Hands on Workshop: Embedded RTOS Enabled Systems

Bryan Brauchler

Presenter title goes here
Second line title goes here

October 2019 | Session #AMF-AUT-T3883

SECURE CONNECTIONS FOR A SMARTER WORLD

Abstract:

Devices are constantly increasing in complexity and functionality by managing more resources, naturally resulting in a need for more sophisticated software architectures. One of such is the application of real time operating systems in embedded applications. This presentation outlines the basic usage of FreeRTOS for the S32K as will as the rudimentary concepts of operating system operation, scheduling, and resource management as it applies in the embedded environment. User applications can be written inside this environment to maximize the usage of hardware resources and prioritize operations based on their importance to the system.

Agenda

- Introduction to Embedded RTOS
- Sharing Limited Recourses Using an OS
- Tasks and Task Management
- Task Scheduling
- Using Shared Data
- Deadlock
- Application Hooks
- Timing and Software Timers

1. Introduction to Embedded RTOS

RTOS – Real Time Operating System

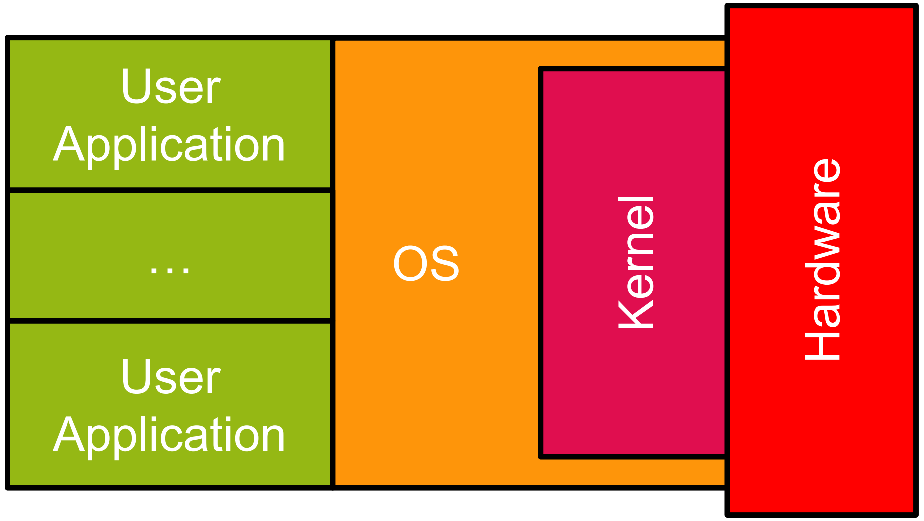
Purpose: Support a MCU's basic functions and provide a platform for applications to run on.

- Help to manage resources during runtime.
- Allows tasks and their data to be separate from other tasks.

- Real Time - Uses a scheduler that is **deterministic** to meet real time requirements.

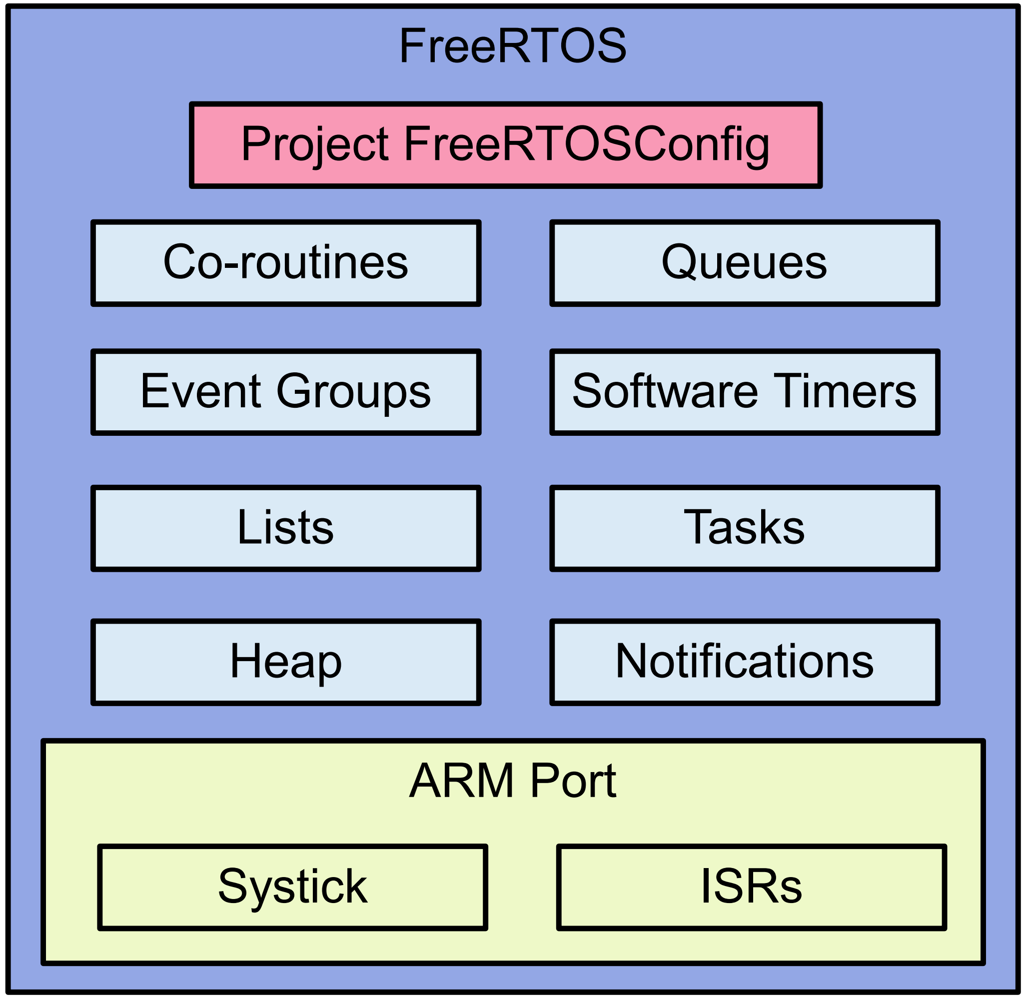
RTOS – Real Time Operating System

- Operating System – Software that manages the system resources and acts as an interface between the user and the hardware, allowing the user to execute programs conveniently and efficiently.
- Kernel – Core of an operating System, and is the first program loaded into memory, and remains the entire time the OS is running. The kernel interfaces the OS software to the hardware and manages processes, memory, and disks for the OS.

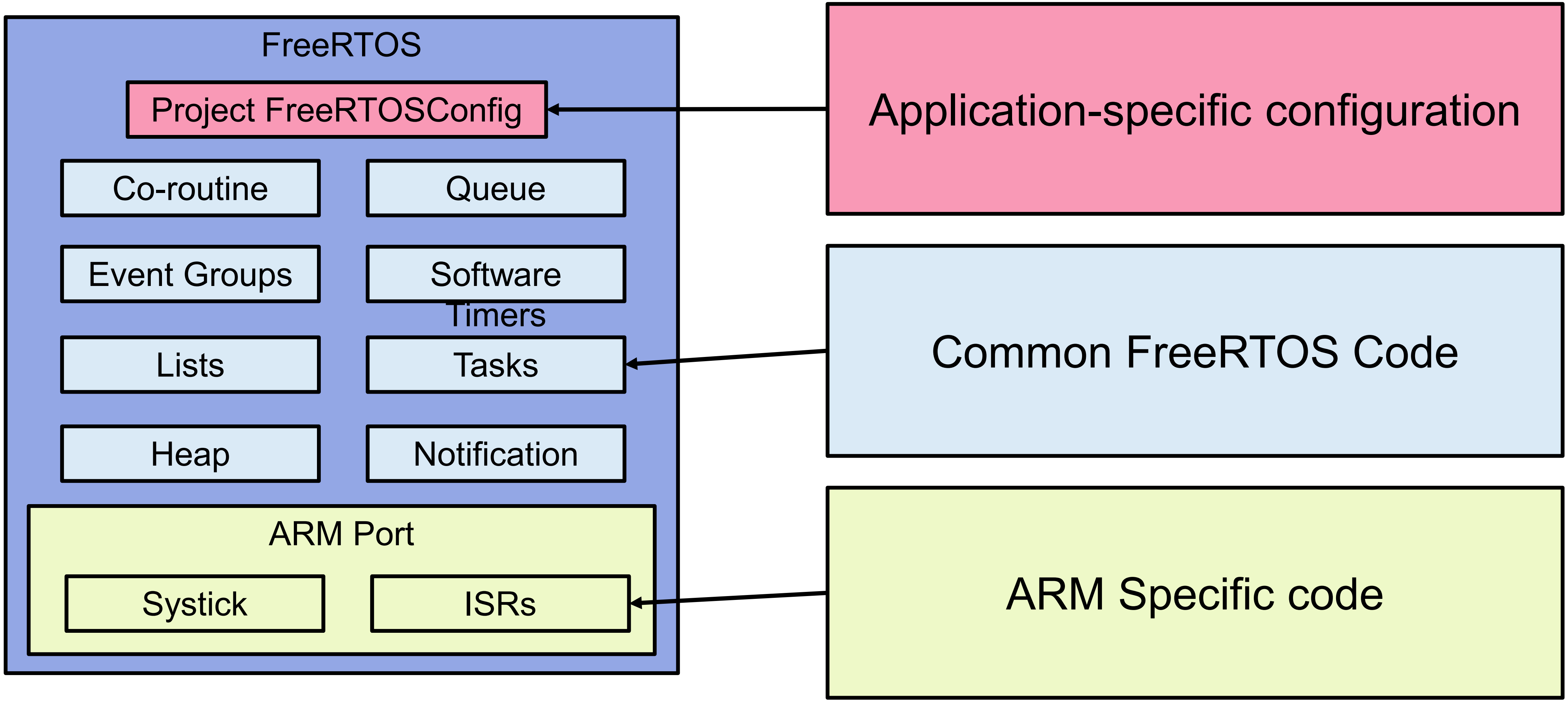


Features – FreeRTOS

- Scheduler
- Tasks with multiple priority lists
- Dynamic memory (heap)
- Pre-emptive or co-operative operation
- Very flexible task priority assignment
- Message queue
- Software timer
- Semaphore and Mutex
- Tick hook functions
- Idle hook functions
- Stack overflow checking
- Tick less idle mode
- Flexible, fast and light weight task notification mechanism



Features – FreeRTOS



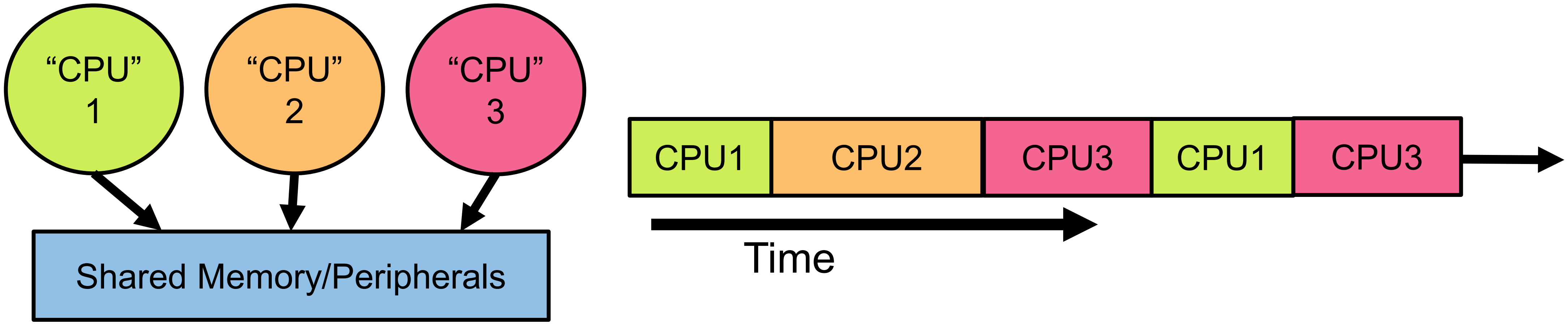
FreeRTOS Kernel – Philosophy

- Small Kernel, **implemented in** C*, compiled and linked with application
- Kernel configuration with #define in **FreeRTOSConfig.h**
- Kernel only needs **tick interrupt** and **software interrupt**
- Scheduler variables and task stack in dynamic memory (**heap**)
- Multiple tasks with same priority
- Minimal overhead with large scalability

