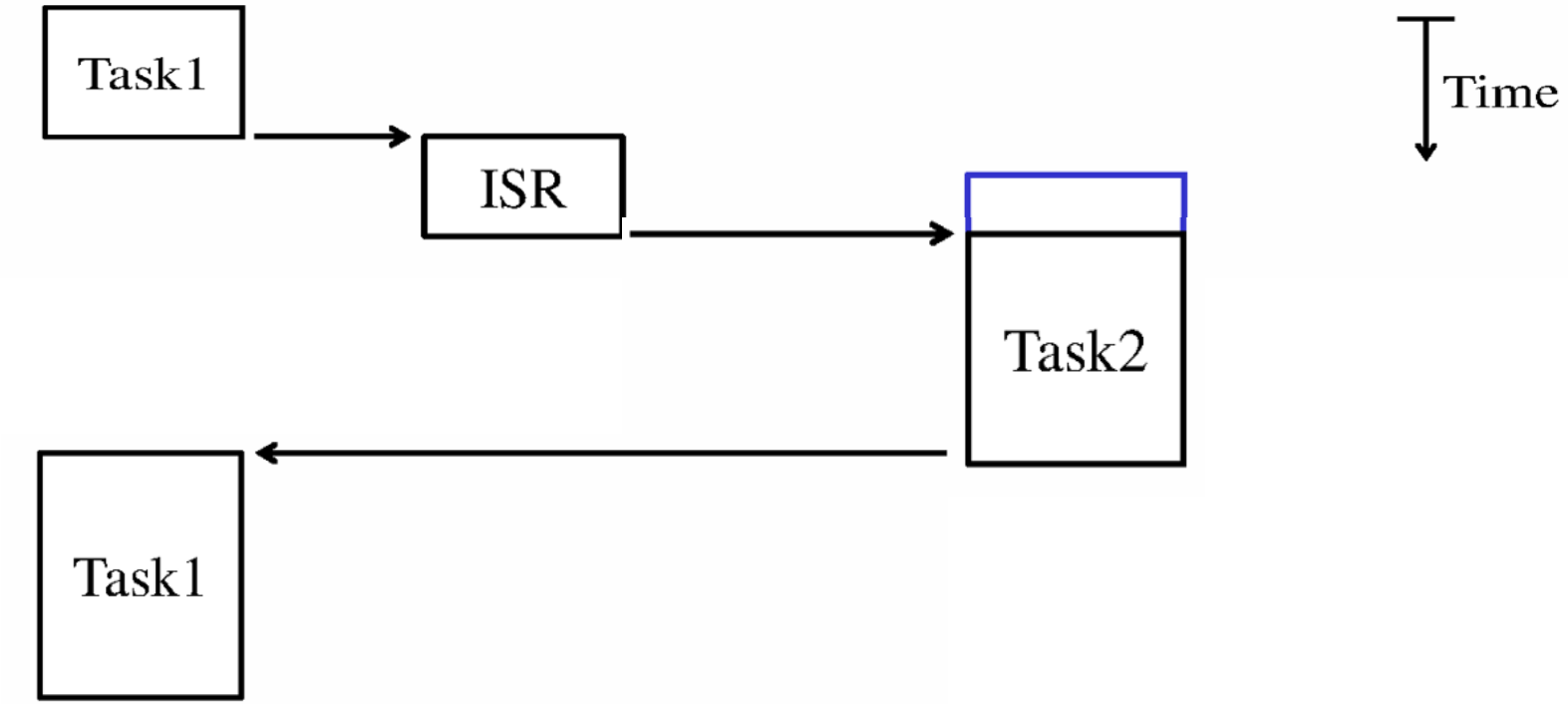
- Non-preemptive (Suppose it is priority based):



Although task2 is higher in priority than task1, task 1 gets to finish before task2 gets to start.

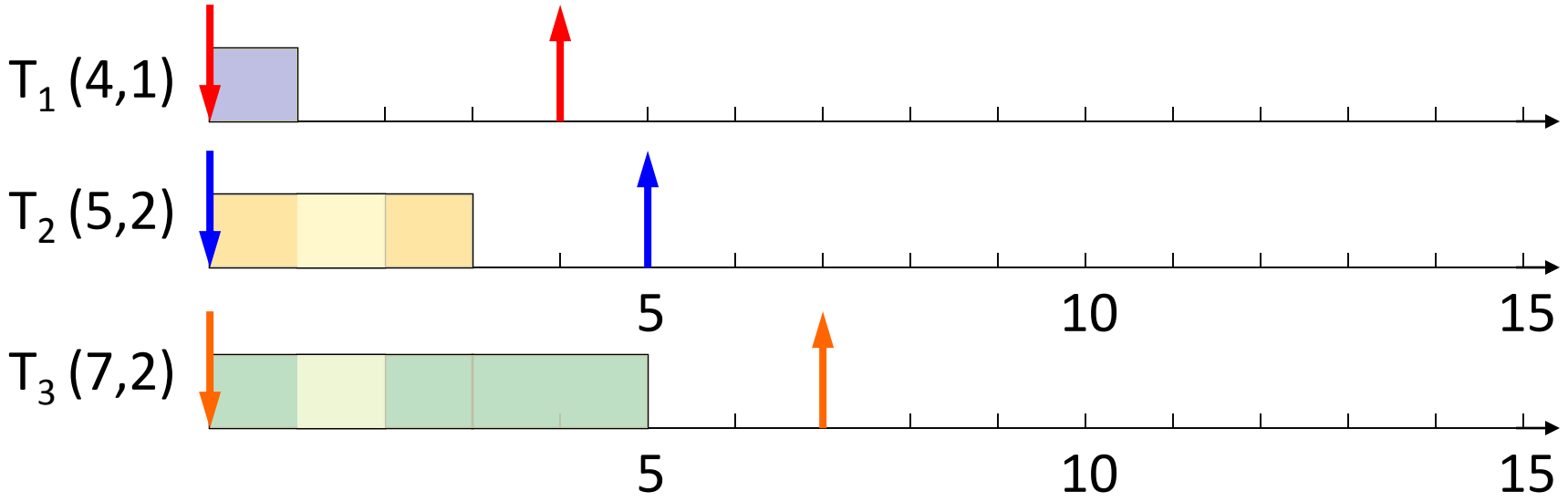
Scheduling Algorithms

- Preemptive (Suppose it is priority based):



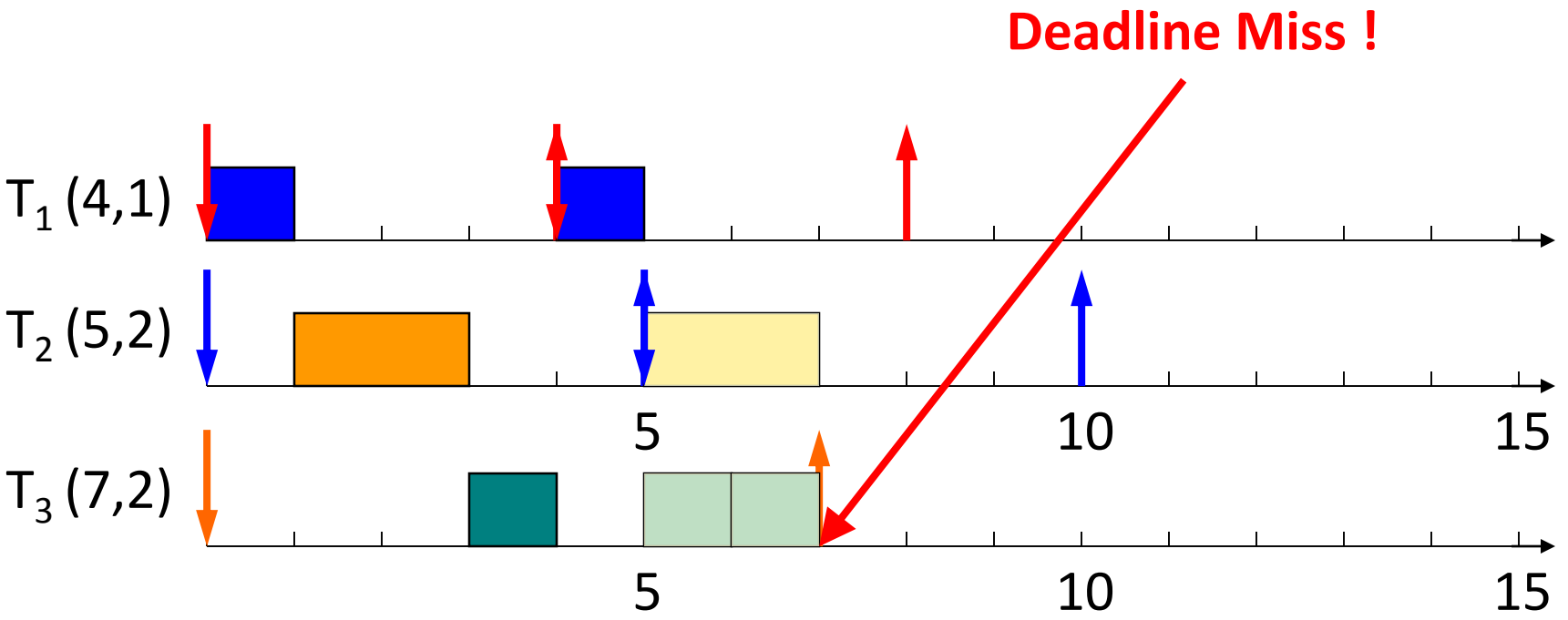
RM (Rate Monotonic)

- Optimal static-priority scheduling
- It assigns priority according to period
- A task with a shorter period has a higher priority
- Executes a job with the shortest period



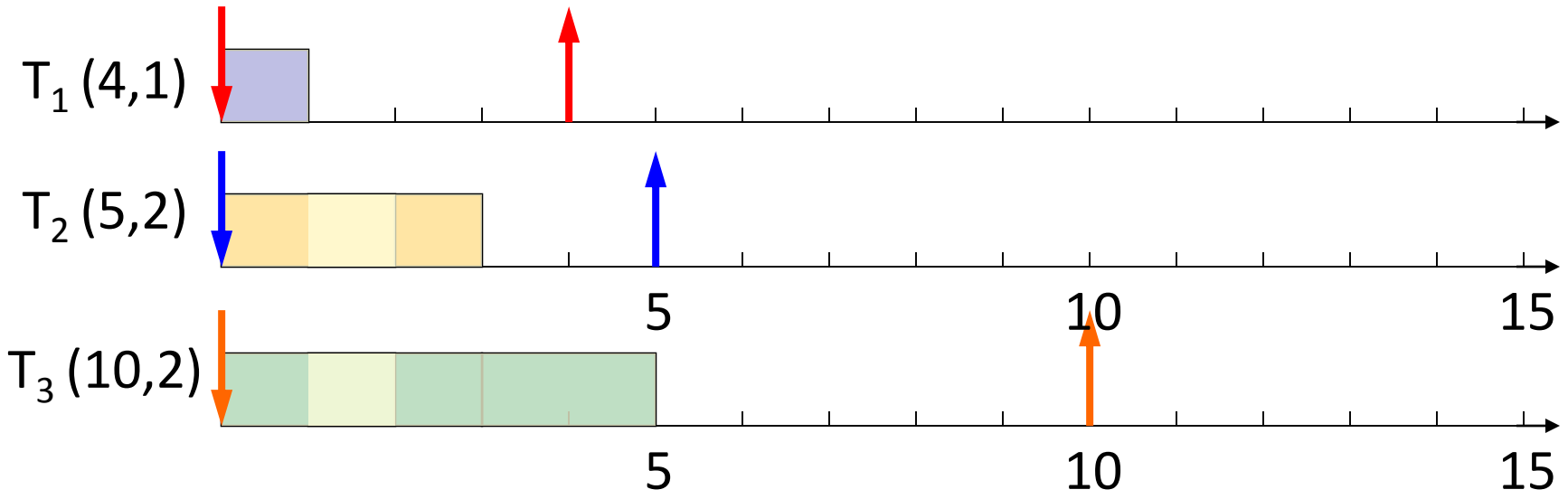
RM (Rate Monotonic)