2. Sharing Limited Resources

Sharing Limited Resources

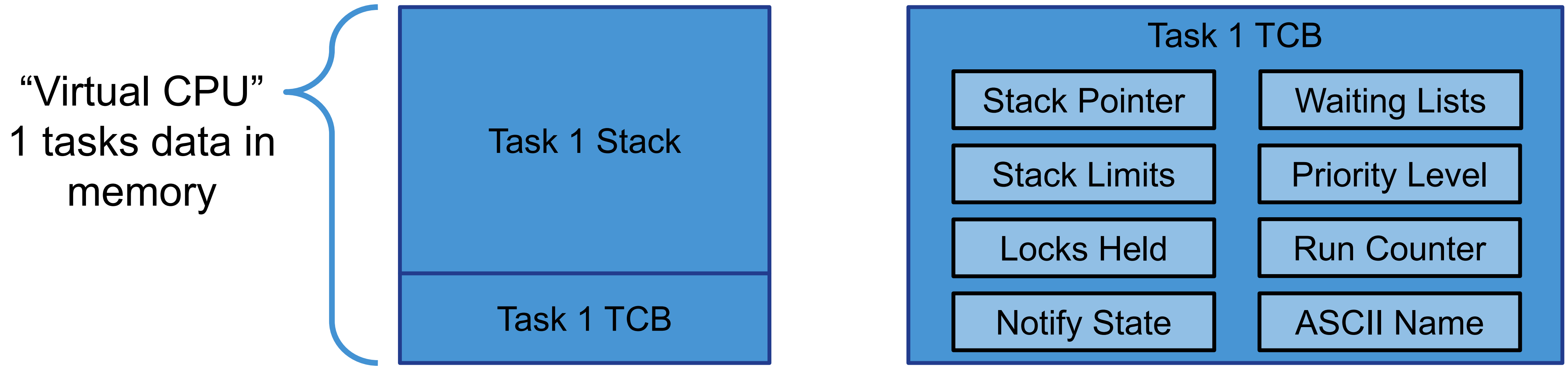
- Each process is sharing
- The OS gives the illusion of exclusive CPU access to every task that is running
- Done by switching between virtual “CPU” configurations in time



Sharing Limited Resources

- Even though resources are limited, the OS is designed to give all tasks access to the entire CPU

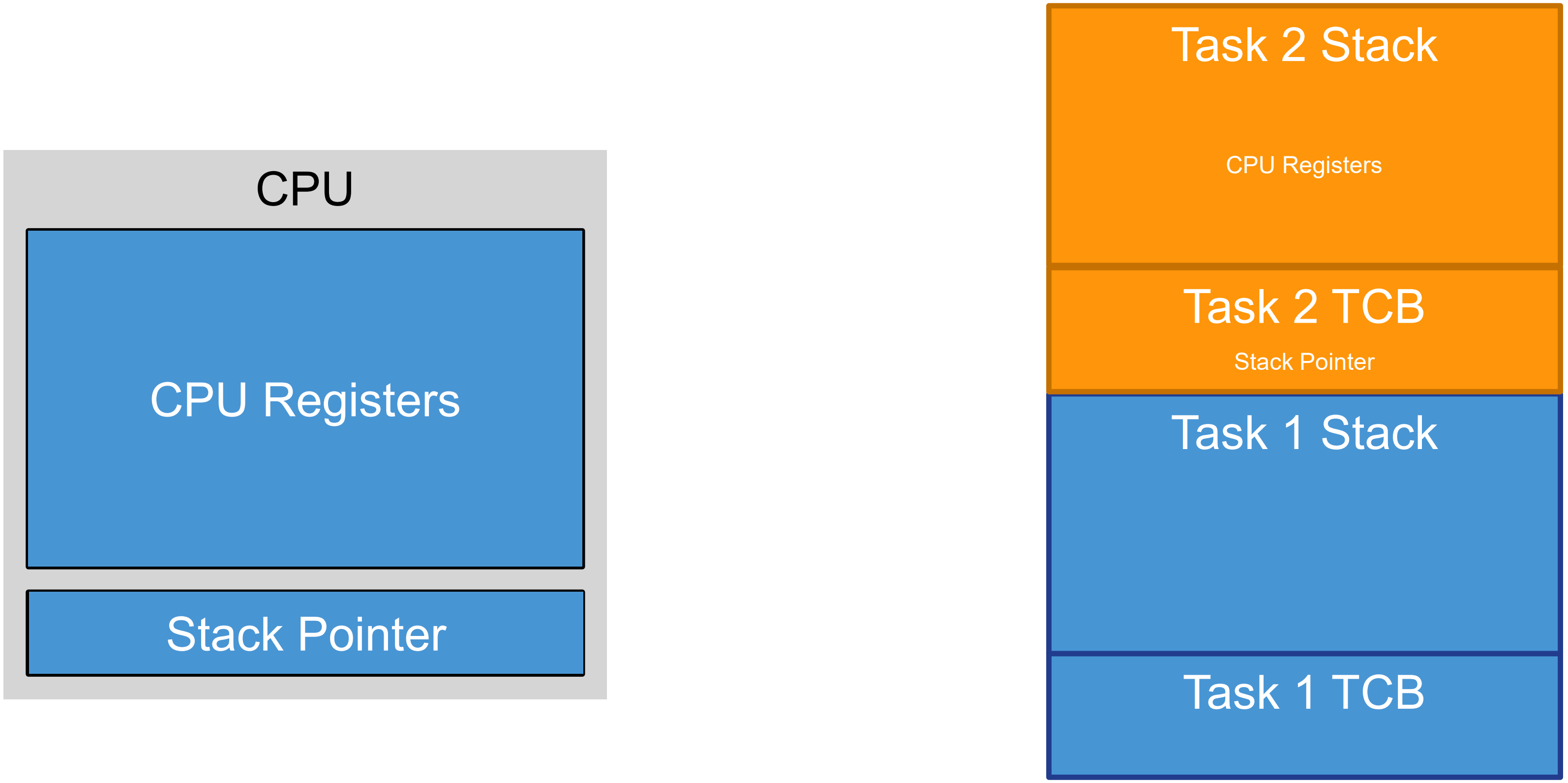
- Tasks have their own current state, Set of processer flags, set of CPU registers, stack, and control block.



Context Switching (ARM)

- On entry to the interrupt handler some processer registers are stacked
- Scheduler determines a context switch is required
- Remaining CPU registers are stacked onto the processes stack
- Stack pointer is saved to the TCB
- Stack pointer of new task is set
- CPU registers are unstacked for the new process
- Control is given to the new process to run

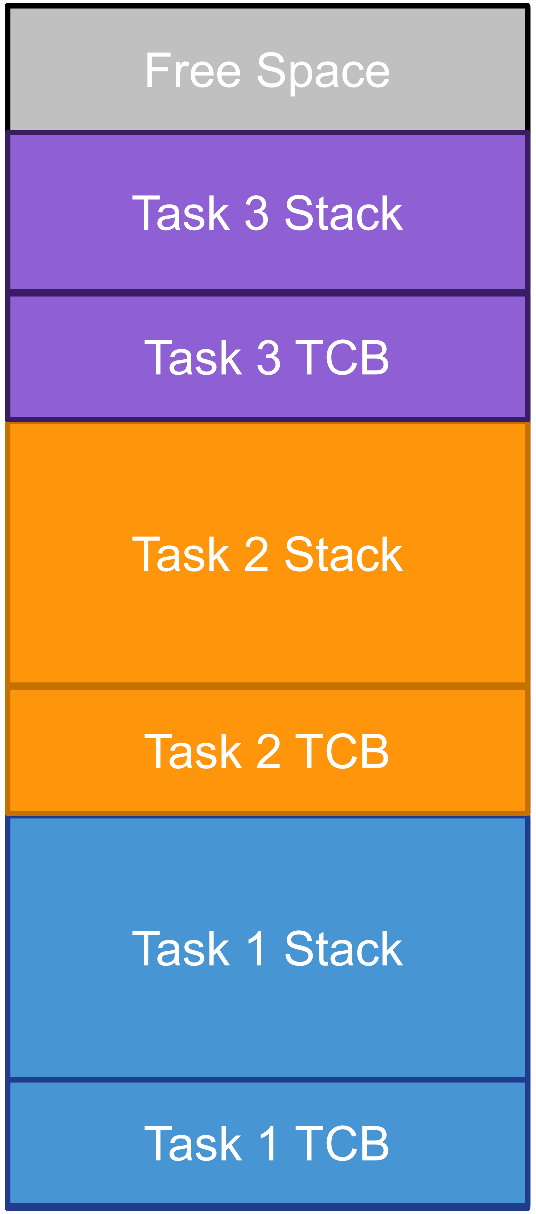
Context Switching (ARM)

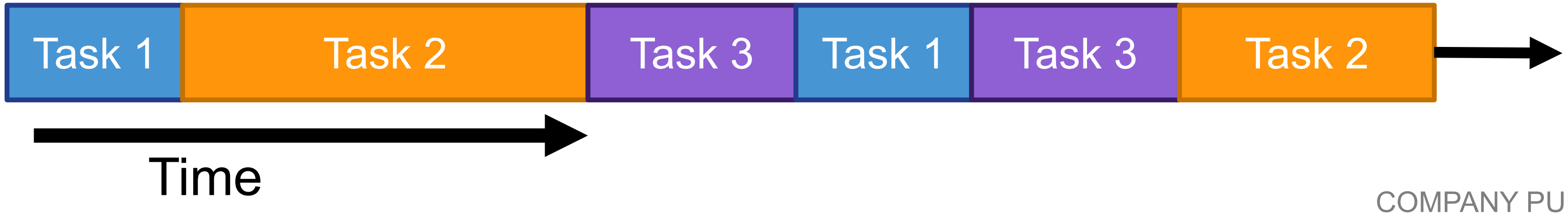


3. Tasks and Task Management

Tasks/Threads

- Created with **xTaskCreate()**
- Allocates space for Task Control Block (TCB) and a task stack
- This task will be ready to run immediately and scheduled according to the scheduling policy



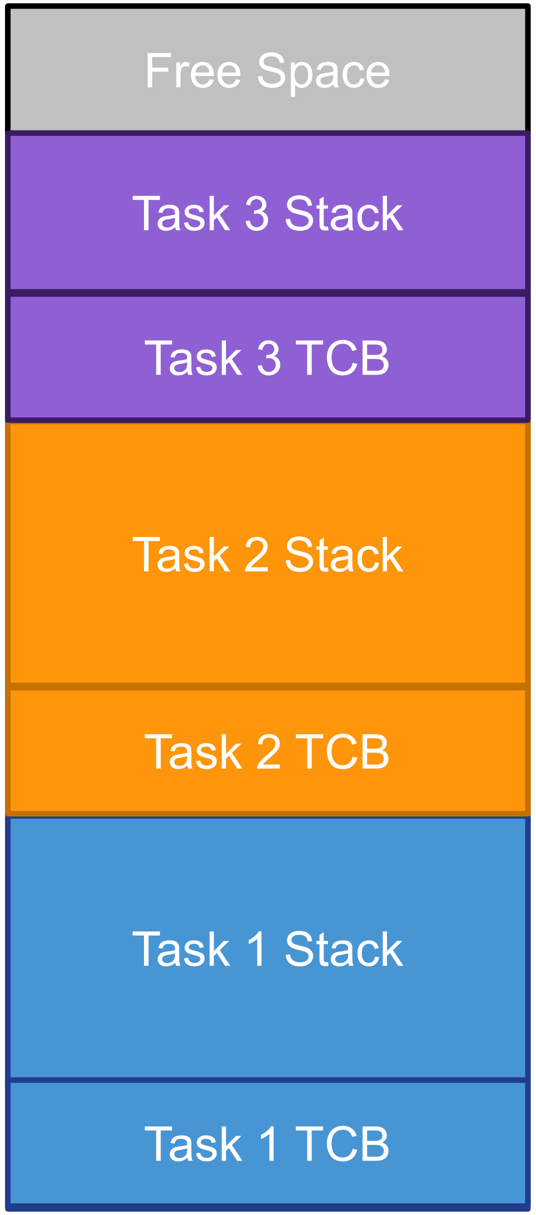


TCB – Task Control Block

Used to keep track of task data

- Stack pointer
- Runtime
- Task Priority
- Resources held by the task
- And more

Internal to FreeRTOS API



Task Management APIs

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

void vTaskDelete( TaskHandle_t xTask );

void vTaskDelay( const TickType_t xTicksToDelay );

void vTaskPrioritySet( TaskHandle_t xTask,
                       UBaseType_t uxNewPriority );

void vTaskSuspend( TaskHandle_t xTaskToSuspend );

void vTaskResume( TaskHandle_t xTaskToResume );
```

API Documentation: https://freertos.org/a00106.html

FreeRTOS API Conventions

- API functions are prefixed with their return type
  - **U** – Unsigned
  - **L** – Long
  - **S** – Short
  - **C** – Char
  - **P** – Pointer
  - **X** – Non-stdint variables or size_t
  - **E** – Enumerated variables

- For example: The prefix ***ul*** would refer to a function that returns an **unsigned long**

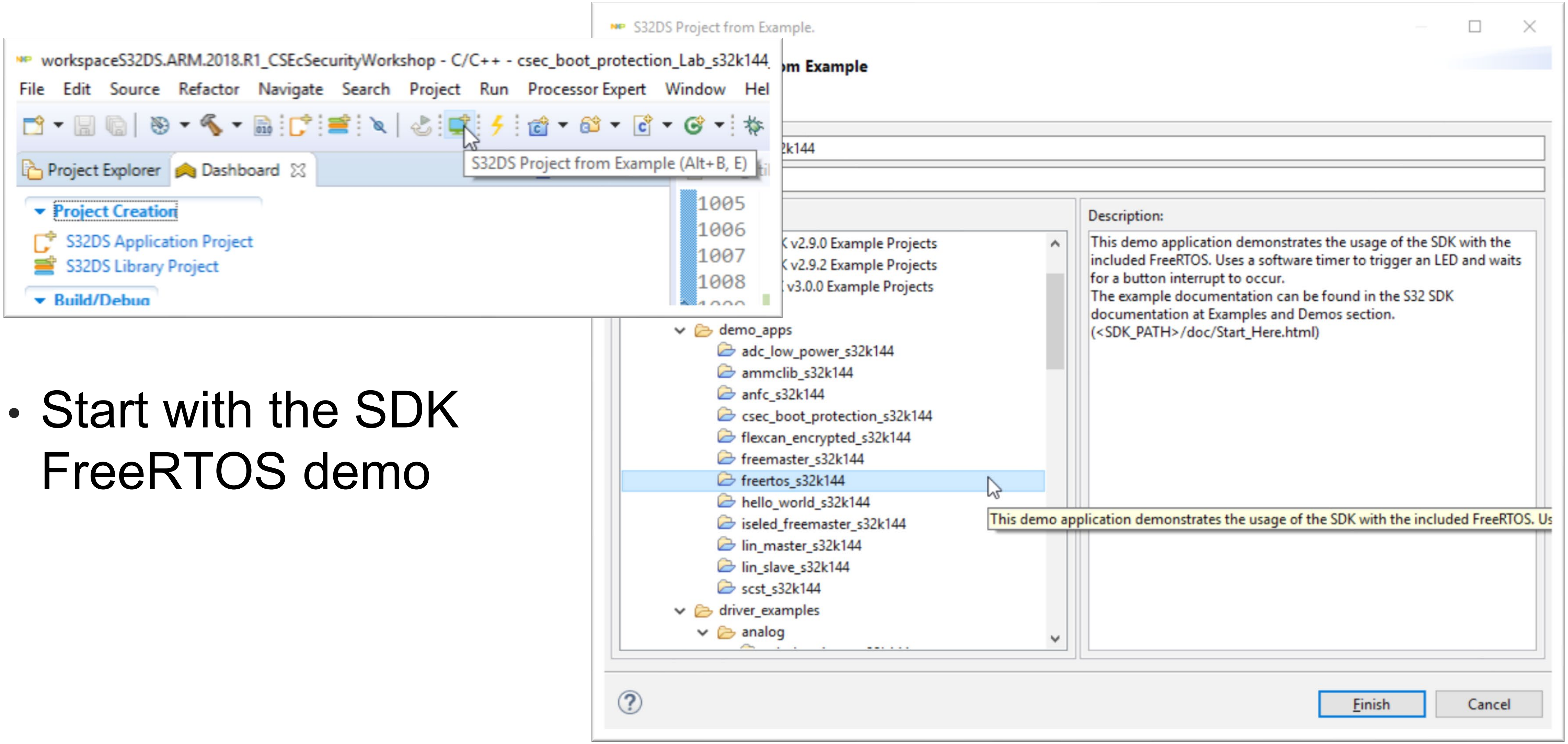
Lab 1: Multitasking with FreeRTOS

Purpose:

- Run a simple application to see the OS running and the scheduler.
- View debug information about running tasks.
- Watch task states change in different sections of the code.

Tasks can be written to take care of only 1 job all tasks will look as if they are all running at once.

Lab 1: Multitasking with FreeRTOS



- Start with the SDK FreeRTOS demo

Lab 1: Customizing rtos.c

1. Modify includes and definitions at the top of the file.

```
#include "Cpu.h"
#include "LCD.h"
#include "NXP_logo.h"

#define mainLCD_INT_TASK_PRIORITY( tskIDLE_PRIORITY + 3 )
#define mainLCD_TASK_PRIORITY( tskIDLE_PRIORITY + 3 )
#define mainAPP_TASK_PRIORITY( tskIDLE_PRIORITY + 1 )

//This should be at least 40 times the number of tasks running
#define STATISTICS_PC_BUFFER_LENGTH(256)
#define STATISRICS_TASK_STACK_SIZE(STATISTICS_PC_BUFFER_LENGTH + configMINIMAL_STACK_SIZE)
```

Lab 1: Customizing rtos.c

2. Write hardware configuration code in prvSetupHardware

```
static void prvSetupHardware( void ) {

    /* Initialize and configure clocks
     *  -   Setup system clocks, dividers
     *  -   see clock manager component for more details
     */
    CLOCK_SYS_Init(g_clockManConfigsArr, CLOCK_MANAGER_CONFIG_CNT,
                   g_clockManCallbacksArr, CLOCK_MANAGER_CALLBACK_CNT);
    CLOCK_SYS_UpdateConfiguration(1U, CLOCK_MANAGER_POLICY_AGREEMENT);

    /* Set the run more to HSRUN to get a 112MHz clock going */
    POWER_SYS_Init(&powerConfigsArr, POWER_MANAGER_CONFIG_CNT,
       &powerStaticCallbacksConfigsArr, POWER_MANAGER_CALLBACK_CNT);
    POWER_SYS_SetMode(1U, POWER_MANAGER_POLICY_AGREEMENT);

    /* Initialize the pins according to the pin_mux module */
    PINS_DRV_Init(NUM_OF_CONFIGURED_PINS, g_pin_mux_InitConfigArr);
}
```

Lab 1: Customizing rtos.c

3. Callbacks to display runtime information on the LCD

```
static void vTimer_callback_display_statistics(TimerHandle_t xTimer) {
    /* Validate the timer */
    configASSERT( xTimer );

    TaskHandle_t displayTaskHandle = (TaskHandle_t)pvTimerGetTimerID( xTimer );

    xTaskNotify(displayTaskHandle, 0, eNoAction);
}

/*-----------------------------------------------------------------------------------------*/

static void task_display_statistics( void *pvParameters ) {
    uint8_t buff[STATISTICS_PC_BUFFER_LENGTH];
    for (;;) {
        xTaskNotifyWait(pdFALSE, pdFALSE, NULL, portMAX_DELAY);

        //get runtime stats
        vTaskGetRunTimeStats(buff);

        //update LCD
        LCD_DrawWrappedString(0, 0, buff, WHITE, BLACK, 1);
    }
}
```

Lab 1: Customizing rtos.c

4. Create a task that's purpose is to initialize the LCD screen

```
static void task_initalize_screen( void *pvParameters ) {
    /* init the display */
    LCD_InitDisplay();

    /* draw NXP logo */
    LCD_DrawImage(TFTHEIGHT-200, TFTWIDTH-80, 200, 80, NXP_logo_bytes);

    /* Start the service task to print out information about the OS on the LCD screen */
    TaskHandle_t displayTaskHandle;
    TimerHandle_t statsTimerHandle;
    xTaskCreate( task_display_statistics, "LCD Stats", 3*configMINIMAL_STACK_SIZE, NULL, mainLCD_TASK_PRIORITY, &displayTaskHandle);

    /* Create a timer to periodically signal processing for the display.
     * 5 second period.
     * Automatically reloaded.
     * The associated task handle will be used as the id of the timer. */
    statsTimerHandle = xTimerCreate("LCD Timer", pdMS_TO_TICKS(1000), pdTRUE, displayTaskHandle,
vTimer_callback_display_statistics);
    xTimerStart(statsTimerHandle, 0);

    /* after running code for the display this process exits and
     * deletes itself from all running queues. */
    vTaskDelete(NULL);
}
```

Lab 1: Customizing rtos.c

5. Blink the red and the blue LEDs independently in different tasks written just like normal C functions.

```
static void task_blink_red_led( void *pvParameters ) {
    for (;;) {
        /* wait approximently one second
         * The ticks can be delayed slightly by interrupts and higher priority
         * tasks so this is not a good method to wait a specific amount of time*/
        vTaskDelay(pdMS_TO_TICKS(1000));

        /* Toggle the red led */
        PINS_DRV_TogglePins(LED_RED_PORT, 1 << LED_RED_PIN);
    }
}

/*-----------------------------------------------------------------------------------*/

static void task_blink_blue_led( void *pvParameters ) {
    /* move the two blinking lights slightly out of sync */
    vTaskDelay(pdMS_TO_TICKS(500));
    for (;;) {
        /* wait approximently one second
         * The ticks can be delayed slightly by interrupts and higher priority
         * tasks so this is not a good method to wait a specific amount of time*/
        vTaskDelay(pdMS_TO_TICKS(1000));

        /* Toggle the blue led */
        PINS_DRV_TogglePins(LED_BLUE_PORT, 1 << LED_BLUE_PIN);
    }
}
```

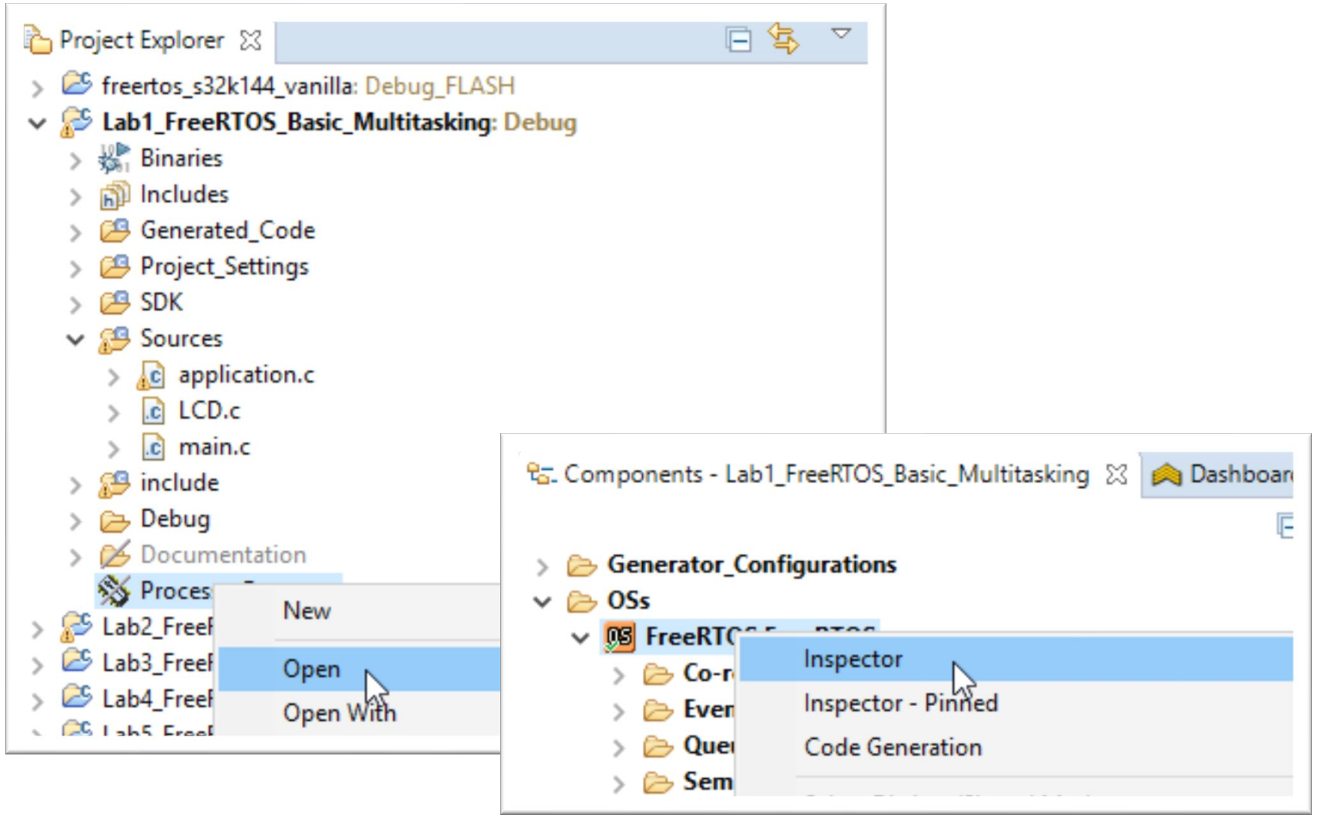
Lab 1: Customizing rtos.c

6. Write the rtos_start function that will be called by main to start the scheduler.