- Executes a job with the shortest period



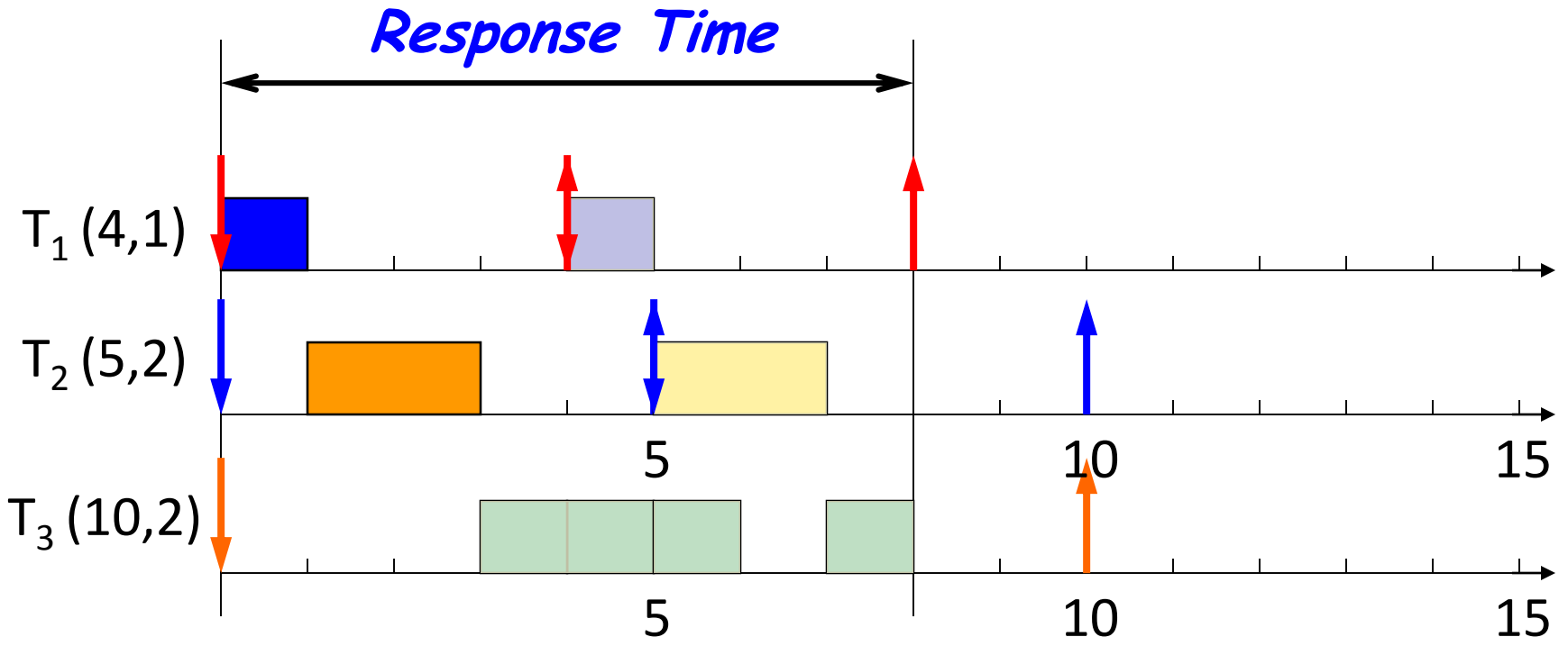
Response Time

- Response time
  - Duration from released time to finish time



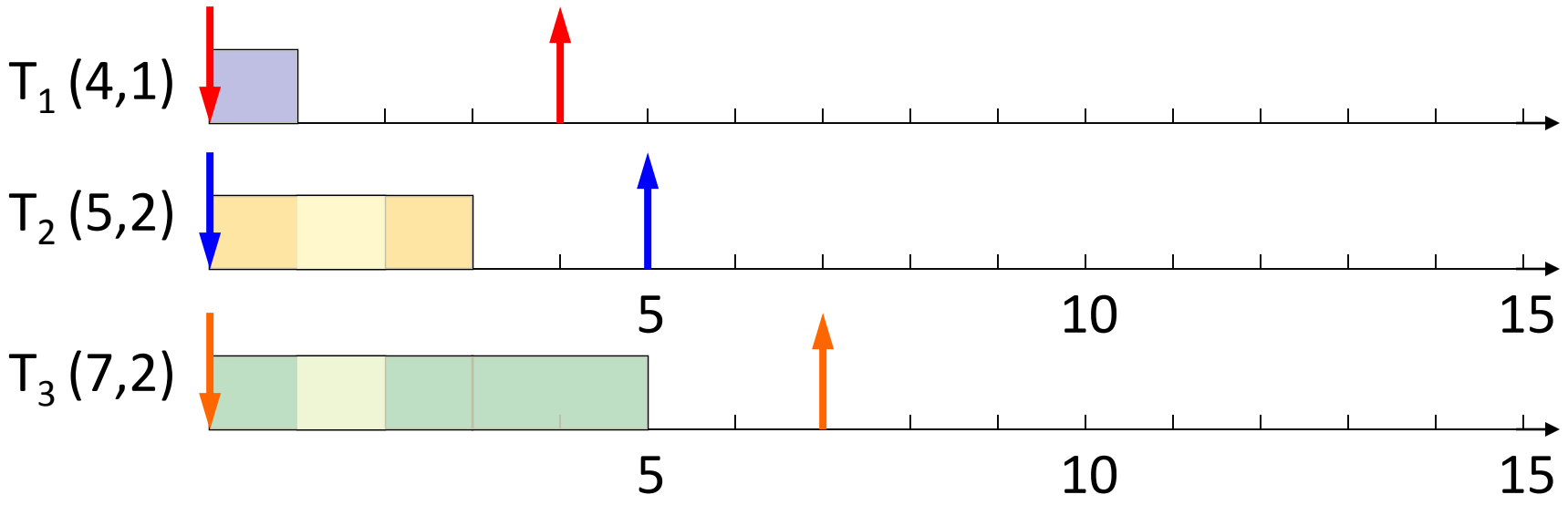
Response Time

- Response time
  - Duration from released time to finish time



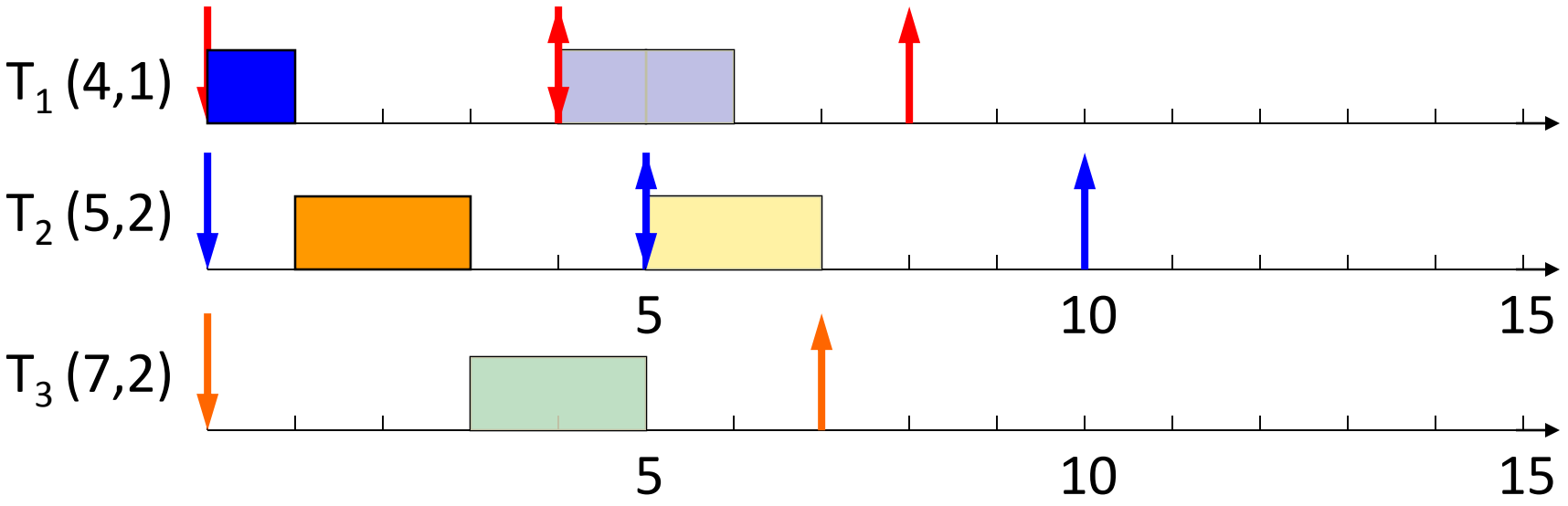
EDF (Earliest Deadline First)

- Optimal dynamic priority scheduling
- A task with a shorter deadline has a higher priority
- Executes a job with the earliest deadline



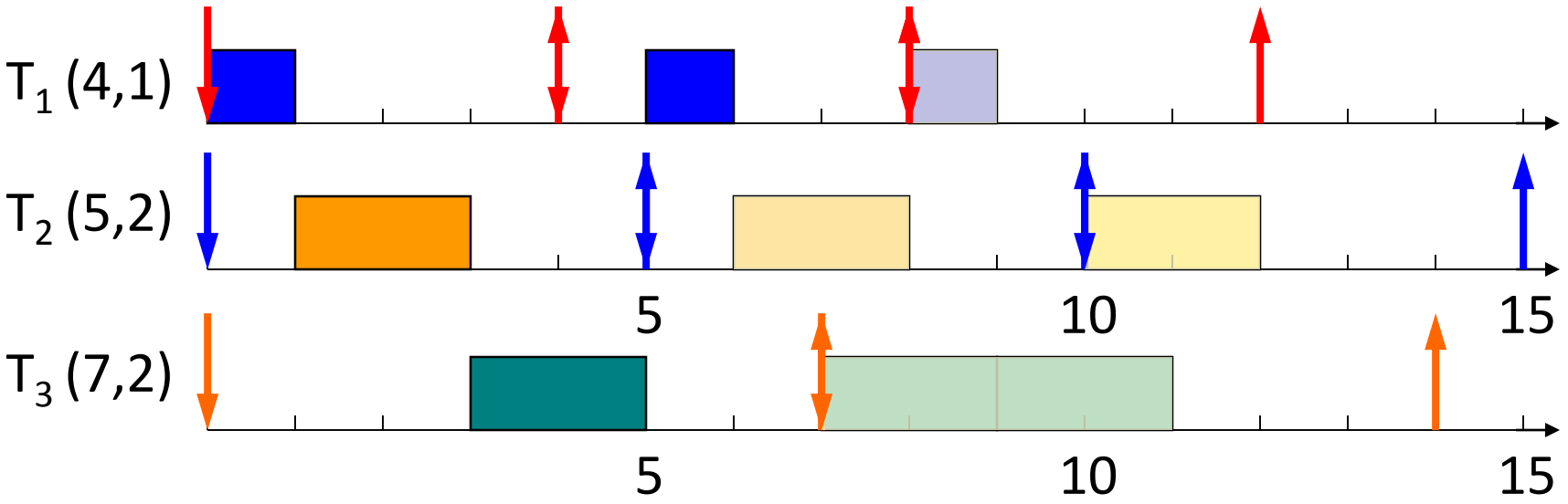
EDF (Earliest Deadline First)

- Executes a job with the earliest deadline



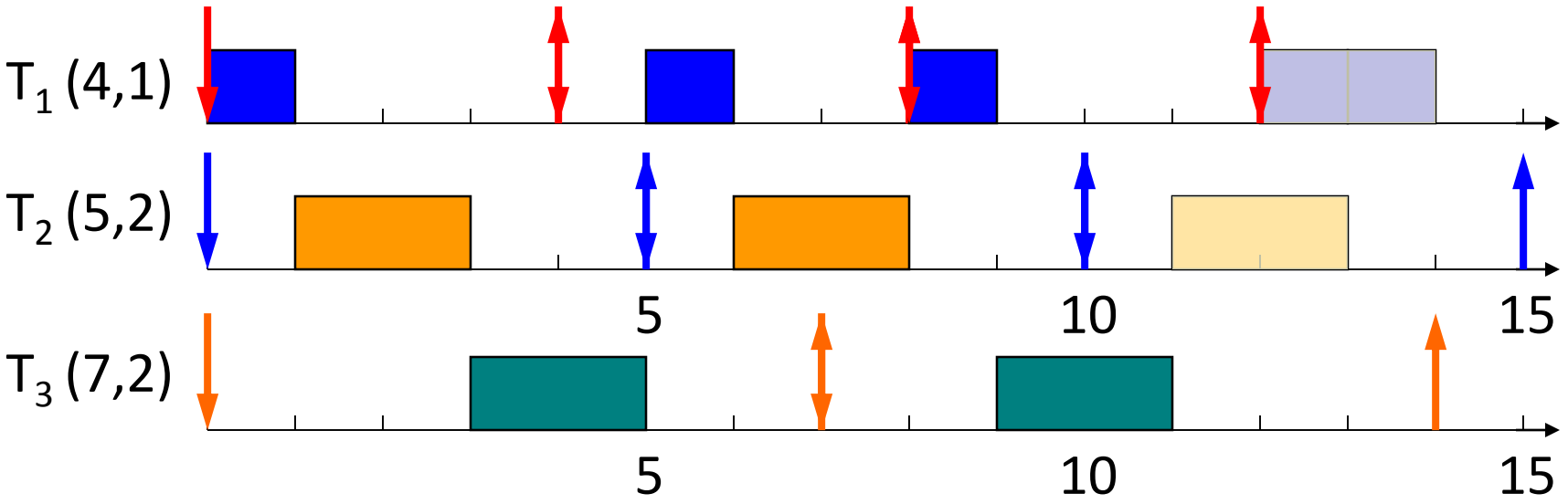
EDF (Earliest Deadline First)

- Executes a job with the earliest deadline



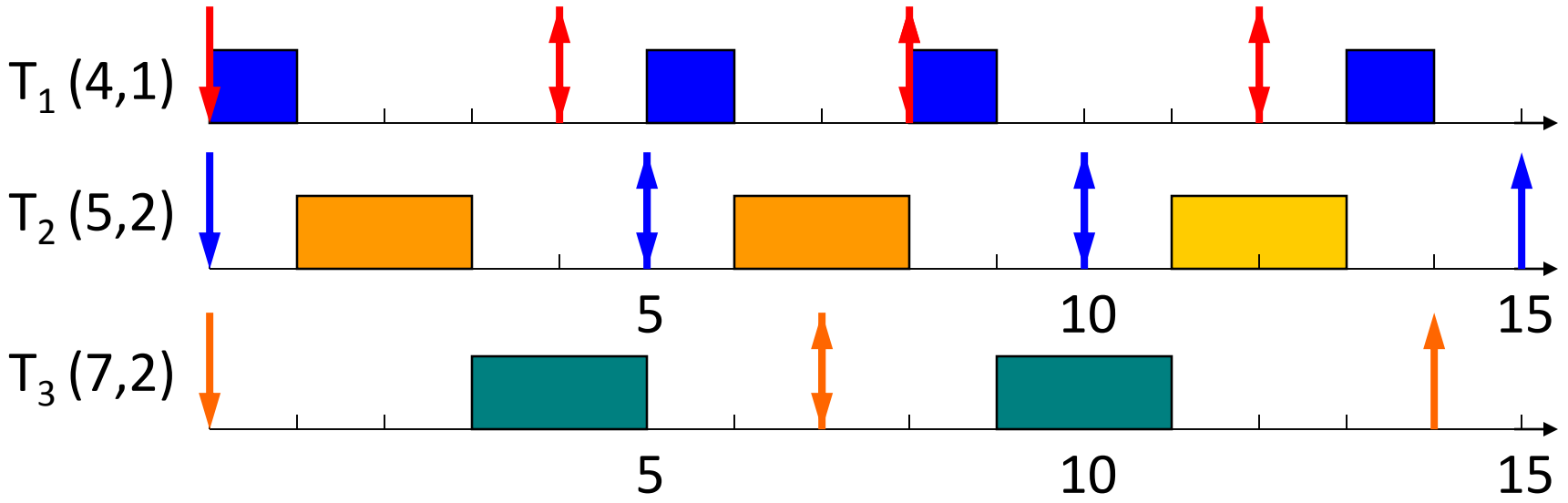
EDF (Earliest Deadline First)

- Executes a job with the earliest deadline



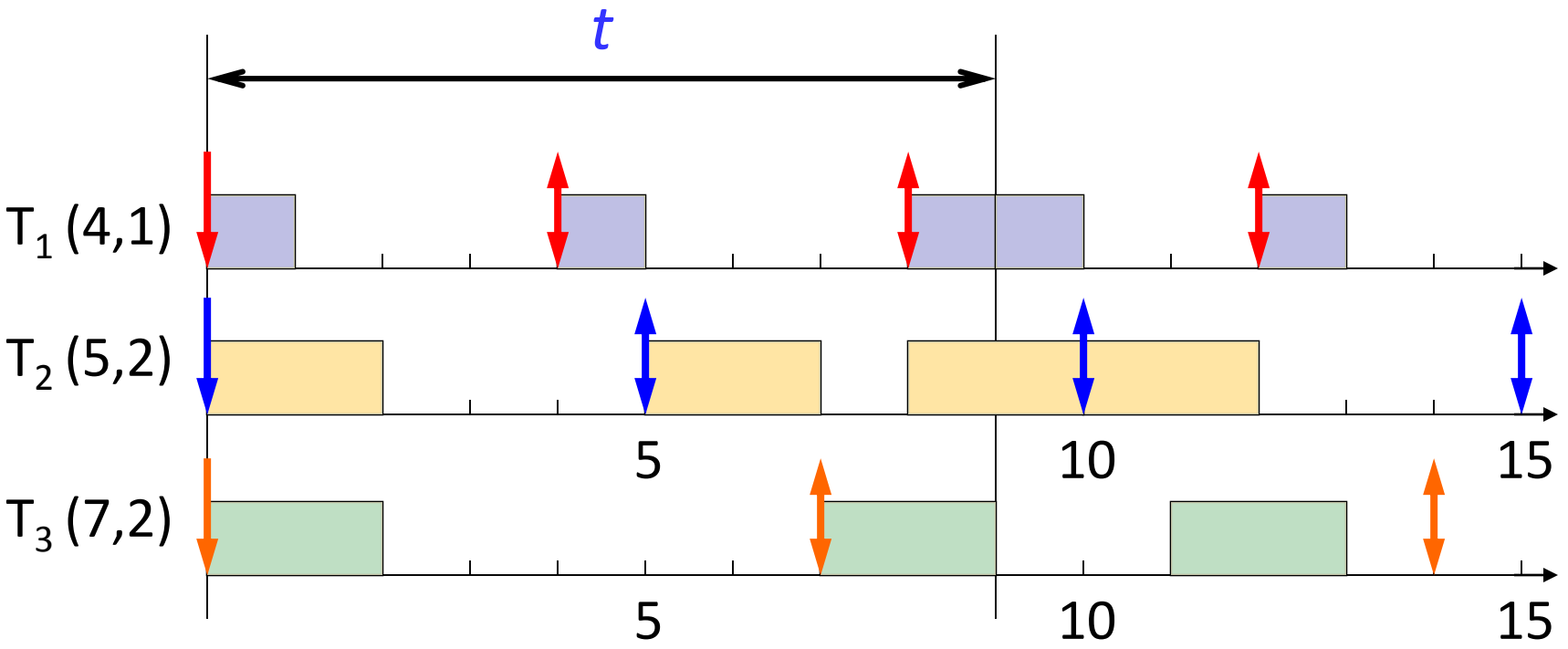
EDF (Earliest Deadline First)

- Optimal scheduling algorithm
  - if there is a schedule for a set of real-time tasks, EDF can schedule it.