```
void rtos_start( void ) {
    /* Configure the NVIC, LED outputs and button inputs. */
    prvSetupHardware();

    /* Start the two tasks as described in the comments at the top of this
    file. */

    xTaskCreate( task_initalize_screen, "LCD Init", configMINIMAL_STACK_SIZE, NULL, mainLCD_INT_TASK_PRIORITY, NULL );

    /* create tasks to toggle the different LEDs. */
//  xTaskCreate( task_blink_red_led, "RED LED Task", configMINIMAL_STACK_SIZE, NULL, 1, NULL);
//  xTaskCreate( task_blink_blue_led, "BLUE LED Task", configMINIMAL_STACK_SIZE, NULL, 1, NULL);

    /* Start the tasks and timer running. */
    vTaskStartScheduler();

    /* If all is well, the scheduler will now be running, and the following line
    will never be reached.  If the following line does execute, then there was
    insufficient FreeRTOS heap memory available for the idle and/or timer tasks
    to be created.  See the memory management section on the FreeRTOS web site
    for more details. */
    for( ;; );
}
```

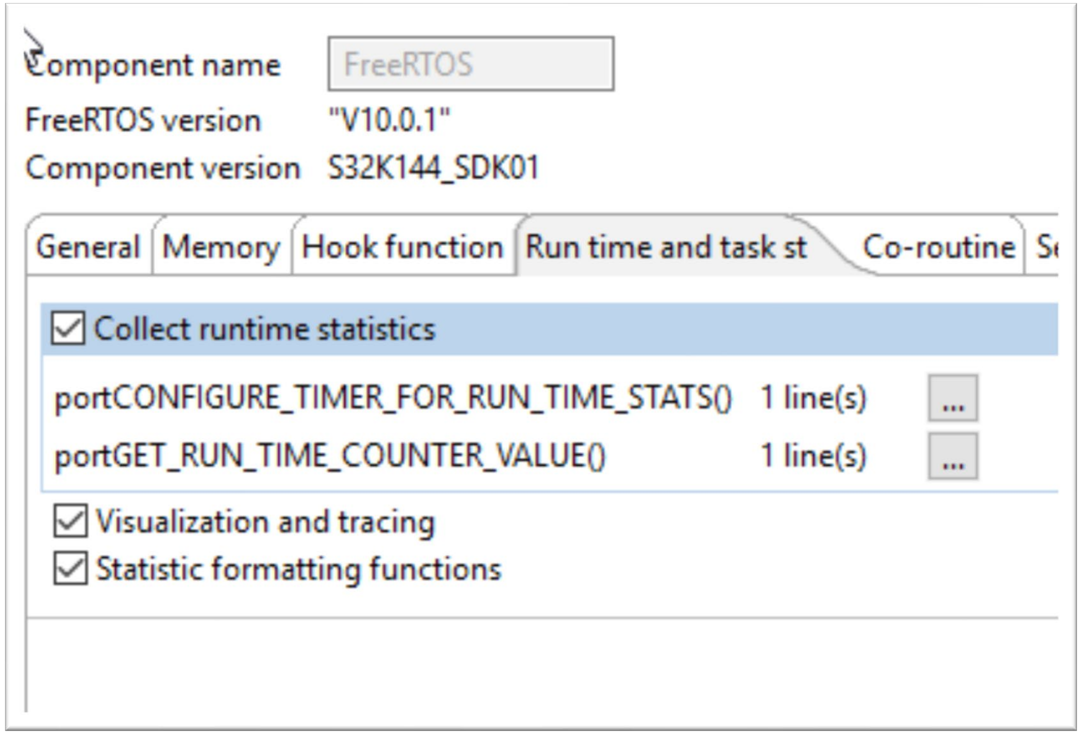
Lab 1: Customizing FreeRTOS Settings

- S32 Design studio allows you to modify all OS settings using Processor Expert



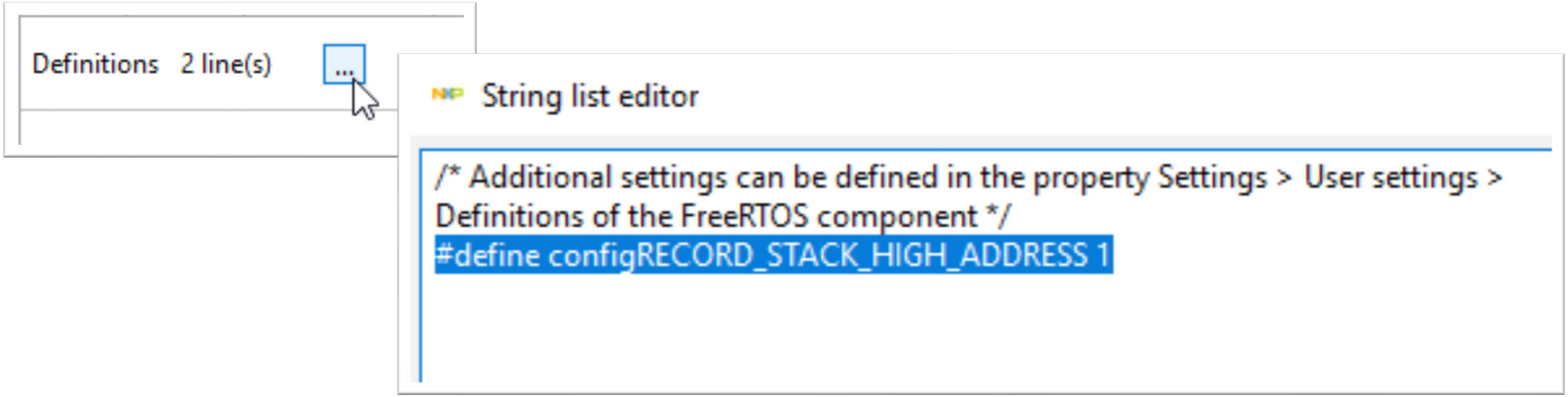
Lab 1: Customizing FreeRTOS Settings

- Enable generation of run time debug information



Lab 1: Customizing FreeRTOS Settings

- Under the “User settings” tab, add another debug macro




```
#define configRECORD_STACK_HIGH_ADDRESS 1
```

Lab 1: Multitasking with FreeRTOS

Running the task shows both the LEDs blinking and the LCD displaying information about the different tasks. Using the FreeRTOS aware features in S32 Design Studio, one can peek into the different elements of the Operating System.

TCB#	Task Name	Task Handle	Task State	Priority	Stack Usage	Event Ob...	Runtime
1	LCD Init	0x20000208	Suspended	3 (3)	340 B / 392 B		0x1d3 (9.1%)
2	IDLE	0x20000408	Running	0 (0)	72 B / 392 B		0xe14 (70.1%)
3	Tmr Svc	0x20000770	Blocked	3 (3)	264 B / 504 B	Unknown ...	0x17f (7.5%)
4	RED LED Tas	0x200009f0	Blocked	1 (1)	96 B / 392 B		0x0 (0.0%)
5	BLUE LED Ta	0x20000bf0	Blocked	1 (1)	96 B / 392 B		0x0 (0.0%)
6	LCD Stats	0x20001110	Suspended	3 (3)	1.07 kB / 1.16 kB		0x2ad (13.3%)

Lab 1: Multitasking with FreeRTOS

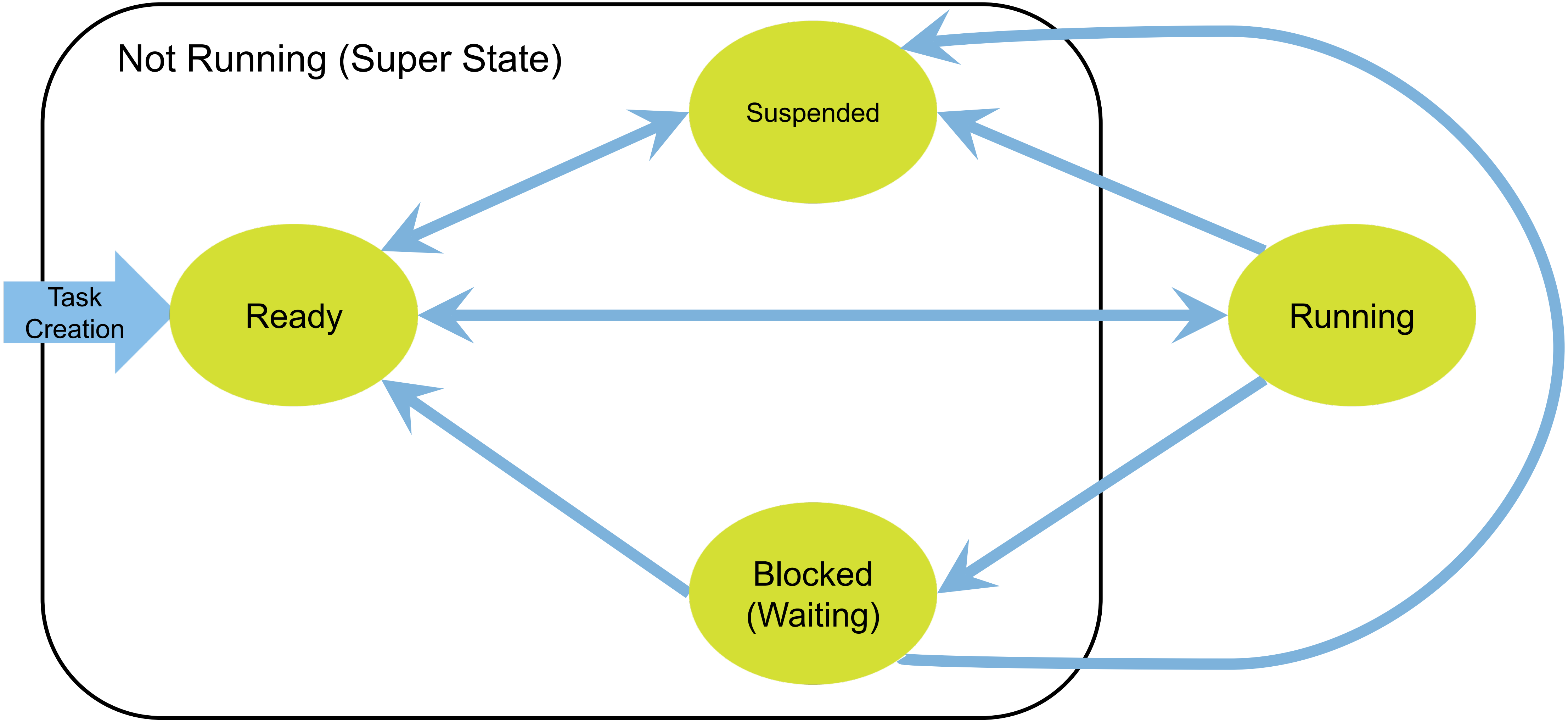
TCB#	Task Name	Task Handle	Task State	Priority	Stack Usage	Event Ob...	Runtime
1	LCD Init	0x20000208	Suspended	3 (3)	340 B / 392 B		0x1d3 (9.1%)
2	IDLE	0x20000408	Running	0 (0)	72 B / 392 B		0xe14 (70.1%)
3	Tmr Svc	0x20000770	Blocked	3 (3)	264 B / 504 B	Unknown ...	0x17f (7.5%)
4	RED LED Tas	0x200009f0	Blocked	1 (1)	96 B / 392 B		0x0 (0.0%)
5	BLUE LED Ta	0x20000bf0	Blocked	1 (1)	96 B / 392 B		0x0 (0.0%)
6	LCD Stats	0x20001110	Suspended	3 (3)	1.07 kB / 1.16 kB		0x2ad (13.3%)

Some observations we can see from the above

- The Idle task runs the majority of the time.
- The stacks of “LCD Init” and “LCD Stats” are very full, and may overflow if more is stacked during their runtime.
- There is very little overhead to run the LEDs.

4. Task Scheduling

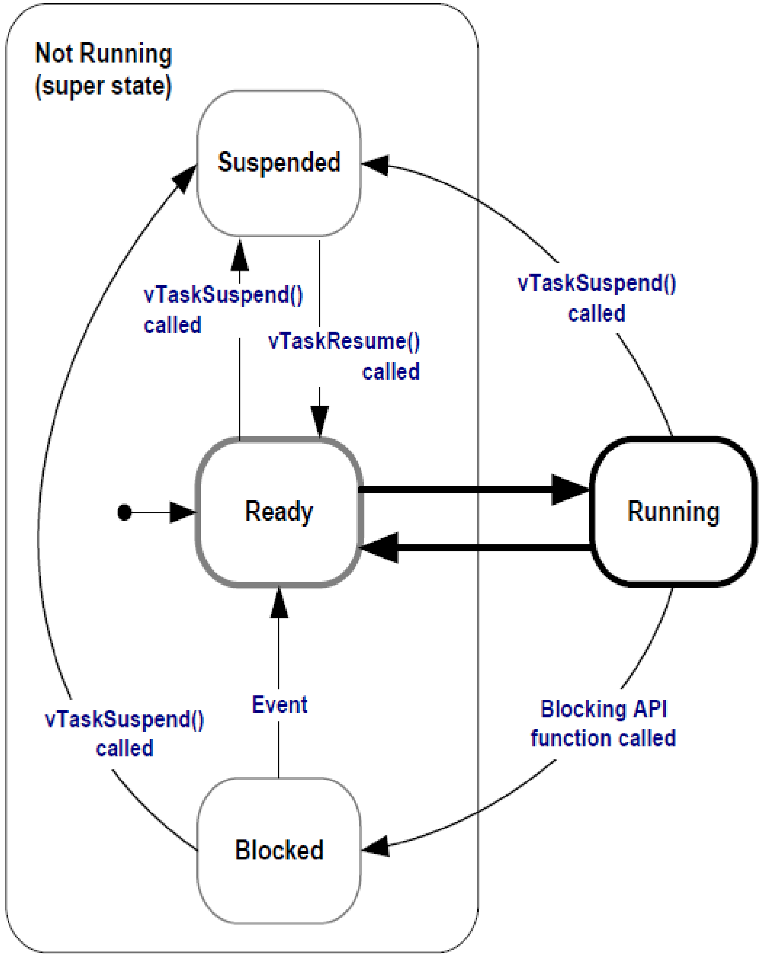
Task States and Transitions



Task State

Running	Given to tasks when they are actively executing their code. The task that has active control over the processer.
Ready	Task state that indicates the task is ready to run. This task is one of the possible choices for the scheduler when picking which task to run.
Blocked (Waiting)	Run state of a task that has purposely given up control in order to wait for some event (Timing, I/O, Other tasks). Not available for the scheduler to pick to run.
Suspended	Similar to Blocked, except that the task is disabled indefinitely. The only way for a suspended task to re-enter the ready queue is explicitly resuming the suspended task

State Transitions



- A task can be in different states during its lifetime
- Only one task can be running on a core at a time
- OS function calls, OS Events, and Hardware Interrupts can cause a state task state transition

Scheduler Policies