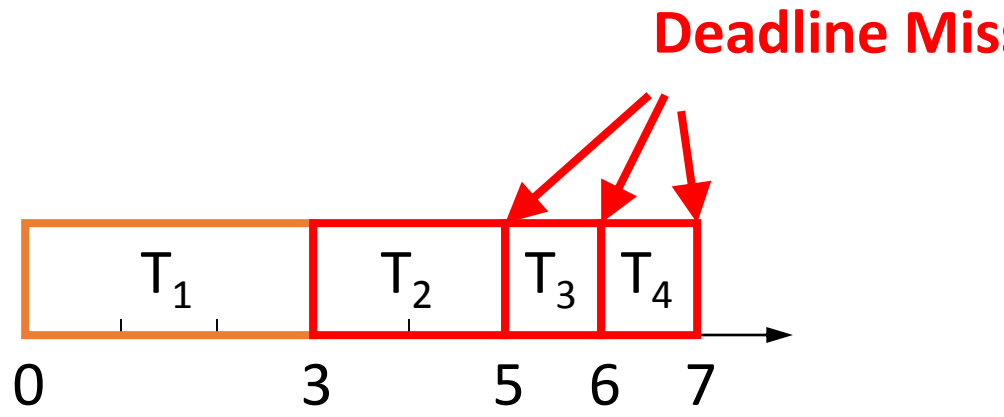
Processor Demand Bound

- Demand Bound Function : *dbf(t)*
  - the maximum processor demand by workload over any interval of length *t*

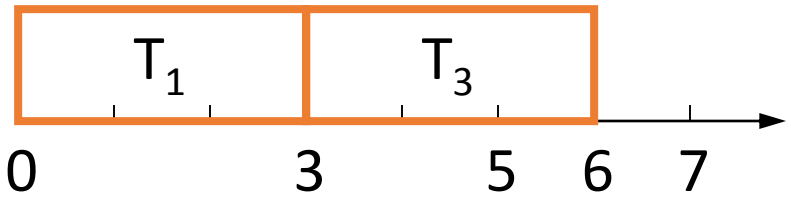


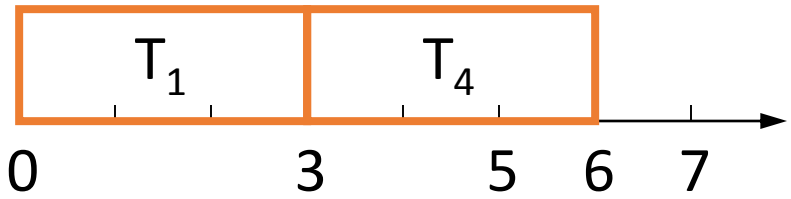
EDF – Overload Conditions

- Domino effect during overload conditions
  - Example: $T_1(4,3)$, $T_2(5,3)$, $T_3(6,3)$, $T_4(7,3)$

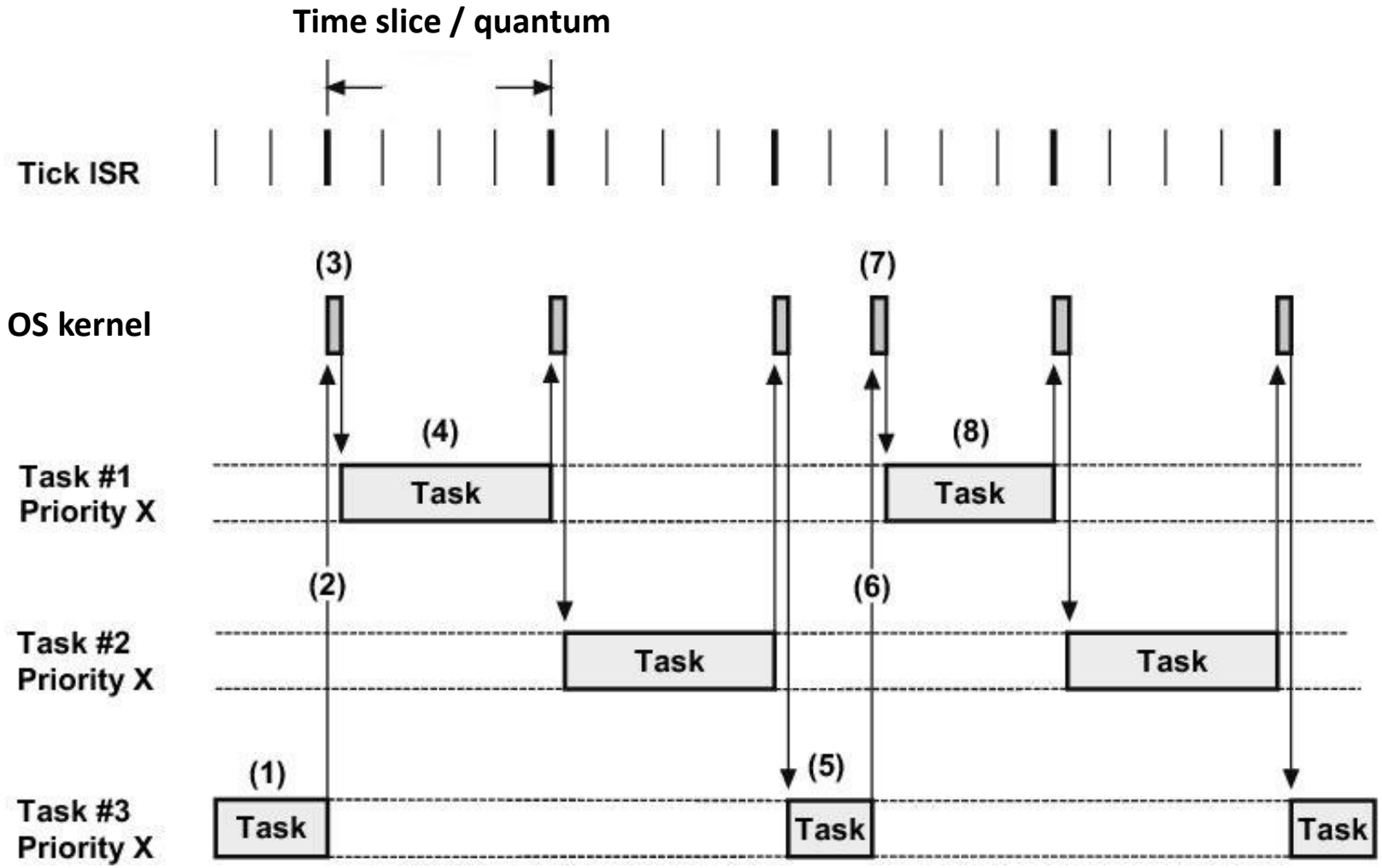


Better schedules :



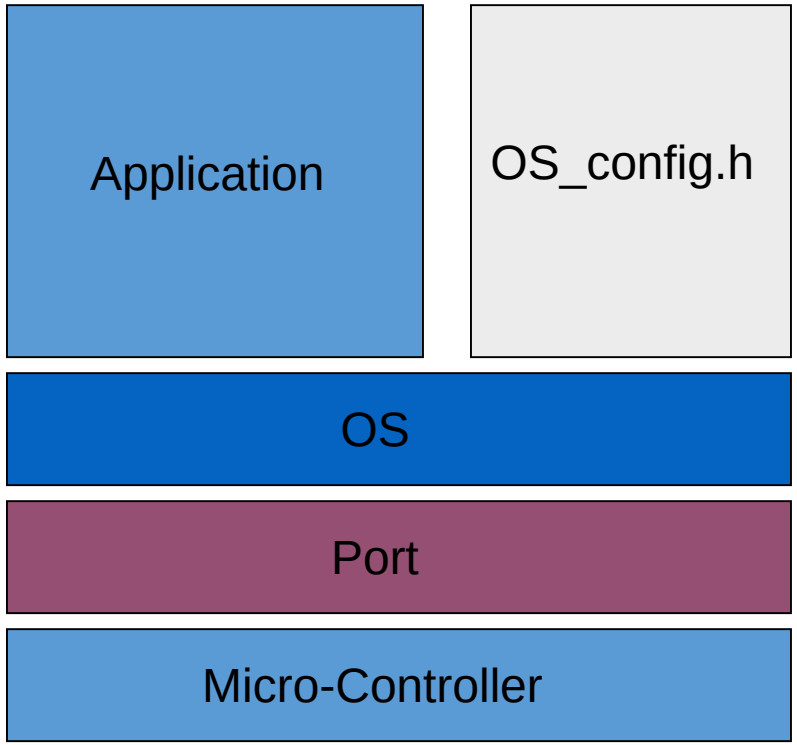


Round Robbin



RTOS

- If all tasks are waiting → Idle task
- Initialization task (creation).
- TCB → Task control block.
- Layers with OS:



Let's get our hands dirty

- Download FreeRTOS latest version and unzip it

The structure of the FreeRTOS/Source directory is shown below.

```
FreeRTOS
    ¦
    +-Source          The core FreeRTOS kernel files
        ¦
        +-include     The core FreeRTOS kernel header files
        ¦
        +-Portable    Processor specific code.
            ¦
            +-Compiler x    All the ports supported for compiler x
            +-Compiler y    All the ports supported for compiler y
            +-MemMang       The sample heap implementations
```

Create your project

1.Source Files

As a minimum, the following source files must be included in your project:

FreeRTOS/Source/tasks.c
FreeRTOS/Source/queue.c
FreeRTOS/Source/list.c
FreeRTOS/Source/portable/[compiler]/[architecture]/port.c.
FreeRTOS/Source/portable/MemMang/heap_x.c where 'x' is 1, 2, 3, 4 or 5.

If the directory that contains the port.c file also contains an assembly language file, then the assembly language file must also be used.

If you need software timer functionality, then include FreeRTOS/Source/timers.c.
If you need co-routine functionality, then include FreeRTOS/Source/croutine.c.

Create your project

2.Header Files

As a minimum, the following directories must be in the compiler's include path (the compiler must be told to search these directories for header files):

•FreeRTOS/Source/include

•FreeRTOS/Source/portable/[compiler]/[architecture].

Depending on the port, it may also be necessary for the same directories to be in the assemblers include path.

3.Configuration File

Every project also requires a file called FreeRTOSConfig.h.
FreeRTOSConfig.h tailors the RTOS kernel to the application being built. It is therefore specific to the application, not the RTOS, and should be located in an application directory, not in one of the RTOS kernel source code directories.

FreeRTOSConfig.h

```
#ifndef  FREERTOS_CONFIG_H
#define FREERTOS_CONFIG_H
//these are the minimum configuration to run the FreeRTOS
#define configCPU_CLOCK_HZ                  16000000UL
#define configUSE_TICKLESS_IDLE             0
#define configTICK_RATE_HZ                  ( ( TickType_t ) 100 )
#define configUSE_PREEMPTION                1
#define configUSE_IDLE_HOOK                 0
#define configUSE_TICK_HOOK                 0
#define configMAX_PRIORITIES                ( 5 )                    // system priorities are from 0-4 and 4 is the highest
#define configMINIMAL_STACK_SIZE            ( ( unsigned short ) 100 )
#define configTOTAL_HEAP_SIZE               ( ( size_t ) ( 1024 ) )
#define configMAX_TASK_NAME_LEN             ( 10 )
#define configUSE_16_BIT_TICKS              0                        // for 16-bit or 8-bit machines but this 1
/* The highest interrupt priority that can be used by any interrupt service routine that makes calls to interrupt safe FreeRTOS API functions.
DO NOT CALL INTERRUPT SAFE FREERTOS API FUNCTIONS FROM ANY INTERRUPT THAT HAS A HIGHER PRIORITY THAN THIS! (higher priorities
are lower numeric values. */
#define configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY    10
/* Interrupt priorities used by the kernel port layer itself.  These are generic
to all Cortex-M ports, and do not rely on any particular library functions. */
#define configKERNEL_INTERRUPT_PRIORITY             ( 7 << 5 )   /* Priority 7, or 0xE0 as only the top three bits are implemented.
                                                                    This is the lowest priority. */
#define configMAX_SYSCALL_INTERRUPT_PRIORITY        ( 5 << 5 )   /* Priority 5, or 0xA0 as only the top three bits are implemented. */

#endif /* FREERTOS_CONFIG_H */
```

For ARM CORTEXM PORT only

AMIT
Beyond Education

Create your project

4.Interrupt Vectors

Every RTOS port uses a timer to generate a periodic tick interrupt. Many ports use additional interrupts to manage context switching. The interrupts required by an RTOS port are serviced by the provided RTOS port source files.
The method used to install the interrupt handlers provided by the RTOS port is dependent on the port and compiler being used.

5.#include “FreeRTOS.h”

If using IAR or CCS with your TIVAC go to the startup code file and apply these changes

```
extern void vPortSVCHandler(void);
extern void xPortPendSVHandler(void);
extern void xPortSysTickHandler(void);

int const __vector_table[] @ ".intvec" = {
  (int)&CSTACK$$Limit,
  (int)&__iar_program_start,
  (int)&NMI_Handler,
  (int)&HardFault_Handler,
  (int)&MemManage_Handler,
  (int)&BusFault_Handler,
  (int)&UsageFault_Handler,
  0,  /* Reserved */
  0,  /* Reserved */
  0,  /* Reserved */
  0,  /* Reserved */
  (int)&vPortSVCHandler,
  (int)&DebugMon_Handler,
  0,  /* Reserved */
  (int)&xPortPendSVHandler,
  (int)&xPortSysTickHandler,
};
```

In Atmel studio while porting the FreeRTOS to the Amit board go to the port.c file and apply these changes

```
#if configUSE_PREEMPTION == 1

    /*
     * Tick ISR for preemptive scheduler.  We can use a naked attribute as
     * the context is saved at the start of vPortYieldFromTick().  The tick
     * count is incremented after the context is saved.
     */
    void SIG_OUTPUT_COMPARE1A( void ) __attribute__ ( ( signal, naked ) );
    void SIG_OUTPUT_COMPARE1A( void )
    {
        vPortYieldFromTick();
        asm volatile ( "reti" );
    }
#if configUSE_PREEMPTION == 1

    /*
     * Tick ISR for preemptive scheduler.  We can use a naked attribute as
     * the context is saved at the start of vPortYieldFromTick().  The tick
     * count is incremented after the context is saved.
     */
    void TIMER1_COMPA_vect( void ) __attribute__ ( ( signal, naked ) );
    void TIMER1_COMPA_vect( void )
    {
        vPortYieldFromTick();
        asm volatile ( "reti" );
    }
```

AMIT
Beyond Education

Task Creation

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

void ATaskFunction(void *pvParameters);

Don't forget to :

#include "task.h"

- 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.