Non-Preemptive

- Tasks run to completion then return control to the kernel

Pros:

- Allows more predictable task lengths
- Less scheduler overhead

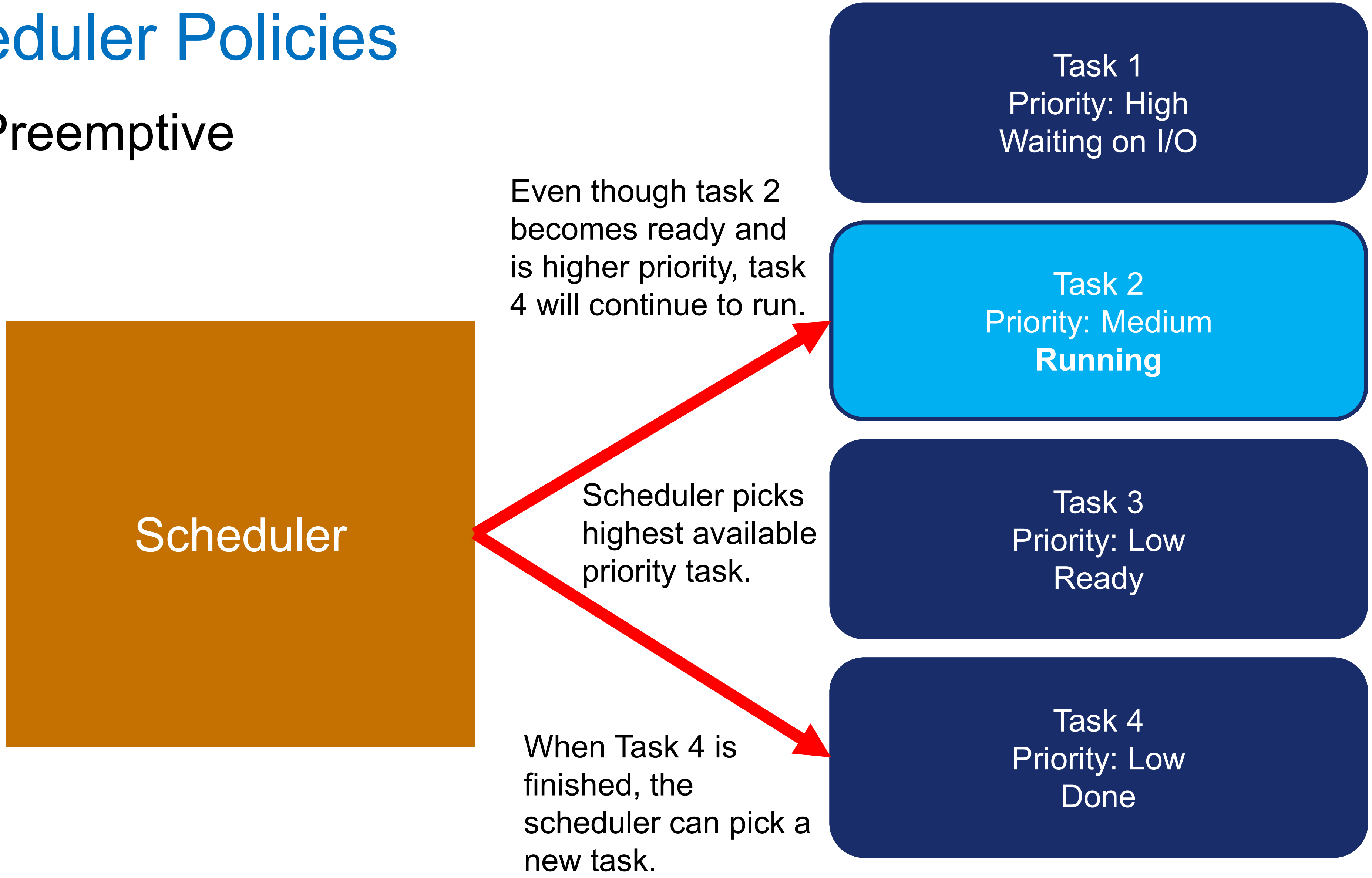
Cons:

- May block a higher priority event
- Longer tasks may hog the CPU

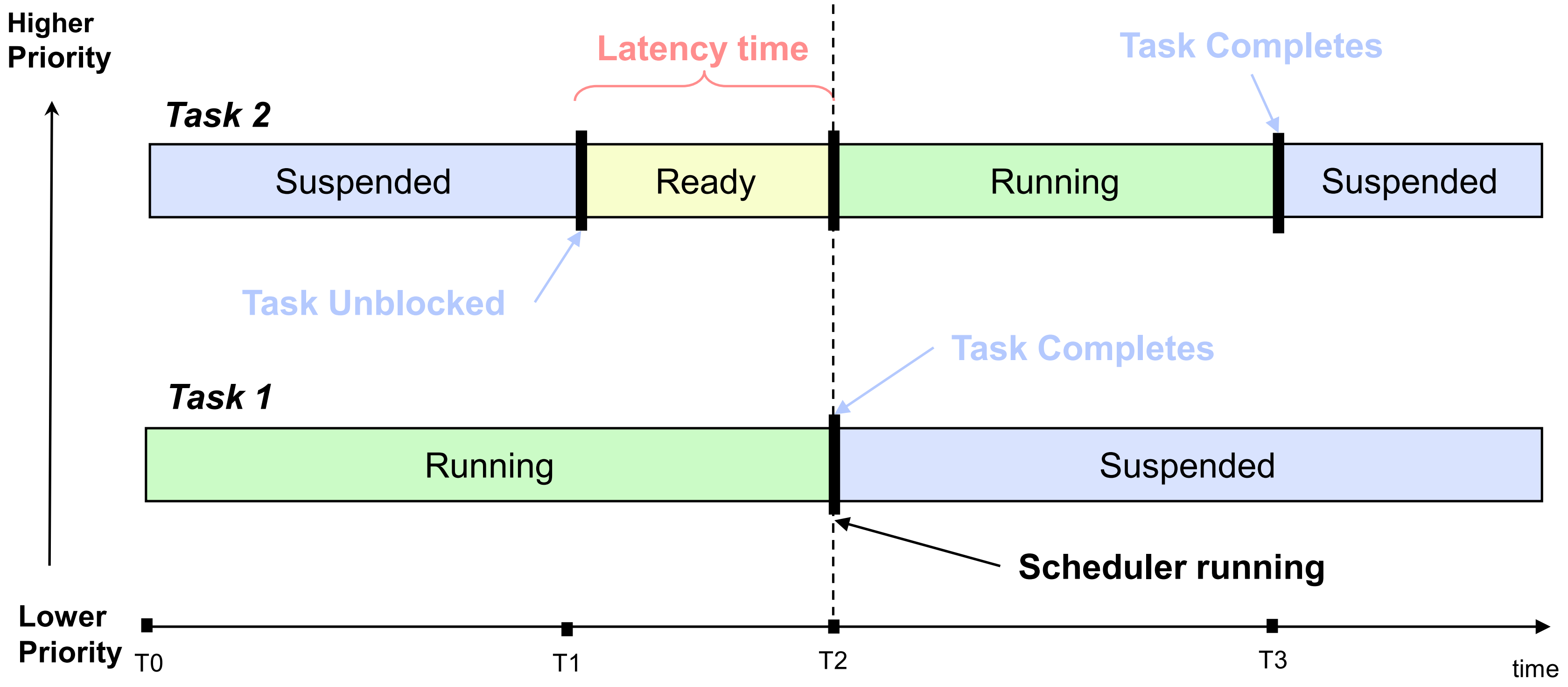
Similar to using a rental car. When the done, control returns to the rental company (scheduler) and is given to a new user (task)

Scheduler Policies

Non-Preemptive



Non-Preemptive Scheduling



Scheduler Policies

Preemptive

- The kernel can forcibly take control away from a task to allow another higher priority task to run

Pros:

- High priority tasks run immediately.
- Makes tasks feel more responsive

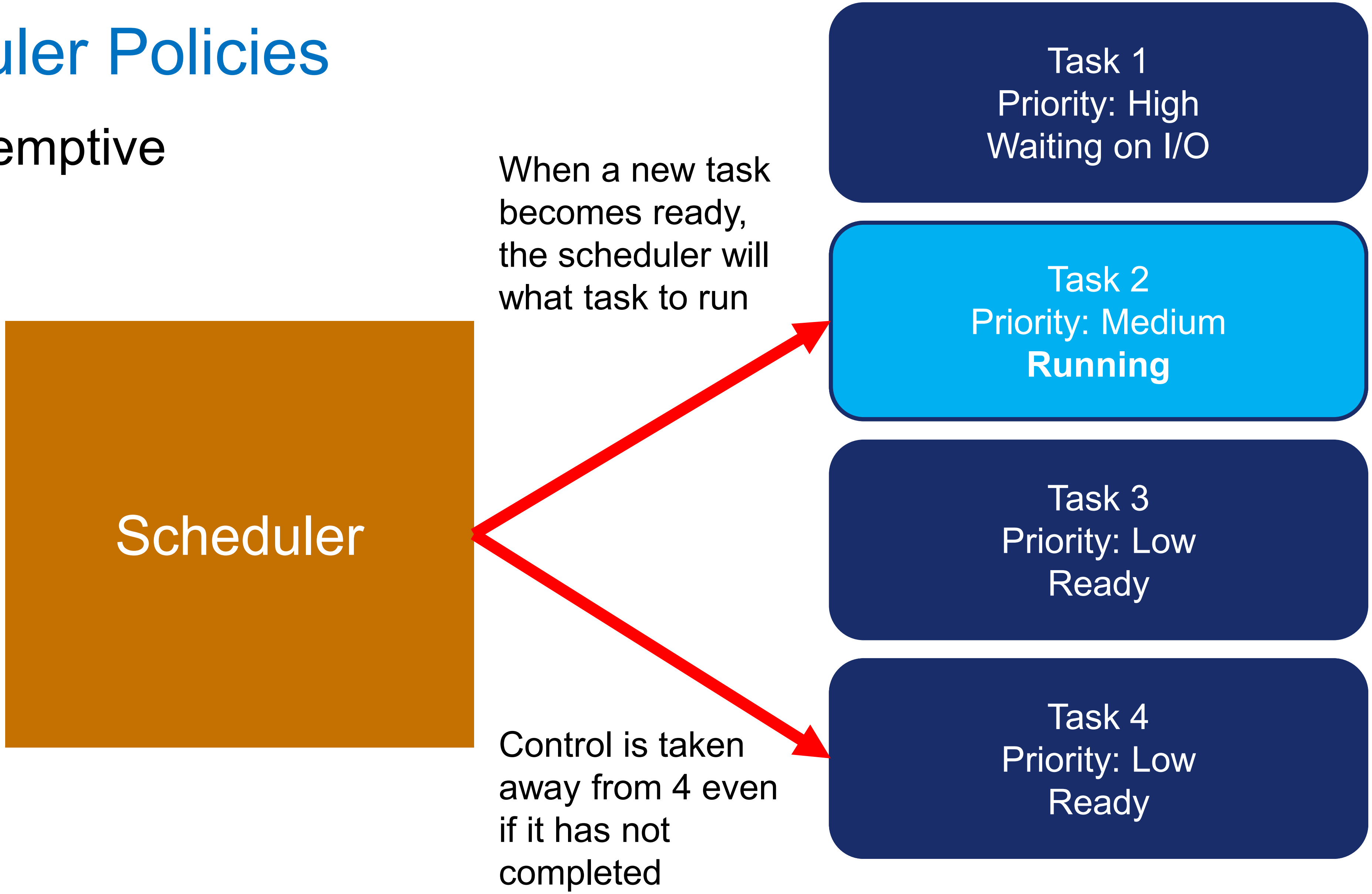
Cons:

- Tasks are able to be interrupted and stopped.
- More overhead due to more task switching

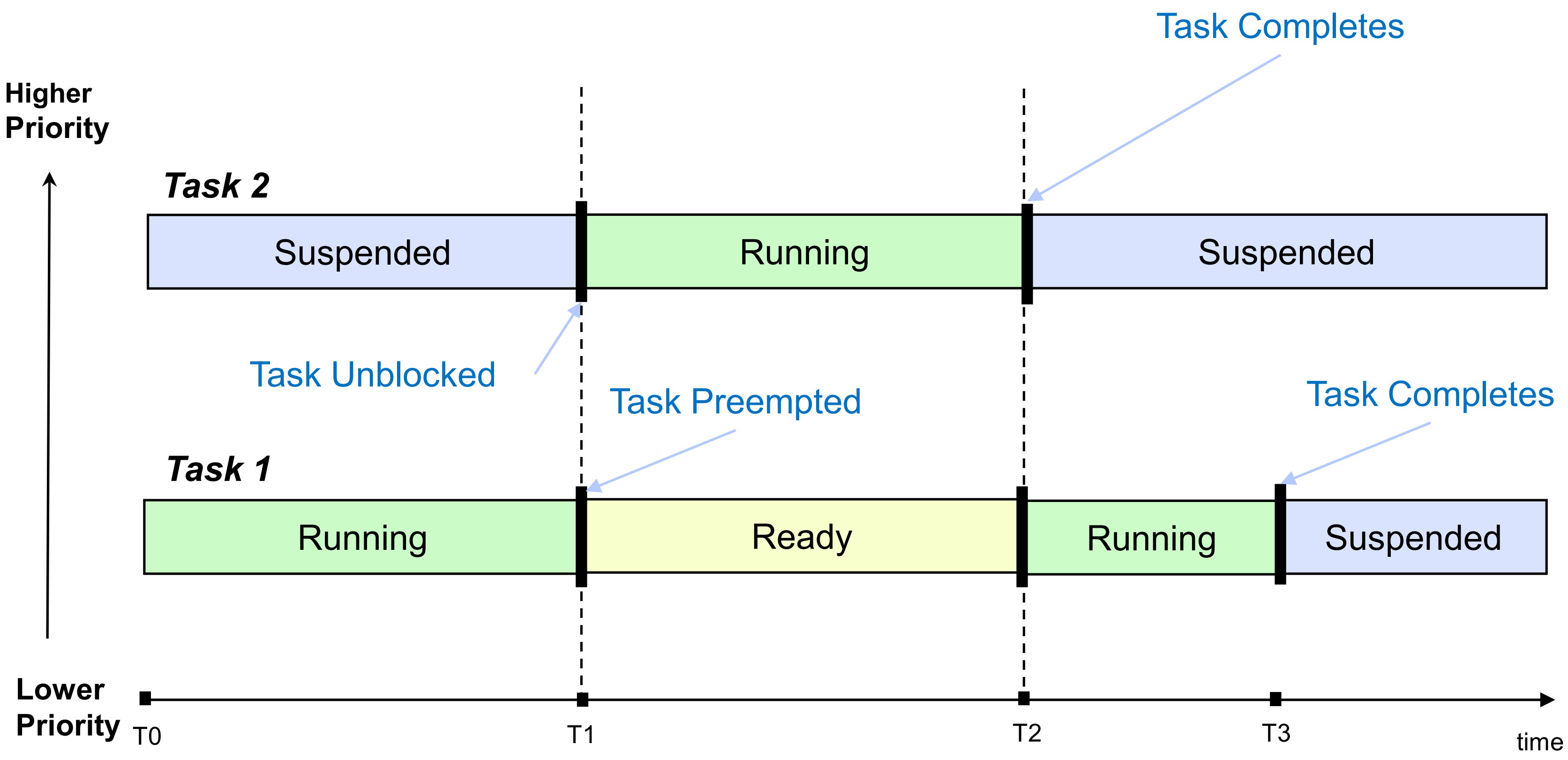
When your kids all want to use the Xbox and you, the parentm must make decisions who gets to play at a given time.

Scheduler Policies

Non-Preemptive

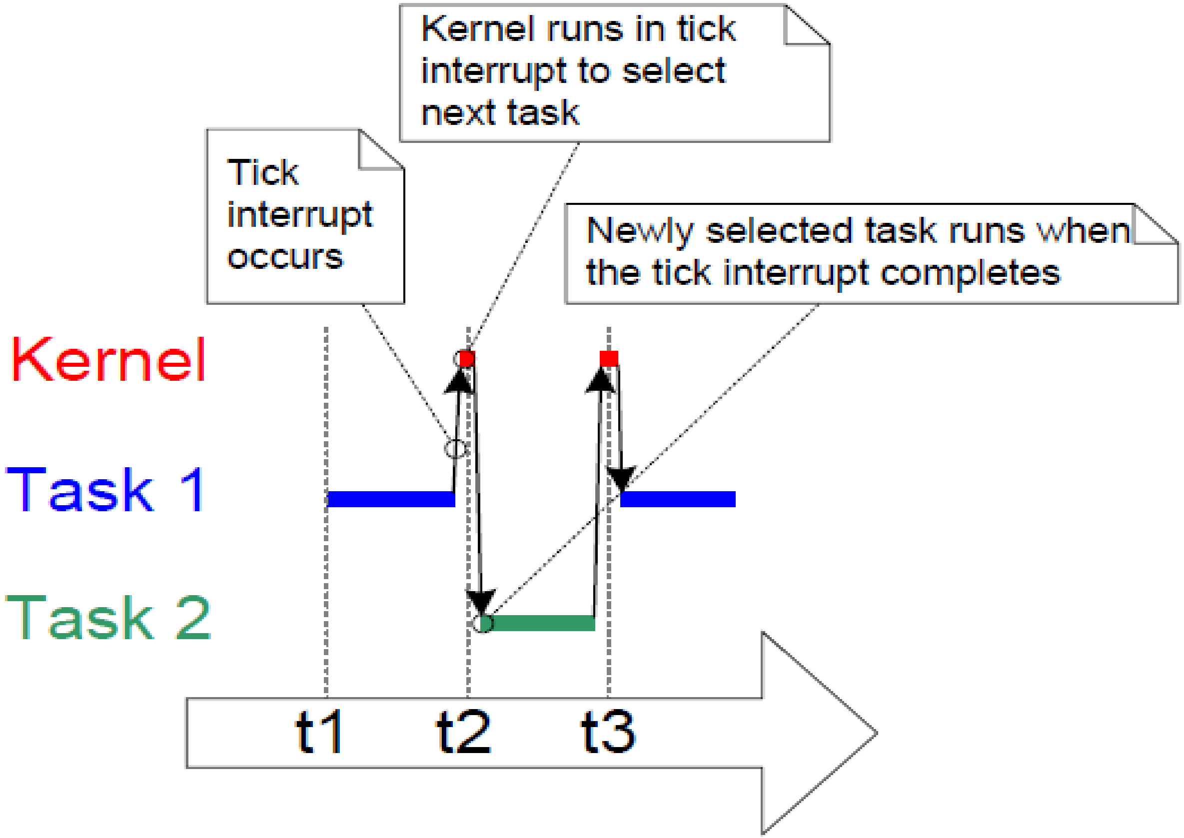


Preemptive Scheduling



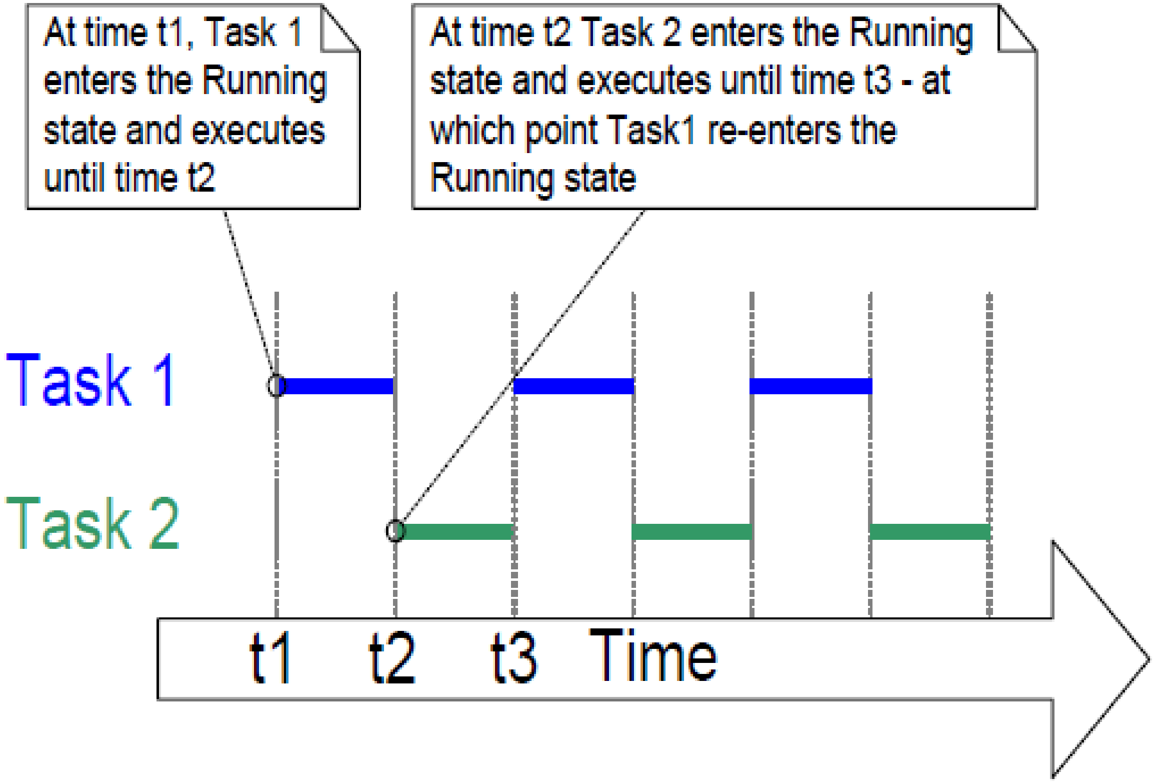
Tick Interrupt

- Configurable periodic interrupt that allows the Kernel to run
- Used for timing and scheduling in preemptive scheduling algorithm
- User code can be inserted in a hook if there are other things that should happen every tick in their design



Example

Two tasks are running at the same priority, with a preemptive scheduling algorithm with time sharing enabled



Task Priorities

- Higher Number = Higher Priority
- Assigned on creation of the task
- Can be changed by API calls
- Lower priority tasks gets preempted by higher priority tasks.
- **“vTaskPrioritySet()”** API function can be used to change the priority of any task after the scheduler has been started.

Task Priorities – Caution

What happens when a high priority task is constantly doing work?

- Task Starvation – Lower priority tasks will not get a chance to run
- High priority tasks must have time they are blocked or that they yield to allow lower priority tasks to run

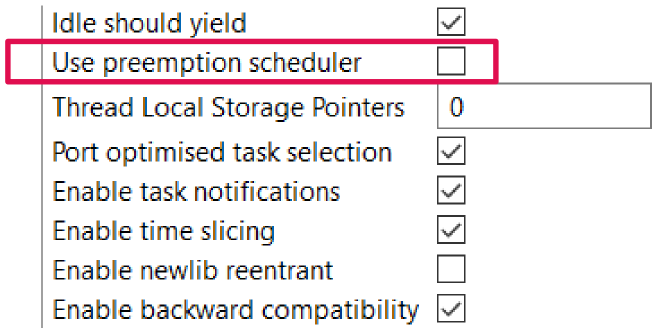
Lab 2: Scheduler Policies – Non-Preemptive

Idle should yield	☑
Use preemption scheduler	☐
Thread Local Storage Pointers	0
Port optimised task selection	☑
Enable task notifications	☑
Enable time slicing	☑
Enable newlib reentrant	☐
Enable backward compatibility	☑

- A running task must yield to allow any other tasks to run
- Only the Red LED because the Red LED task does not yield to let other tasks run unless it is modified

```
static void task_blink_red_led( void *pvParameters ) {
    for (;;) {
        /* wait approximently one second
         * The ticks can be delayed slightly by interrupts and higher priority
         * tasks so this is not a good method to wait a specific amount of time*/
        //vTaskDelay(pdMS_TO_TICKS(1000));

        /* delay that does not use the operating system but rather hogs the processor */
        Delay(5000000);

        /* Toggle the red led */
        PINS_DRV_TogglePins(LED_RED_PORT, 1 << LED_RED_PIN);
        portYIELD();
    }
}
```

Lab 2: Scheduler Policies – Non-Preemptive

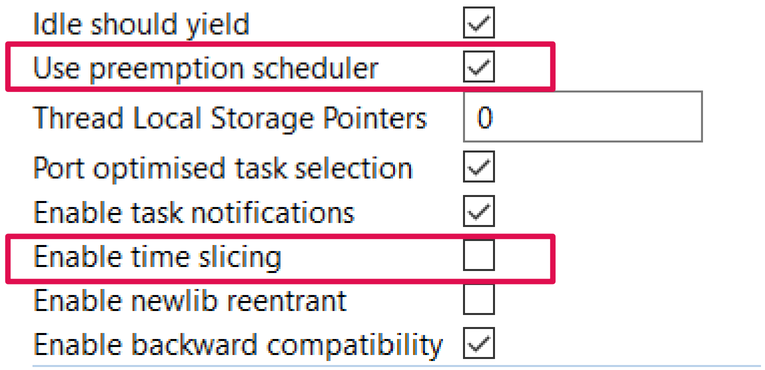


- A running task must yield to allow any other tasks to run
- Only the Red LED because the Red LED task does not yield to let other tasks run unless it is modified

```
static void task_blink_red_led( void *pvParameters ) {
    for (;;) {
        /* wait approximently one second
         * The ticks can be delayed slightly by interrupts and higher priority
         * tasks so this is not a good method to wait a specific amount of time*/
        //vTaskDelay(pdMS_TO_TICKS(1000));

        /* delay that does not use the operating system but rather hogs the processor */
        Delay(5000000);

        /* Toggle the red led */
        PINS_DRV_TogglePins(LED_RED_PORT, 1 << LED_RED_PIN);
        portYIELD();
    }
}
```

To allow another task of the same priority to run, task_blink_red_led must give up control (yield)

Lab 2: Scheduler Policies – Non-Time Slicing



- Preemptive scheduler without time slicing
- Only the Red LED will blink because the it is at the same priority as the Blue LED task and will not yield

```
static void task_blink_red_led( void *pvParameters ) {
    for (;;) {
        /* wait approximently one second
         * The ticks can be delayed slightly by interrupts and higher priority
         * tasks so this is not a good method to wait a specific amount of time*/
        //vTaskDelay(pdMS_TO_TICKS(1000));

        /* delay that does not use the operating system but rather hogs the processor */
        Delay(5000000);