Task Creation

- If a task is no longer required it should instead be explicitly deleted.

- 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 Creation

```
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 in 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 );
}
```

Task Creation

xTaskCreate() API Function

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

There are two possible return values:

1. pdTRUE

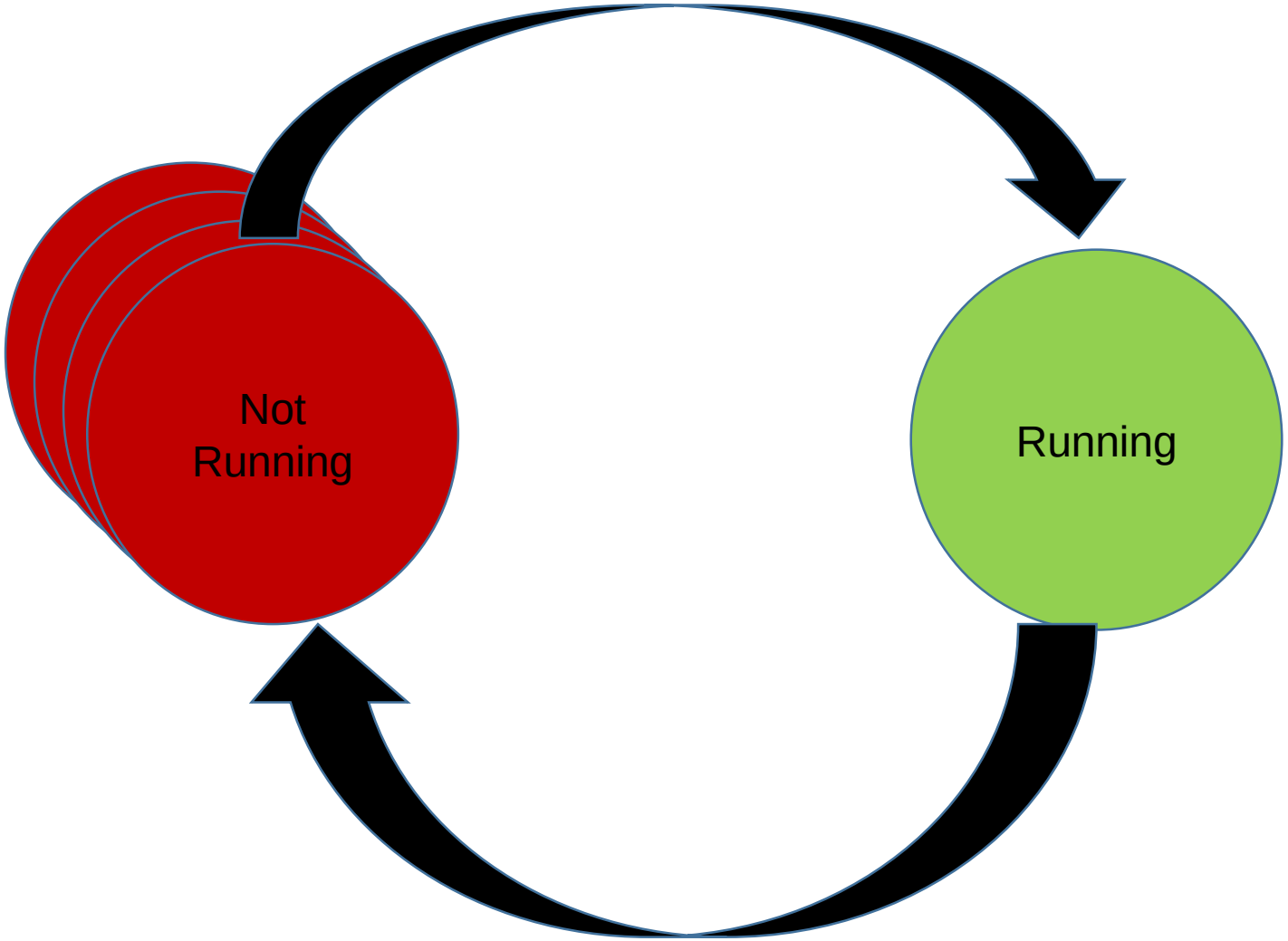
   This indicates that the task was created successfully.

2. errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY

   This indicates that the task could not be created because there was insufficient heap memory available for FreeRTOS to allocate enough RAM to hold the task data structures and stack.

TOP LEVEL TASK STATES

All tasks that are not running are in the not running state



Only one task can be in the running state at any one time

LET'S CODE IT!

```
/* Task code */
void vTask( void *pvParameters )
{
    char *pcTaskName;
    volatile unsigned long ul;
    pcTaskName = ( char * ) pvParameters;
    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
        {
        /* Print out the name of this task. */
        Uart_write_string( pcTaskName );
        /* Delay for a period. */
        for( ul = 0; ul < 0xFFFF; 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. */
            }
        }
}
```

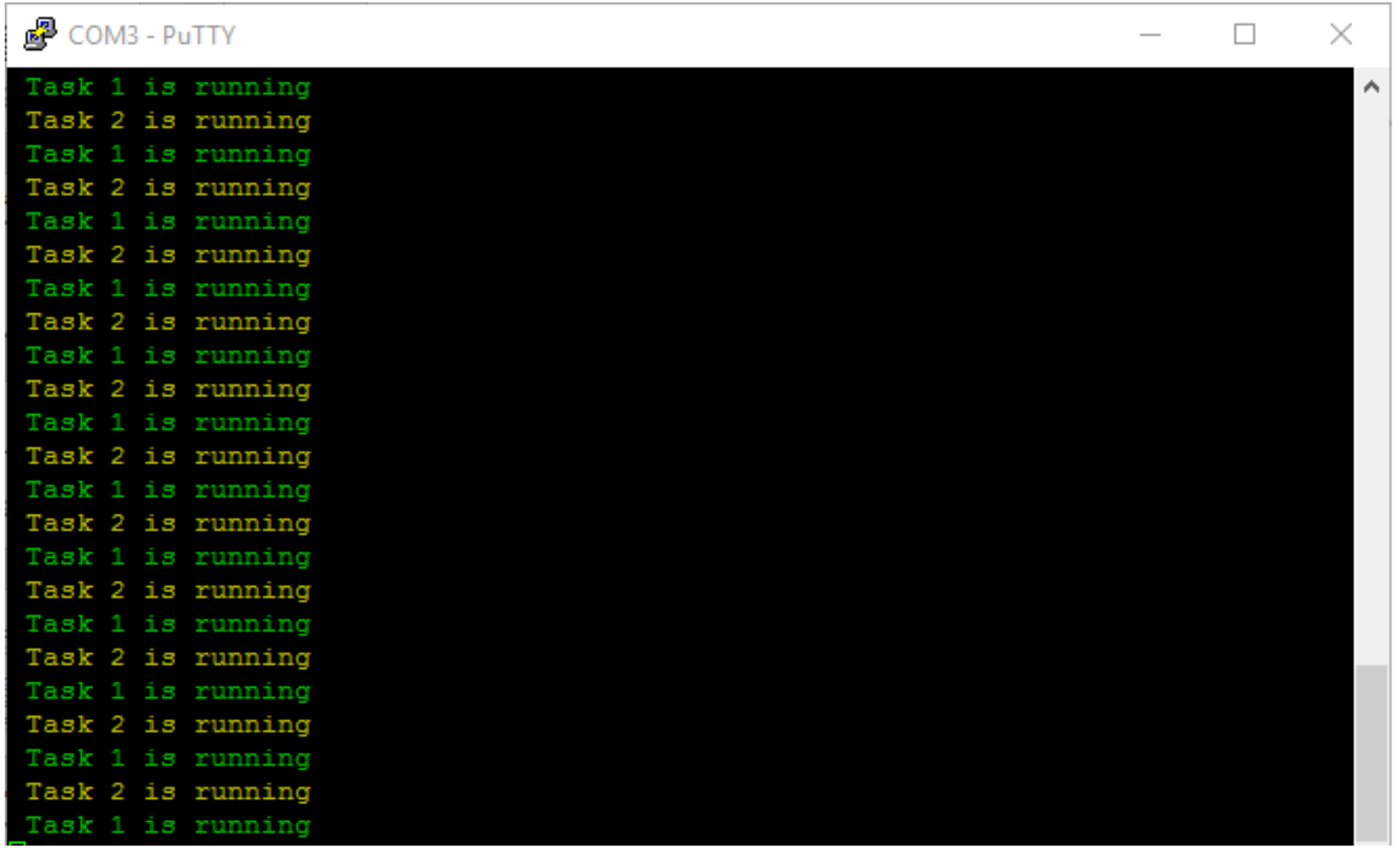
AMIT
Beyond Education

LET'S CODE IT!

```
/* 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 = "\x1B[32m Task 1 is running\r\n";
static const char *pcTextForTask2 = "\x1B[33m Task 2 is running\t\n";
int main( void )
{
    uart_init (9600,16000000UL);
    /* Create one of the two tasks. */
    xTaskCreate (vTask,"Task 1",100,(void*)pcTextForTask1, 1, NULL );
    /* Create the other task in exactly the same way. Note this time that multiple tasks are
    being created from the SAME task implementation (vTask). Only the value passed in the
    parameter is different. Two instances of the same task are being created. */
    xTaskCreate (vTask, "Task 2",100, (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.
    while( 1 );
}
```

AMIT
Beyond Education

Test it on your board!



Time slicing

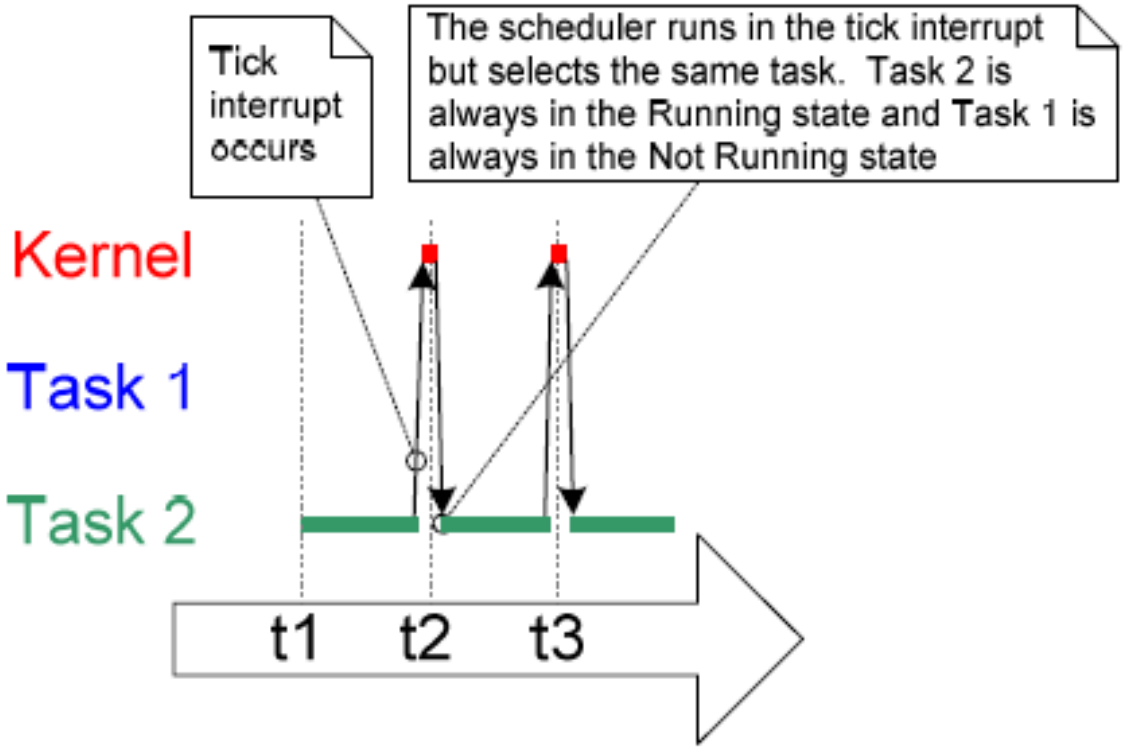
```
#define configUSE_TIME_SLICING       1
```

configUSE_TIME_SLICING

By default (if configUSE_TIME_SLICING is not defined, or if configUSE_TIME_SLICING is defined as 1) FreeRTOS uses prioritised preemptive scheduling with time slicing. That means the RTOS scheduler will always run the highest priority task that is in the Ready state, and will switch between tasks of equal priority on every RTOS tick interrupt. If configUSE_TIME_SLICING is set to 0 then the RTOS scheduler will still run the highest priority task that is in the Ready state, but will not switch between tasks of equal priority just because a tick interrupt has occurred.

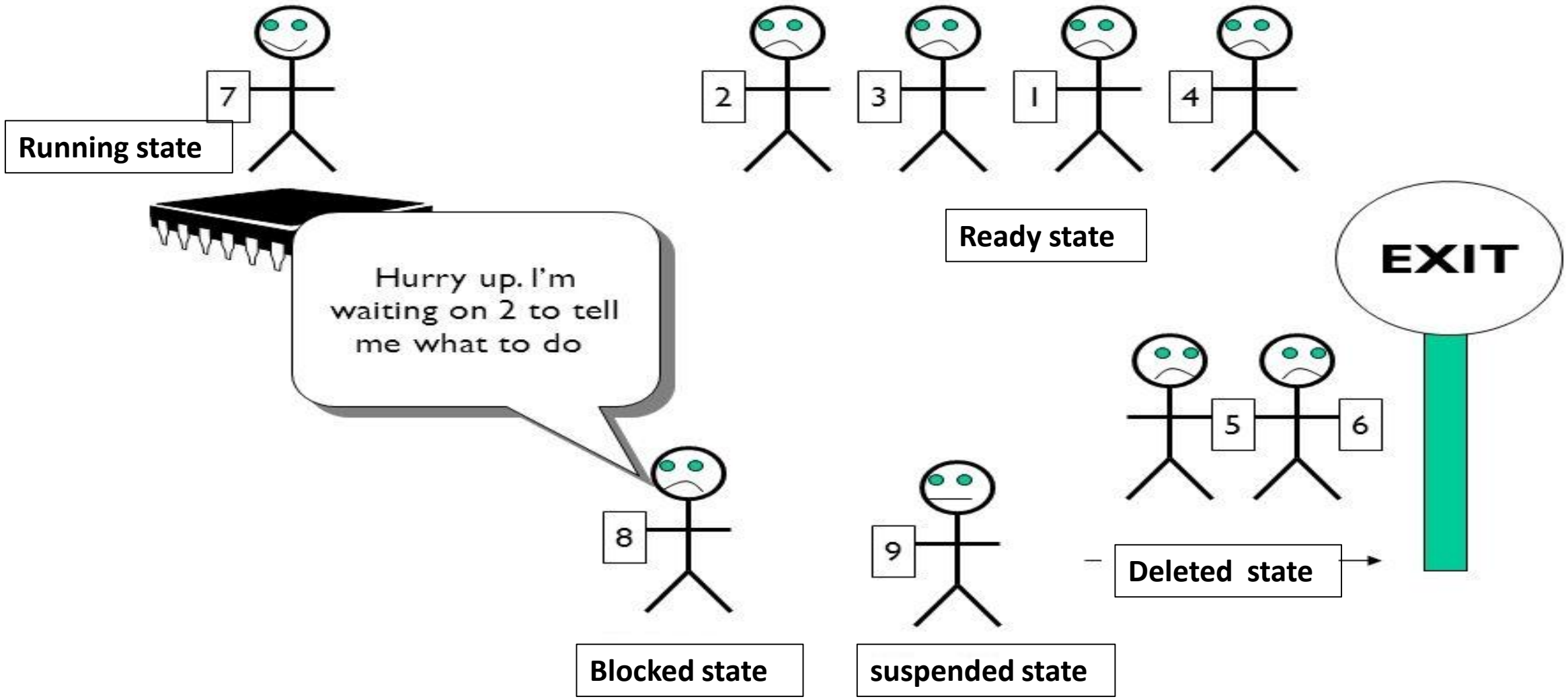
Let's code it!

- Now put task2 priority 2 and leave task1 as 1
- What did you get as a result and why?!

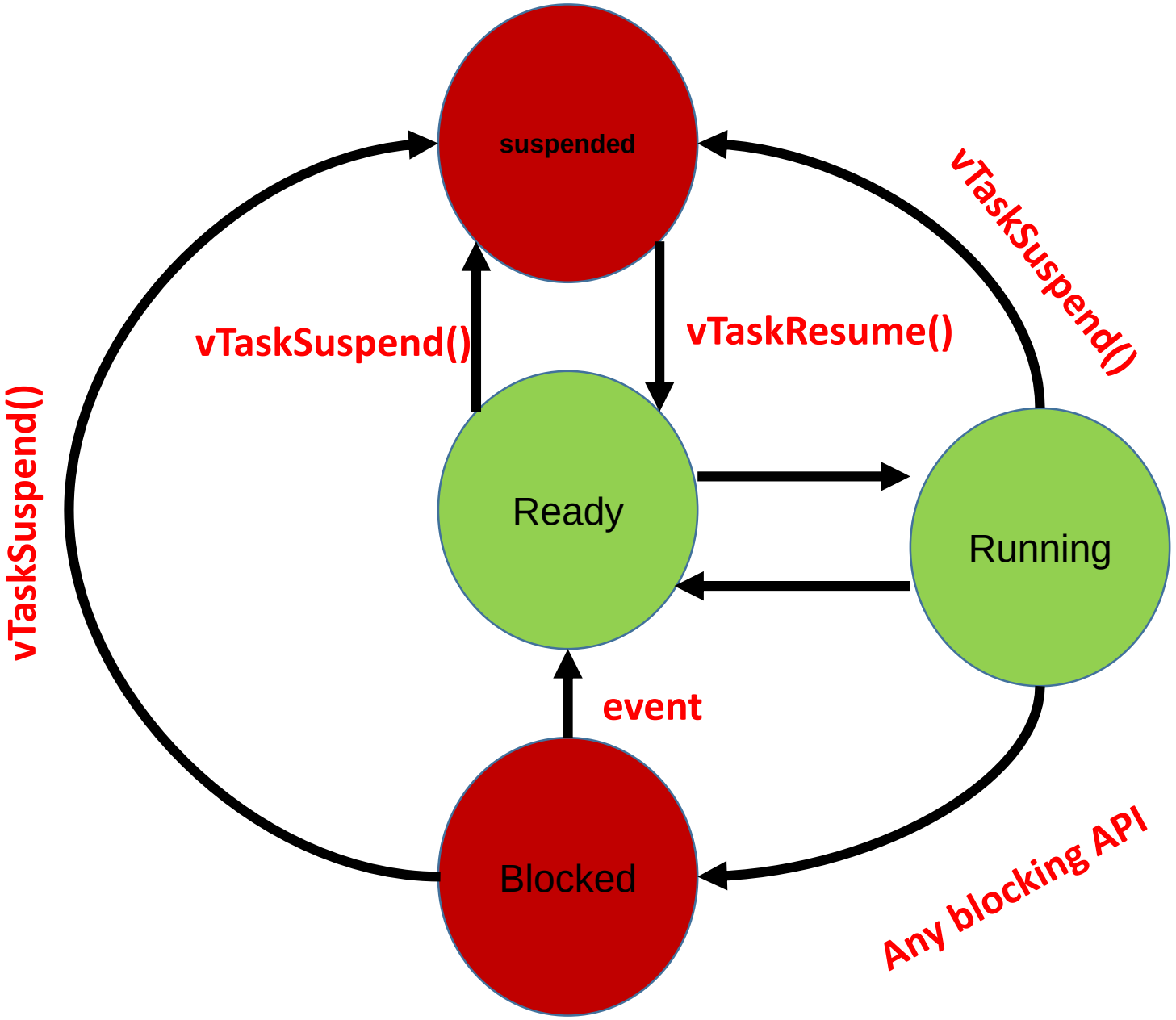


AMIT
Beyond Education

EXPANDING THE 'NOT RUNNING' STATE



EXPANDING THE 'NOT RUNNING' STATE



USING THE BLOCKING STATE TO GENERATE DELAYS

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

Before using vTaskDelay API add this line to FreeRTOSConfig.h file

```
#define INCLUDE_vTaskDelay              1
```

```
void vTaskDelayUntil( portTickType * pxPreviousWakeTime, portTickType xTimeIncrement );
```

Before using vTaskDelayUntil API add this line to FreeRTOSConfig.h file

```
#define INCLUDE_vTaskDelayUntil         1
```

Let's code it!

- Use vTaskDelay instead of the for loop delay
- what is the result and why?
- What is the difference between vTaskDelay and vTaskDelayUntil?

AMIT
Beyond Education

LET'S CODE IT!

Toggle the RED Led on your Tiva C every 200 ms, the green led every 350 ms and blue led every 500 ms using FreeRTOS

Now it's getting easy again

AMIT
Beyond Education

Task Control APIs

- vTaskDelay
- vTaskDelayUntil
- uxTaskPriorityGet
- vTaskPrioritySet
- vTaskSuspend
- vTaskResume
- xTaskResumeFromISR
- xTaskAbortDelay

LET'S CODE IT!

Use SW1 to change the toggling frequency of the red led between 100ms to 900ms

AMIT
Beyond Education

LET'S CODE IT!

```
void vButtonTask (void * para)
{
        uint8_t flag_low = Pin0 ;  //to work on falling edge
        while (1)
        {
                DIO_PortRead (PortF,Pin0,&data);
                if ((data == DIO_LOW) && (flag_low == Pin0)) //active low , if button is pressed , and last time was high
                {
                        speed += 1;
                        speed = (speed > 10) ? 1 :speed;
                }
                flag_low = data;
                vTaskDelay (50);     //every 50 ticks, as 50 ms which is less than human touch sensitivity
        }
```

AMIT
Beyond Education

LET'S CODE IT!

```
void vLedTask(void * para)
{

        uint8_t  state = DIO_LOW;
        TickType_t xLastWakeTime;
        xLastWakeTime = xTaskGetTickCount();
        while(1)
        {
                DIO_PortToggle (PortF,RED);
                 vTaskDelayUntil ( &xLastWakeTime, 100*speed);

        }
}
```

AMIT
Beyond Education

LET'S CODE IT!

```
uint8_t  speed = 1;   /*shared global variable !!, may cause problems Probably the worst way ever to
                      share data*/
volatile uint32_t data = 0;
 int main(void)
 {

            DIO_PortInit (PortF, Pin0|Pin1|Pin2|Pin3|Pin4 , Pin0|Pin4);
            DIO_PortDirection (PortF , Pin1|Pin2|Pin3, DIO_OUTPUT);
            DIO_PortDirection (PortF , Pin0|Pin4, DIO_INPUT);

            xTaskCreate (vLedTask, "LED",100, NULL ,1, NULL);
            xTaskCreate (vButtonTask, "Button",100, NULL ,2, NULL);

            vTaskStartScheduler();
            for(;;);
            return 0;
}
```

AMIT
Beyond Education

Task Utilities APIs

- uxTaskGetSystemState
- vTaskGetInfo
- xTaskGetCurrentTaskHandle
- xTaskGetIdleTaskHandle
- uxTaskGetStackHighWaterMark
- eTaskGetState
- pcTaskGetName
- xTaskGetHandle
- xTaskGetTickCount
- xTaskGetTickCountFromISR
- xTaskGetSchedulerState
- uxTaskGetNumberOfTasks
- vTaskList
- vTaskStartTrace
- ulTaskEndTrace
- vTaskGetRunTimeStats
- vTaskSetApplicationTaskTag
- xTaskGetApplicationTaskTag
- xTaskCallApplicationTaskHook
- pvTaskGetThreadLocalStoragePointer
- vTaskSetThreadLocalStoragePointer
- vTaskSetTimeOutState
- xTaskCheckForTimeOut

LET'S CODE IT!

Add vTaskState to the previous example

```
void vTaskState (void *para)
{
        TaskHandle_t state1 =  ((struct Two_handle *)para)->first_task;        //task 1 handle
        TaskHandle_t state2 =  ((struct Two_handle *)para)->second_task;     //task 2 handle
        TaskStatus_t Details1,Details2;                                                    // 2 structs for task1 and 2
        TaskStatus_t *pxTaskStatusArray;
        volatile UBaseType_t number,x;
        unsigned long ulTotalRunTime;
        number =uxTaskGetNumberOfTasks ();
        pxTaskStatusArray = pvPortMalloc ( number * sizeof( TaskStatus_t ) );
        uxTaskGetSystemState ( pxTaskStatusArray,number,&ulTotalRunTime );
         uart_write_string ("Number of tasks = ");
        uart_sendnumber (number);
        for (x = 0 ; x < number ; x ++ )   //To print tasks names
        {
         uart_write_string (pxTaskStatusArray[x].pcTaskName);
         uart_write_string ("/r/n")
        }
```

LET'S CODE IT!

```
While (1)
        {
          vTaskGetInfo (state1,&Details1, pdTRUE ,  eInvalid );
           vTaskGetInfo (state2,&Details2, pdTRUE ,eInvalid );
           uart_write_string (Details1.pcTaskName);
           uart_write_string ("/r/n");
           uart_write_string (state[Details1.eCurrentState]);
           uart_write_string ("/r/n");
           uart_write_string (Details2.pcTaskName);
           uart_write_string ("/r/n");
           uart_write_string (state[Details2.eCurrentState]);
           uart_write_string ("/r/n");
            vTaskDelay (1000);
        }
}
```

LET'S CODE IT!

```
struct Two_handle{
        TaskHandle_t first_task;
        TaskHandle_t second_task;
        }handles;
uint8_t *state[] = {"running","ready","blocked","suspended"};

 int main(void)
 {
        DIO_PortInit (PortF, Pin0|Pin1|Pin2|Pin3|Pin4 , Pin0|Pin4);
        DIO_PortDirection (PortF , Pin1|Pin2|Pin3, DIO_OUTPUT);
        DIO_PortDirection (PortF , Pin0|Pin4, DIO_INPUT);
        TaskHandle_t  First_handle,Second_handle,Third_handle;
        xTaskCreate (vLedTask, "LED",100, NULL,1, &First_handle);
        xTaskCreate (vButtonTask, "Button",100, NULL,2, &Second_handle);
        handles.first_task = First_handle;
        handles.second_task = Second_handle;
        xTaskCreate (vTaskState, "Statistics",100, (void * )(&handles),3, &Third_handle);
        vTaskStartScheduler();
        for(;;);
        return 0;
}
```

SHARED VARIABLE OR DISASTER?!

- Use these tasks to create an app and run it on your board
- Why do think you got this result?!!!

```
Uint32_t x = 0 , y = 0;


void vTask1 (void * para)
{
while (1)
        {
        If (x != y)
        {
          uart_write_string("Shared data problem /r/n");
        }
        vTaskDelay (47);
        }
}
```

```
void vTask2 (void * para)
{
        while (1)
        {
         x ++;
         //shared data problem happens here
         y ++;
        }
}
```

AMIT
Beyond Education

RACE CONDITION

C

```
char x;
void foo1(void)
{
    x++;
}
```

Assembly

```
char x;
foo1:
    mov x,R1;
    add R1,1;
    mov R1,x;
```

Task1

```
/*Some Code*/
    foo1();
/*Some Code*/
```

Task2

```
/*Some Code*/
    foo1();
/*Some Code*/
```

RACE CONDITION

x = 1

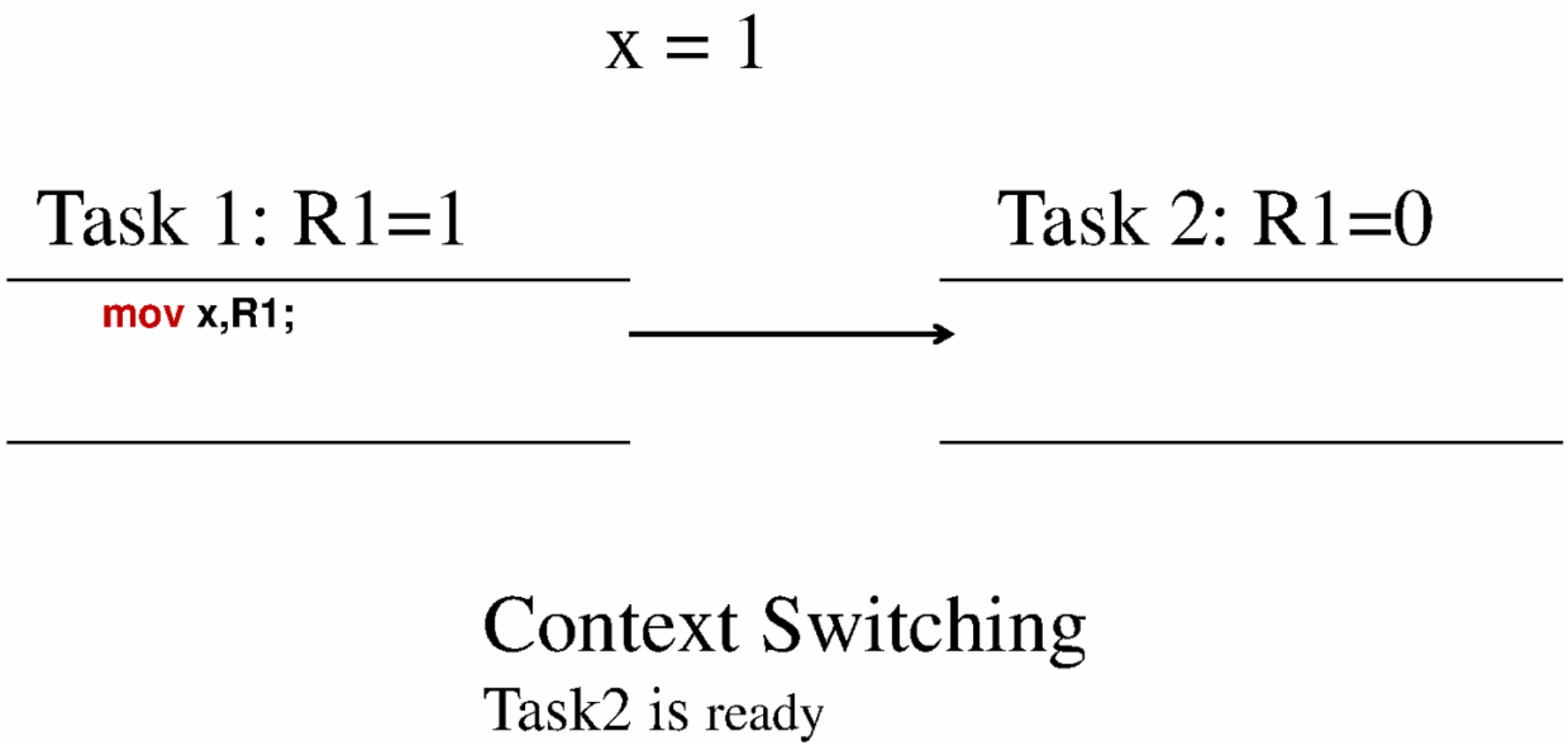
Task 1: R1=0	Task 2: R1=0

RACE CONDITION

x = 1

Task 1: R1=1	Task 2: R1=0
mov x,R1;	

RACE CONDITION



RACE CONDITION

x = 1

Task 1: R1=1	Task 2: R1=1
mov x,R1;	mov x,R1;

RACE CONDITION

$$x = 1$$

Task 1: R1=1	Task 2: R1=2
mov x,R1;	mov x,R1; add R1,1;

RACE CONDITION

$$x = 2$$

Task 1: R1=1	Task 2: R1=2

mov x,R1;	mov x,R1;
	add R1,1;
	mov R1,x;

RACE CONDITION

x = 2

Task 1: R1=1	Task 2: R1=2

mov x,R1;	mov x,R1;
	add R1,1;
	mov R1,x;

Context Switching

Task2 is back to waiting

RACE CONDITION

$$x = 2$$

Task 1: R1=2	Task 2: R1=2

mov x,R1;
add R1,1;
	mov x,R1;
	add R1,1;
	mov R1,x;

RACE CONDITION

x = 2!!!!!!!

Task 1: R1=2	Task 2: R1=2

mov x,R1;
add R1,1;
mov R1,x;
	mov x,R1;
	add R1,1;
	mov R1,x;

REENTRANCY

- When to doubt your function reentrancy?
  - When your function accesses a global variable while this variable is accessed in:
    - ISR
    - Another task
    - Hardware module.

REENTRANCY

- How to make a non-reentrant function reentrant?
  - Either using critical section or any task synchronization services provided by the OS.

- Critical Section:
  - Context Switching always happens after an interrupt.
  - If I disabled interrupts during the time I don't want the schedule nor any ISR interrupts the running task then this part of code is reentrant.

REENTRANCY

Atomic Operation

A single or sequence of operations that cannot be preempted,

Try the example now

```
void vTask2 (void * para)
{
        while (1)
        {
        taskENTER_CRITICAL ();
         x ++;                          //this is also can be called mutual exclusion
         y ++;
        taskEXIT_CRITICAL ();
        }
}
```

AMIT
Beyond Education

TASK COMMUNICATION AND SYNCHRONIZATION

Communication:

queues.

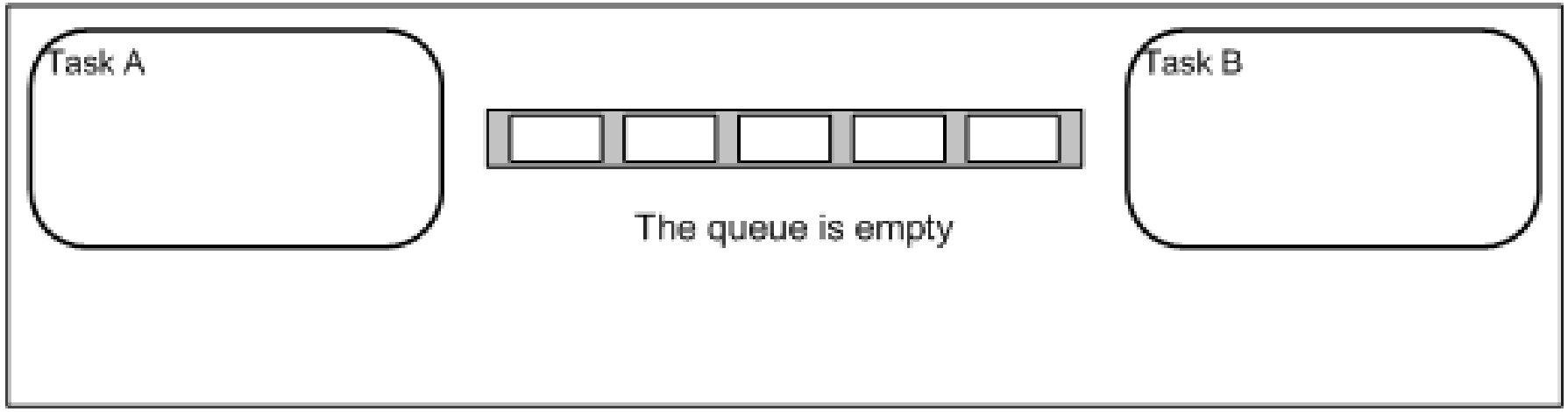
Synchronization:

Semaphores

Mutexes (MUTual EXclusion) semaphores.

QUEUE MANAGEMENT

- 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 data to a queue causes a byte for byte copy of the data to be stored in the queue itself.
- Reading data from a queue causes the copy of the data to be removed from the queue.



QUEUE MANAGEMENT

Access by Multiple Tasks

Queues are objects in their own right that are not owned by or assigned to any particular task. Any number of tasks can write to the same queue and any number of tasks can read from the same queue. A queue having multiple writers is very common while a queue having multiple readers is quite rare.

Blocking on Queue Reads

When a task attempts to read from a queue it can optionally specify a ‘block’ time. This 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 automatically moved from the Blocked state to the Ready state if the specified block time expires before data becomes available.

Blocking on Queue Writes

Just as when reading from a queue, a task can optionally specify a block time when writing to a queue. In this case 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.

USING A QUEUE

```
Don't forget to #include "queue.h"

xQueueHandle xQueueCreate( unsigned portBASE_TYPE uxQueueLength,
                           unsigned portBASE_TYPE uxItemSize
                           );
```

There are two possible return values:

1.NULL : the queue could not be created because there was insufficient heap memory available for FreeRTOS to allocate the queue data structures and storage area.

2.A non-NULL value: indicates that the queue was created successfully. The returned value should be stored as the handle to the created queue.

USING A QUEUE

```
portBASE_TYPE xQueueSendToFront( xQueueHandle xQueue,
                                 const void * pvItemToQueue,
                                 portTickType xTicksToWait
                                 );
portBASE_TYPE xQueueSendToBack( xQueueHandle xQueue,
                                const void * pvItemToQueue,
                                portTickType xTicksToWait
                                 );
```

There are two possible return values:

1. pdPASS: data was successfully sent to the queue. If a block time was specified (xTicksToWait was not zero) then it is possible that the calling task was placed in the Blocked state to wait for space to become available in the queue before the function returned – but data was successfully written to the queue before the block time expired.

2. errQUEUE_FULL: data could not be written to the queue because the queue was already full

USING A QUEUE

For receiving item from the queue and removing it

```
portBASE_TYPE xQueueReceive( xQueueHandle xQueue,
                             const void * pvBuffer,
                             portTickType xTicksToWait
                           );
```

For receiving item from the queue without removing it

```
portBASE_TYPE xQueuePeek( xQueueHandle xQueue,
                          const void * pvBuffer,
                          portTickType xTicksToWait
                           );
```

There are two possible return values:
1. pdPASS
2. errQUEUE_EMPTY

USING A QUEUE

- **NOTE:**

never use `xQueueSendToFront`, `xQueueSendToBack`, `xQueueReceive` or `xQueuePeek` APIs from an ISR, instead use the APIs that ends with `FromISR()`

LET'S CODE IT!

Blocking When Receiving From a Queue

LET'S CODE IT!

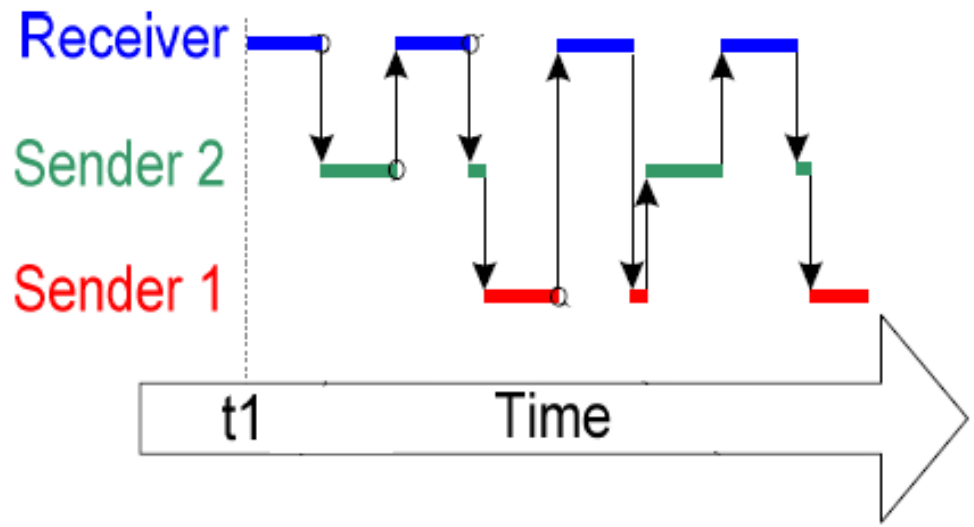
```
void vSenderTask ( void *pvParameters )
{
  long lValueToSend;
  portBASE_TYPE xStatus;
 lValueToSend = ( long ) pvParameters;
for ( ;; )
        {
                 xStatus = xQueueSendToBack ( xQueue, &lValueToSend, 0 );
                if ( xStatus != pdPASS )
                {
                        /* 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! */
                        uart_write_string ( "Could not send to the queue.\r\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 ();
        }
}
```

AMIT
Beyond Education

```
void vReceiverTask ( void *pvParameters )
{
 long lReceivedValue;
 portBASE_TYPE xStatus;
 const portTickType xTicksToWait = 100 / portTICK_RATE_MS;
for ( ;; )
        {
        if( uxQueueMessagesWaiting ( xQueue ) != 0 )
                {
                 uart_write_string ( "Queue should have been empty!\r\n" );
                }
        xStatus = xQueueReceive ( xQueue, &lReceivedValue, xTicksToWait );
         if ( xStatus == pdPASS )
                {
                  if (lReceivedValue==100)       uart_write_string ("\x1B[31mReceived = ");
                  else                           uart_write_string ("\x1B[32mReceived = ");
                   uart_write_int ( lReceivedValue );
                   uart_write_string ("\r\n");
                }
         else
                {
                 uart_write_string ( "Could not receive from the queue.\r\n" );
                }
        }
}
```

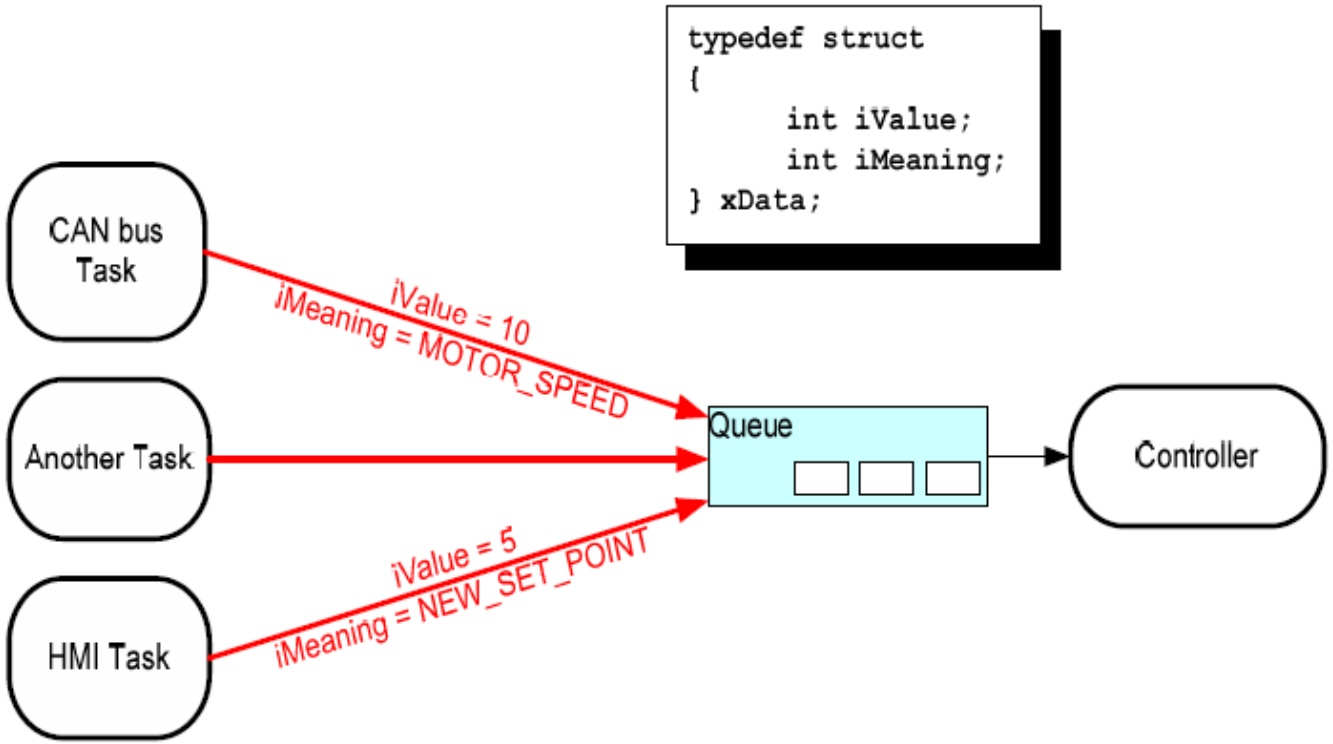
```
xQueueHandle xQueue;

int main( void )
{
  uart_init (9600,16000000);
  xQueue = xQueueCreate ( 1, sizeof( long ) );
  if ( xQueue != NULL )
        {
          /* Create two instances of the task that will send 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", 100, ( void * ) 100, 1, NULL );
          xTaskCreate ( vSenderTask, "Sender2", 100, ( 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", 100, NULL, 2, NULL );
          vTaskStartScheduler ();
        }
  else
        {
          /* The queue could not be created. */
        }
for ( ;; );
}
```

```
Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
```



USING QUEUES TO TRANSFER COMPOUND TYPES

- It is common for a task to receive data from multiple sources on a single queue. Often the receiver of the data needs to know where the data came from so it can determine how it should be processed. A simple way of achieving this is to use the queue to transfer structures where both the value of the data and the source of the data are contained in the structure fields



LET'S CODE IT!

- Blocking When Sending Structures to a Queue

Let's code it!

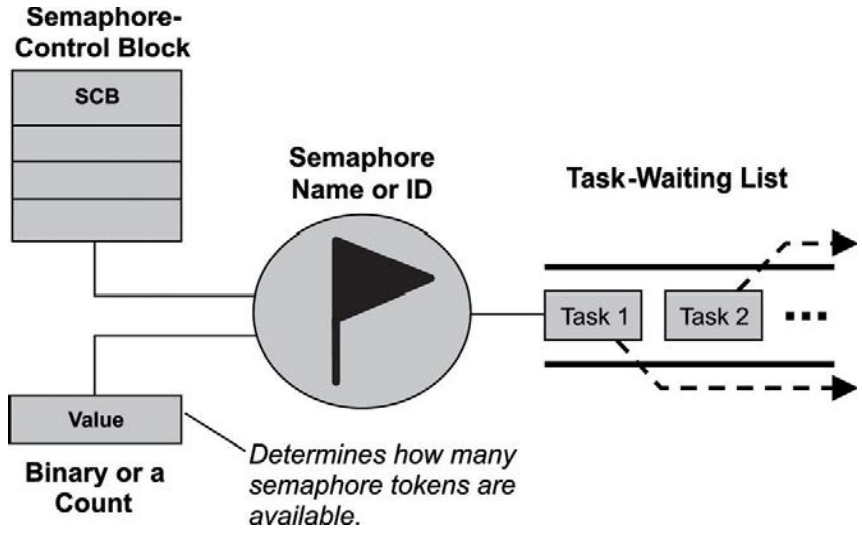
- Use SW1 to change the toggling color of the RGB leds between RED,BLUE,GREEN,YELLOW,CYAN, PINK and WHITE, and send the name of this color to the terminal.

SEMAPHORES

A semaphore (sometimes called a semaphore token) is a kernel object that one or more threads of execution can acquire or release for the purposes of synchronization or mutual exclusion.

When a semaphore is first created, the kernel assigns to it an associated

- semaphore control block (SCB),
- a unique ID,
- a value (binary or a count), and
- a task-waiting list



SEMAPHORES

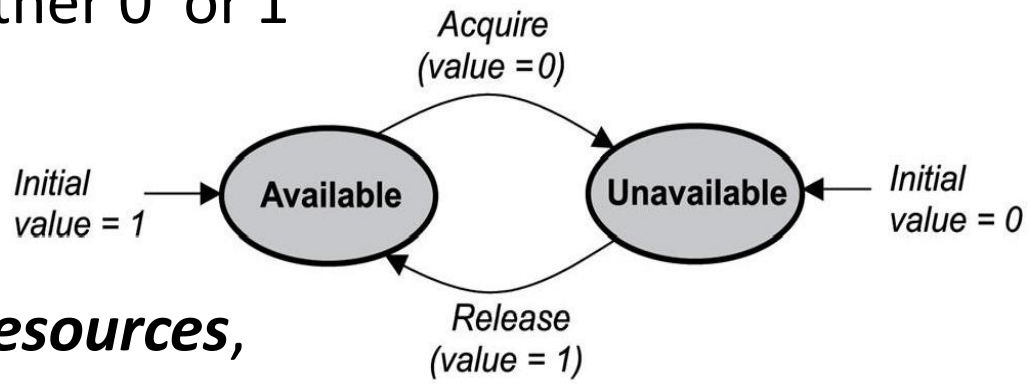
A kernel can support many different types of semaphores, including

- binary
- counting
- mutual-exclusion (mutex) semaphores.

BINARY SEMAPHORES

A *binary semaphore* can have a value of either 0 or 1

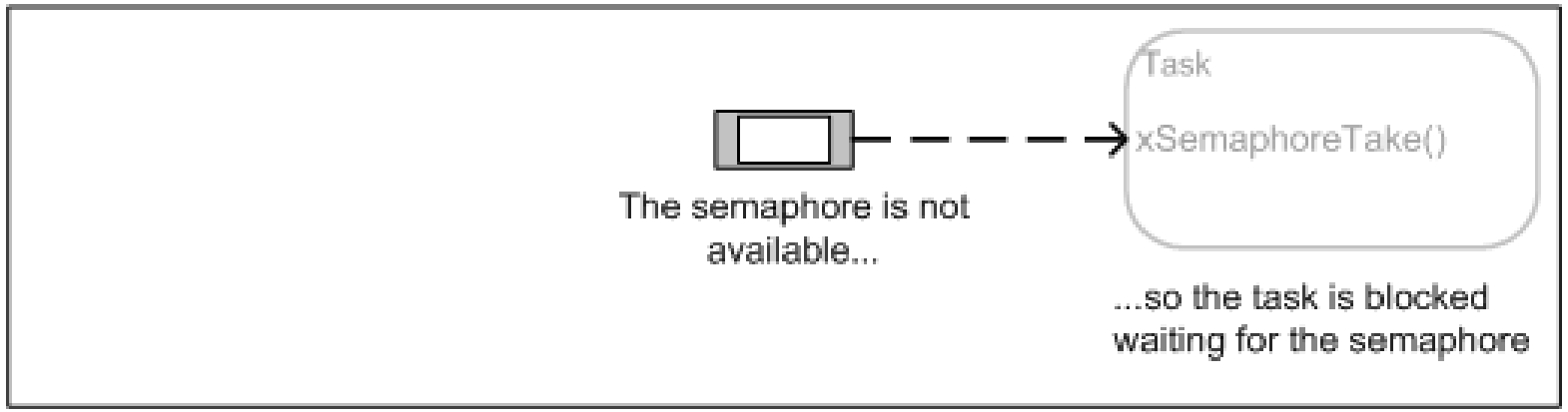
- 0, semaphore is *unavailable* (or *empty)*
- 1, the semaphore is *available* (or *full*).



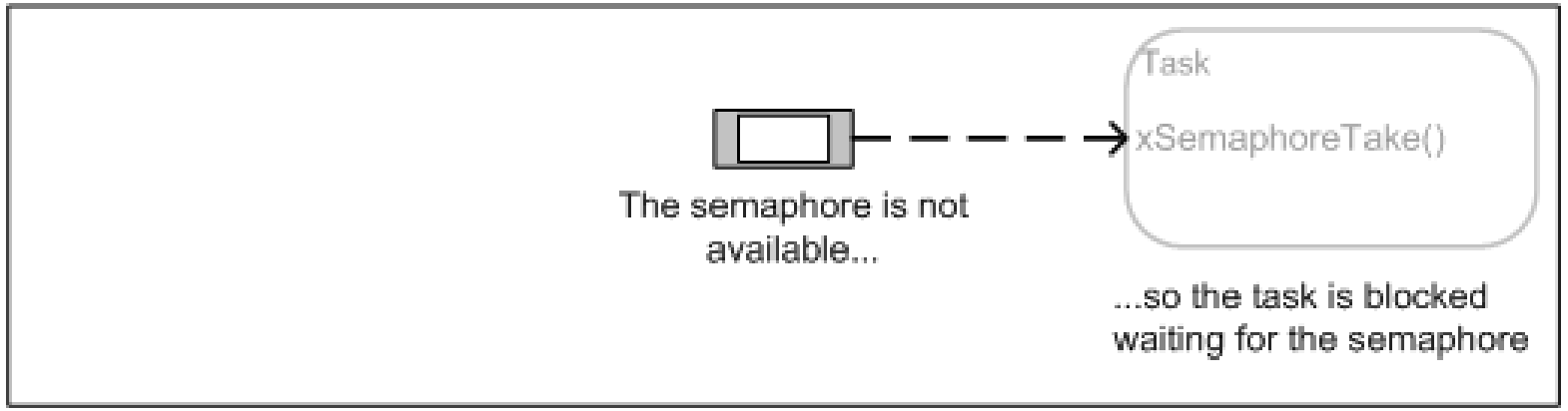
Binary semaphores are treated as ***global resources***,

- they are shared among ***all tasks*** that need them.
- Making the semaphore a global resource allows any task to release it, even if the task did not initially acquire it???.

BINARY SEMAPHORES

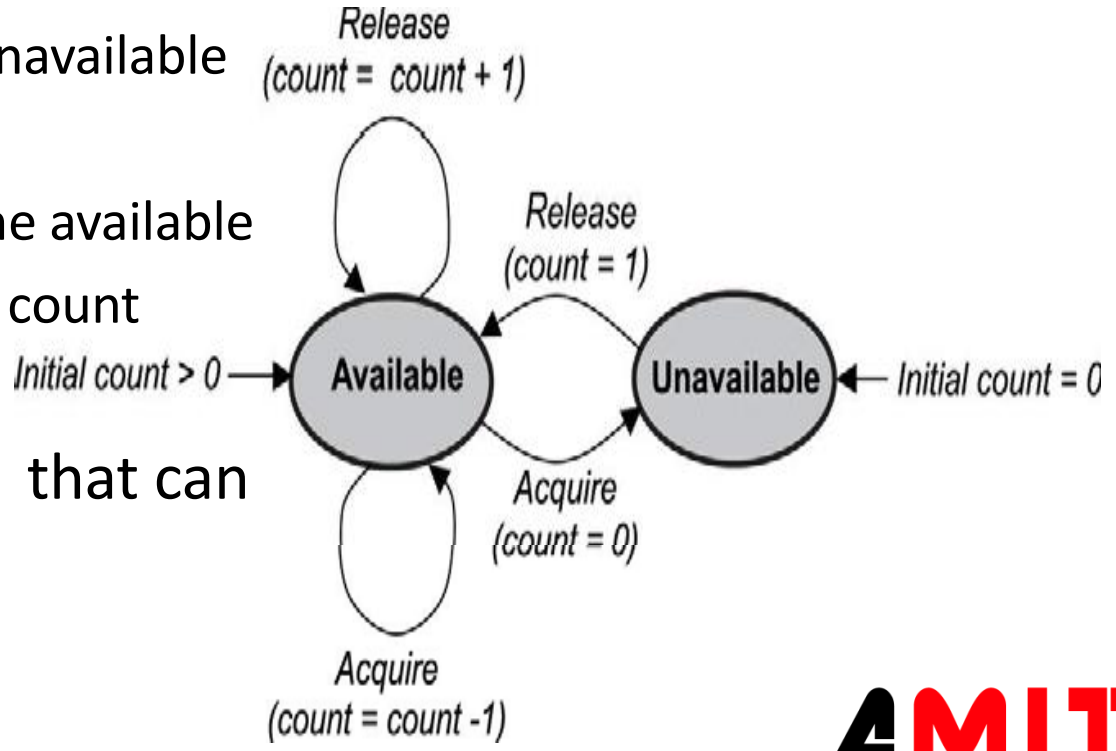


OR



COUNTING SEMAPHORES

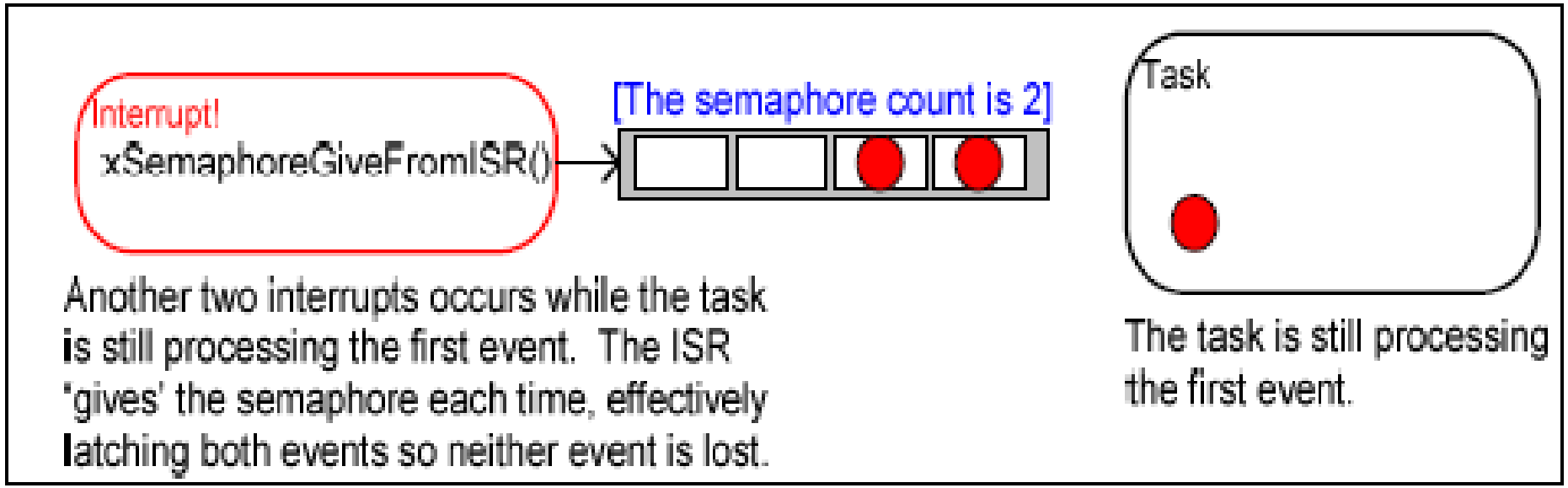
- A *counting semaphore* uses a count to allow it to be acquired or released multiple times.
- If the initial count is
  - 0, the counting semaphore is created in the unavailable state.
  - greater than 0, the semaphore is created in the available state, and the number of tokens it has equals its count



- counting semaphores are **global resources** that can be shared by all tasks that need them

AMIT
Beyond Education

COUNTING SEMAPHORES



USING SEMAPHORES

Don't forget to #include "semphr.h"

```
portBASE_TYPE xSemaphoreCreateBinary( xSemaphoreHandle xSemaphore );

portBASE_TYPE xSemaphoreTake( xSemaphoreHandle xSemaphore,
                              portTickType xTicksToWait );
```

Returned value :
1. pdPASS: the call to xSemaphoreTake() was successful in obtaining the semaphore.
2. pdFALSE: The semaphore was not available.

```
portBASE_TYPE xSemaphoreGive( SemaphoreHandle_t xSemaphore );
```

Returned value :
1.pdTRUE: if the semaphore was released.
2.pdFALSE: if an error occurred. Semaphores are implemented using queues. An error can occur if there is no space on the queue to post a message - indicating that the semaphore was not first obtained correctly.

EXAMPLE USAGE

```
SemaphoreHandle_t xSemaphore = NULL;
/* A task that creates a semaphore. */
 void vATask ( void * pvParameters )
 {
        /* Create the semaphore to guard a shared resource. */
       xSemaphore = xSemaphoreCreateBinary ();
 }
 /* A task that uses the semaphore. */
void vAnotherTask ( void * pvParameters )
{
 /* ... Do other things. */
 if ( xSemaphore != NULL )
        {
        /* See if we can obtain the semaphore. If the semaphore is not available wait 10 ticks to see if it becomes free. */
        if ( xSemaphoreTake ( xSemaphore, ( TickType_t ) 10 ) == pdTRUE )
               {
                      /* We were able to obtain the semaphore and can now access the shared resource. */ /* ... */ /*
                       We have finished accessing the shared resource. Release the semaphore. */
                       xSemaphoreGive ( xSemaphore );
               }
        else
               {
                             /* We could not obtain the semaphore and can therefore not access the shared resource safely. */
                }
        }
 }
```

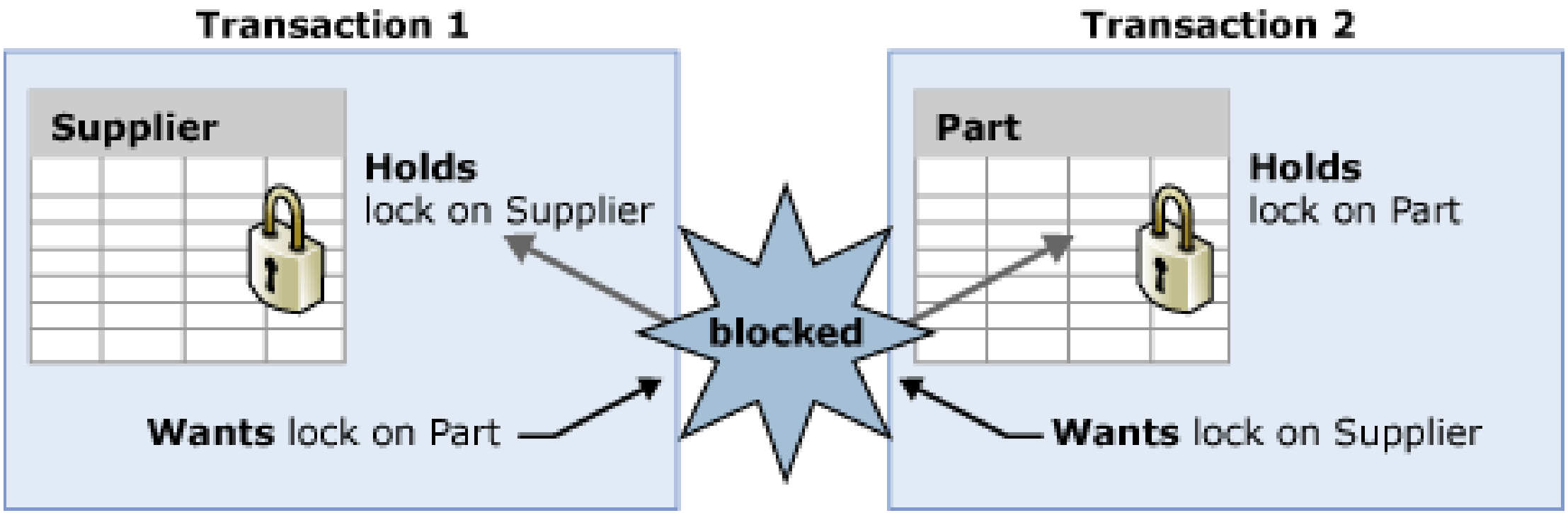
AMIT
Beyond Education

Let's code it!

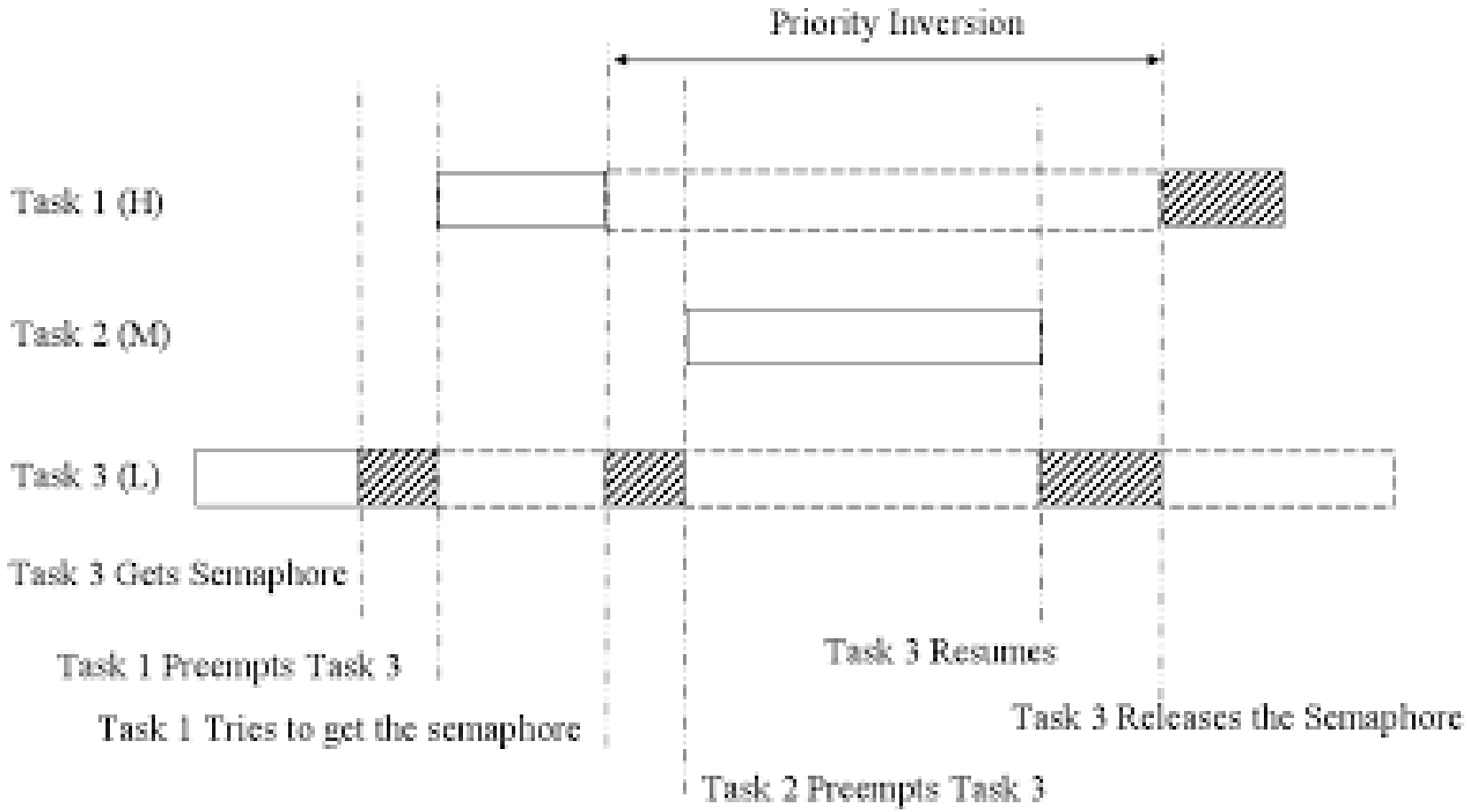
- Block a counts sent over the terminal on SW1 being pressed.

AMIT
Beyond Education

DEADLOCKS


Transaction 1
Transaction 2
Supplier
Holds
lock on Supplier
Part
Holds
lock on Part
blocked
Wants lock on Part
Wants lock on Supplier

PRIORITY INVERSION



MUTEXES

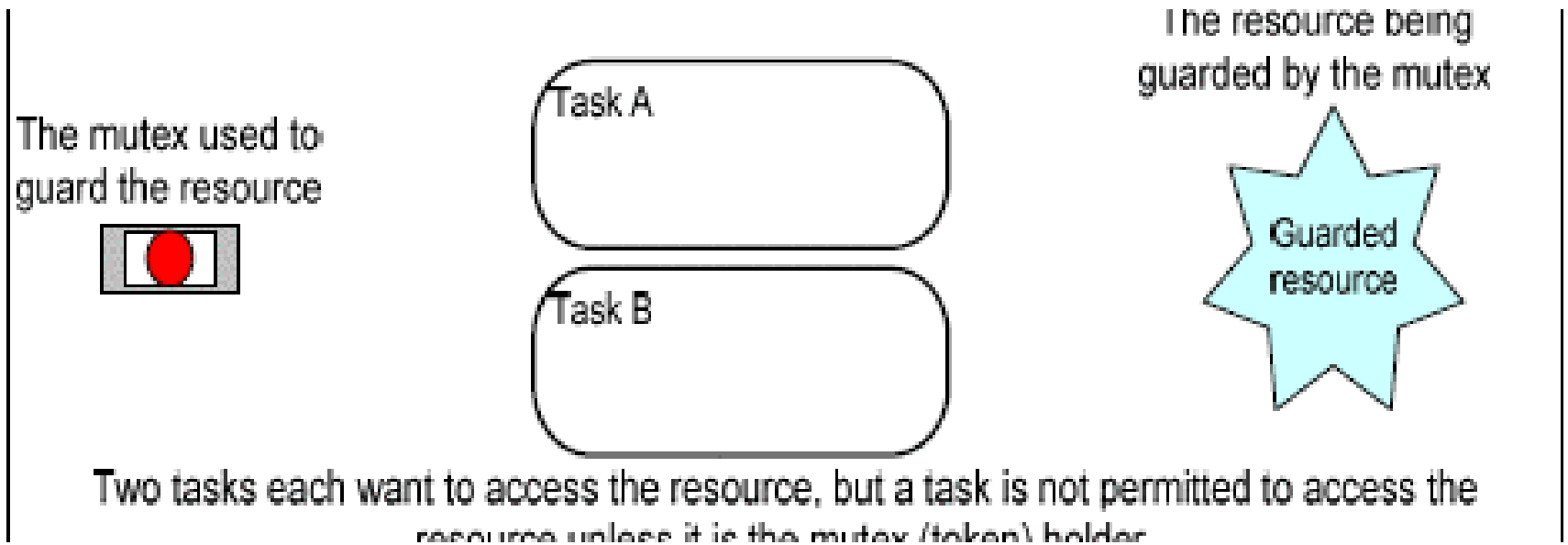
A Mutex is a special type of binary semaphore that is used to control access to a resource that is shared between two or more tasks. The word MUTEX originates from “MUTual EXclusion".

When used in a mutual exclusion scenario the mutex can be conceptually thought of as a token that is associated with the resource being shared. For a task to legitimately access the resource 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.

The primary difference is what happens to the semaphore after it has been obtained:

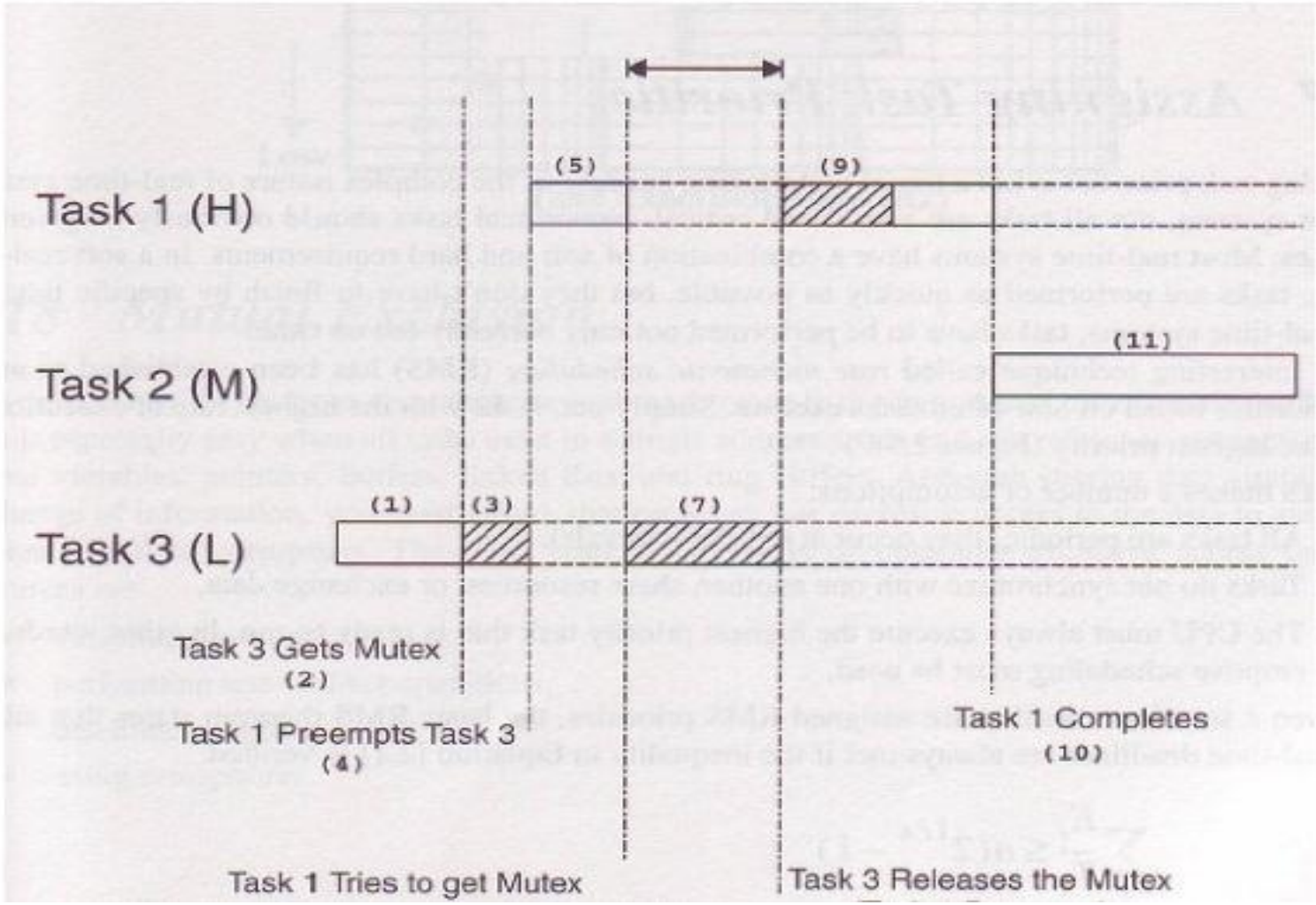
- A semaphore that is used for mutual exclusion must always be returned.
- A semaphore that is used for synchronization is normally discarded and not returned.
- MUTEX applies the mechanism of priority inheritance

MUTEXES



Two tasks each want to access the resource, but a task is not permitted to access the resource unless it is the mutex (token) holder

PRIORITY INHERITANCE



SEMAPHORE APIS

- xSemaphoreCreateBinary
- xSemaphoreCreateBinaryStatic
- xSemaphoreCreateCounting
- xSemaphoreCreateCountingStatic
- xSemaphoreCreateMutex
- xSemaphoreCreateMutexStatic
- xSemaphoreCreateRecursiveMutex
- xSemaphoreCreateRecursiveMutexStatic
- vSemaphoreDelete
- xSemaphoreGetMutexHolder
- xSemaphoreTake
- xSemaphoreTakeFromISR
- xSemaphoreTakeRecursive
- xSemaphoreGive
- xSemaphoreGiveRecursive
- xSemaphoreGiveFromISR
- uxSemaphoreGetCount

EXAMPLE USAGE

```
SemaphoreHandle_t xSemaphore = NULL;
/* A task that creates a semaphore. */
 void vATask ( void * pvParameters )
 {
        /* Create the semaphore to guard a shared resource. */
       xSemaphore = xSemaphoreCreateMutex ();
 }
 /* A task that uses the semaphore. */
void vAnotherTask ( void * pvParameters )
{
 /* ... Do other things. */
 if ( xSemaphore != NULL )
        {
        /* See if we can obtain the semaphore. If the semaphore is not available wait 10 ticks to see if it becomes free. */
        if ( xSemaphoreTake ( xSemaphore, ( TickType_t ) 10 ) == pdTRUE )
              {
                    /* We were able to obtain the semaphore and can now access the shared resource. */ /* ... */ /*
                     We have finished accessing the shared resource. Release the semaphore. */
                     xSemaphoreGive ( xSemaphore );
              }
        else
              {
                          /* We could not obtain the semaphore and can therefore not access the shared resource safely. */
               }
        }
 }
```

AMIT
Beyond Education

Let's code it!

- Port the FreeRTOS to ATMEGA32 and create a task that scan the keypad and if there is an input , send it to another task that shows the input character on the LCD after entering 8 digit password and pressing '#' the LCD task would signal a task that compare the inputs with the string "12345678" and if the comparison succeeded write on the LCD "access granted" otherwise write "access denied".

AMIT
Beyond Education