        /* Toggle the red led */
        PINS_DRV_TogglePins(LED_RED_PORT, 1 << LED_RED_PIN);
        portYIELD();
    }
}
```

To allow another task of the same priority to run, task_blink_red_led must give up control (yield)

Lab 2: Scheduler Policies – Non-Time Slicing

Idle should yield	☑
Use preemption scheduler	☑
Thread Local Storage Pointers	0
Port optimised task selection	☑
Enable task notifications	☑
Enable time slicing	☐
Enable newlib reentrant	☐
Enable backward compatibility	☑

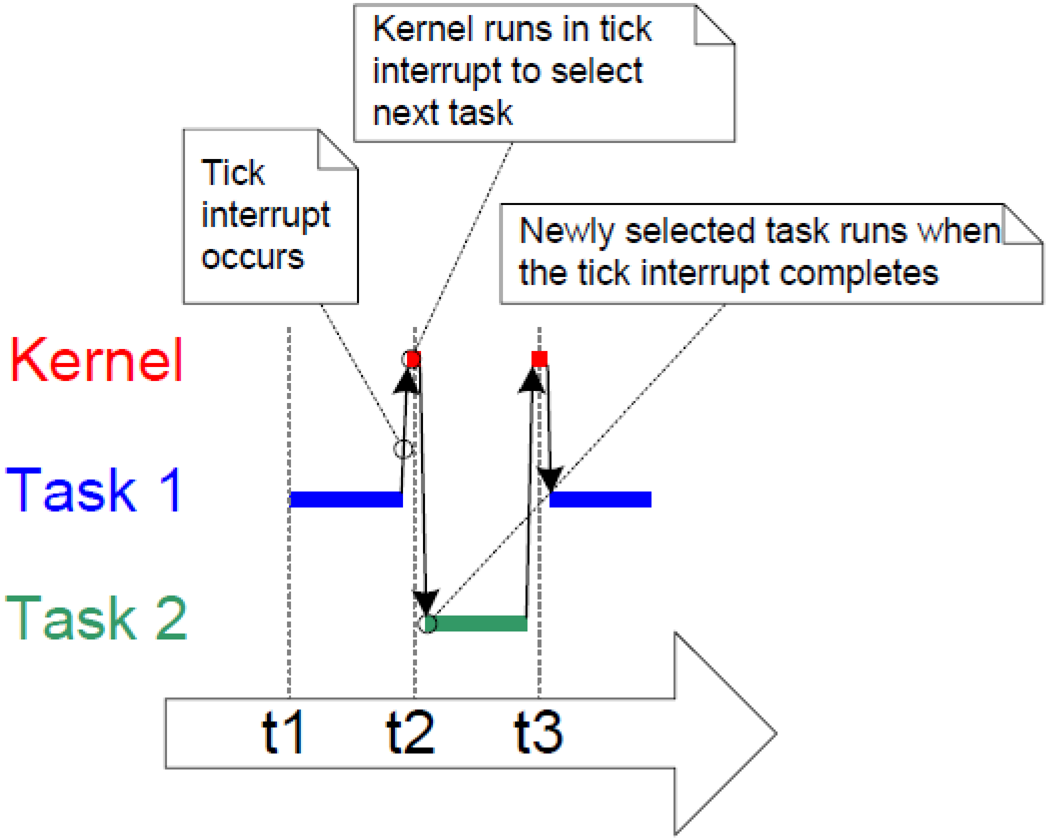
- A task must yield for tasks of the same priority to run
- Otherwise, same-priority tasks will be starved

TCB#	Task Name	Task Handle	Task State	Priority	Stack Usage	Event Ob...	Runtime
1	LCD Init	0x20000208	Suspended	3 (3)	188 B / 268 B		0x1e8 (0.5%)
2	IDLE	0x20000408	Ready	0 (0)	0 B / 300 B		0x78 (0.1%)
3	Tmr Svc	0x20000770	Blocked	3 (3)	140 B / 388 B	Unknown ...	0x1d4f (8.3%)
4	RED LED Tas	0x200009f0	Ready	1 (1)	0 B / 308 B		0x29f (0.7%)
5	BLUE LED Ta	0x20000bf0	Running	1 (1)	0 B / 292 B		0x1036d (73.0%)
6	LCD Stats	0x20001110	Suspended	3 (3)	568 B / 660 B		0x3d6a (17.3%)

Tick Interrupt

- Short periodic interrupt when the kernel is able to run and schedule a new task if necessary.
- Not free: the kernel will run for a short time

- Tick rate is a trade off between amount of overhead and responsiveness of the system



High Kernel Overhead

- Extreme Case: Tick Rate = 170000 Hz

TCB#	Task Name	Task Handle	Task State	Priority	Stack Usage	Event Ob...	Runtime
1	LCD Init	0x20000398	Suspended	3 (3)	216 B / 668 B		0x8ba6 (0.2%)
2	IDLE	0x20000598	Ready	0 (0)	0 B / 300 B		0x297f4 (1.0%)
3	Tmr Svc	0x20000900	Running	3 (3)	52 B / 292 B		0x81874a (49.6%)
4	RED LED Tas	0x20000b80	Ready	1 (1)	12 B / 284 B		0x5 (0.0%)
5	BLUE LED Ta	0x20000d80	Ready	1 (1)	4 B / 284 B		0x6 (0.0%)
6	LCD Stats	0x20001430	Ready	3 (3)	444 B / 828 B		0x805562 (49.2%)

- Tasks get starved because too much time is spent in the kernel for tasks to complete.

FreeRTOS IDLE Task

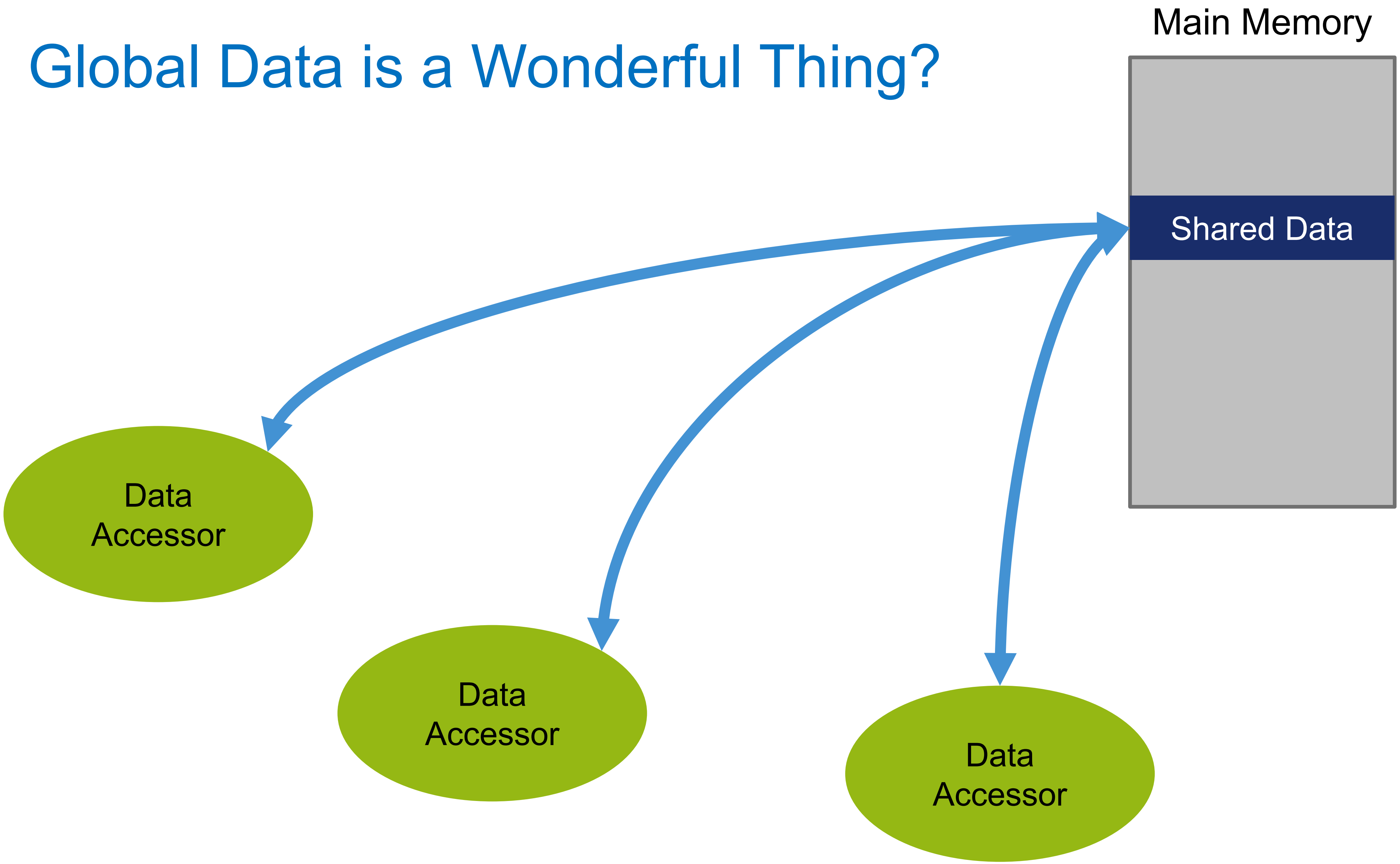
- An Idle task is automatically created by the scheduler
  - Has the lowest priority i.e. 0

- Does clean up for the kernel, meaning the task must not be starved

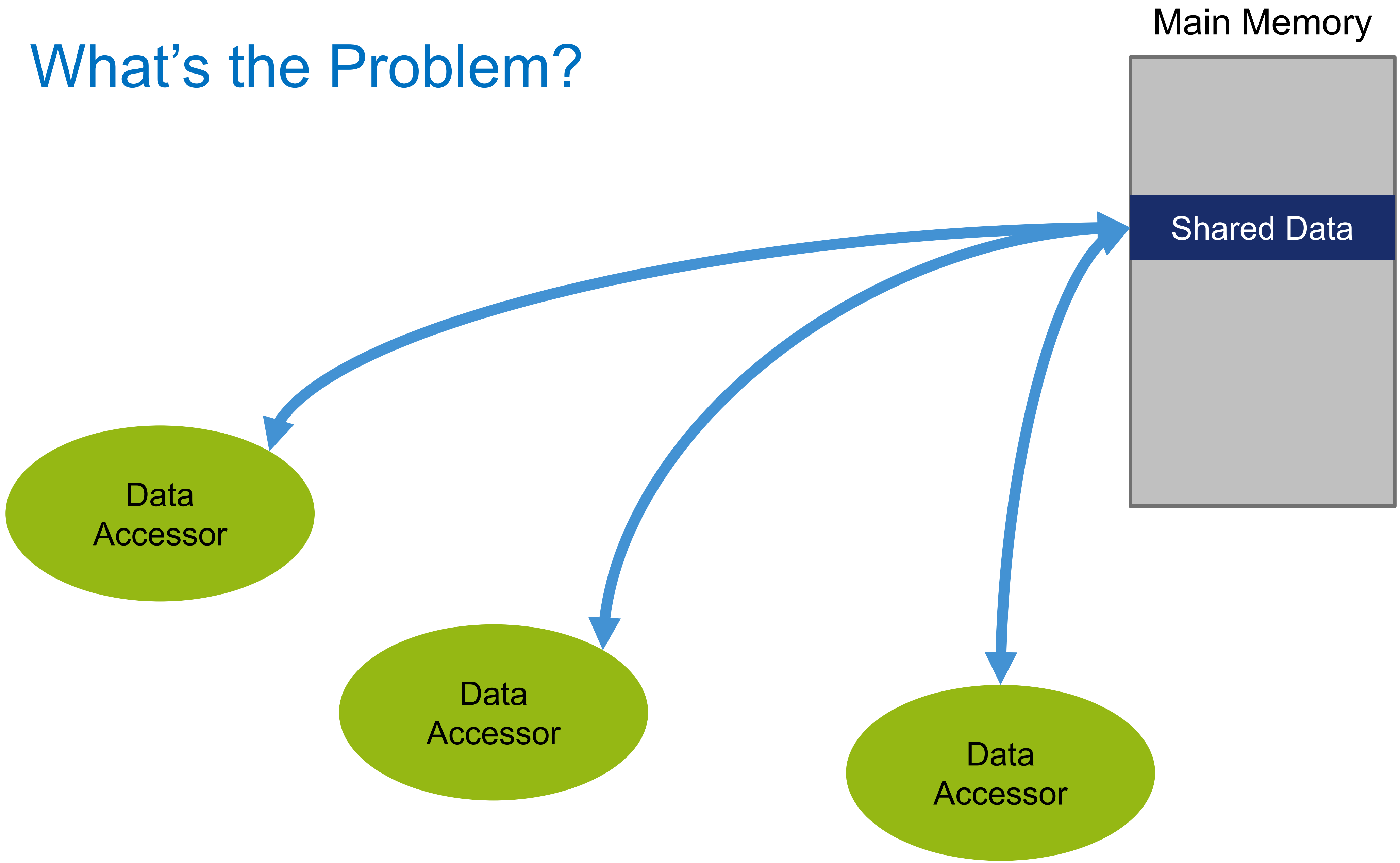
- Idle task Hook
  - It is possible to add application specific functionality directly into the idle task through the use of an idle hook (or idle callback) function.

5. Shared Data

Global Data is a Wonderful Thing?



What's the Problem?



Protection of Shared Data

- Example: Global data updated in interrupt handler. Then accessed by the task.

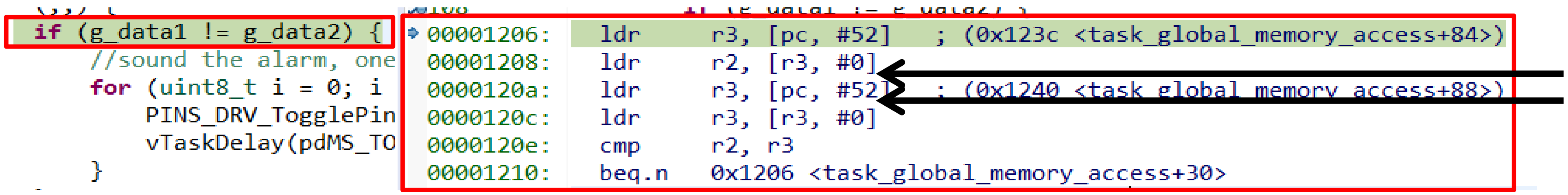
- g_data1 and g_data2 are initialized to the same value

- Can you spot an issue here?

```
static void sw_interrupt_handler(void) {
    PINS_DRV_ClearPortIntFlagCmd(PORTC);

    //manipulate global memory so their sum remains the same
    g_data1++;
    g_data2 = g_data1;

    //very crude de-bounce
    Delay(2000);
    portYIELD_FROM_ISR(pdFALSE);
}

/*-------------------------------------------------------------------

static void task_global_memory_access( void *pvParameters ) {
    for (;;) {
        if (g_data1 != g_data2) {
            //sound the alarm, one of the datas was wrong if we got h
            for (uint8_t i = 0; i < 6; i++) {
                PINS_DRV_TogglePins(LED_RED_PORT, 1 << LED_RED_PIN);
                vTaskDelay(pdMS_TO_TICKS(100));
            }
        }
    }
}
```

Protection of Shared Data

- Context Switches can happen between any two lines of machine code



What if the interrupt occurs at either of these two places?

- Load g_data1 from memory
- Interrupt occurs, values of g_dat1 and g_data2 are updated
- Load g_data2 from memory
- Compare….
- Result is invalid

Protection of Shared Data

- Tricky to debug – This type of error can be intermittent and random
- Can happen:
  - Between ISRs
  - Between ISR and Tasks
  - Between Tasks using preemption
- How can this be prevented?

Lab 3 – Shared Data Problem

- Try it yourself
- Alarm goes off sometimes, but not all the time
- See if you can make a change to prevent this from happening
  - Hint: You should not need to modify code outside of **sw_interrupt_handler()** and **task_global_memory_access()**

Protection of Shared Data

- How can this be prevented?

```
static void task_global_memory_access( void *pvParameters ) {
    for (;;) {
        portDISABLE_INTERRUPTS(); // Masks all OS managed Interrupts (inter
        if (g_data1 != g_data2) {
            portENABLE_INTERRUPTS();
            //sound the alarm, one of the datas was wrong if we got here
            for (uint8_t i = 0; i < 6; i++) {
                PINS_DRV_TogglePins(LED_RED_PORT, 1 << LED_RED_PIN);
                vTaskDelay(pdMS_TO_TICKS(100));
            }
            continue;
        }
        portENABLE_INTERRUPTS();
    }
}
```

Critical section – Only one task can be executing code from this section at once to prevent shared data issues

Protection of Shared Data

FreeRTOS provides some API definitions for these sections

- Macros will disable all interrupts managed by the OS (all interrupts that are a lower priority than the kernel are masked)

portENTER_CRITICAL()

portEXIT_CRITICAL()

Critical sections also lock the scheduler so another task cannot be switched in even if the scheduler were to run

Protection of Shared Data

- Hardware is also essentially global data, so it must also be protected
- Hardware and data can have multiple tasks competing to use them
- Only 1 task must access shared hardware/data at once

- OS has features to help us do this!
  - Locking and unlocking resources using OS API function calls

Protection of Shared Data

- Disabling interrupts is problematic for longer sections
  - Increases maximum potential latency
  - Makes response time more variable

- FreeRTOS provides objects for locking resources with minimal time where interrupts are disabled.

Mutex Locks

- Mutex – Mutual Exclusion

- Designed to restrict access to a resource to one task at a time

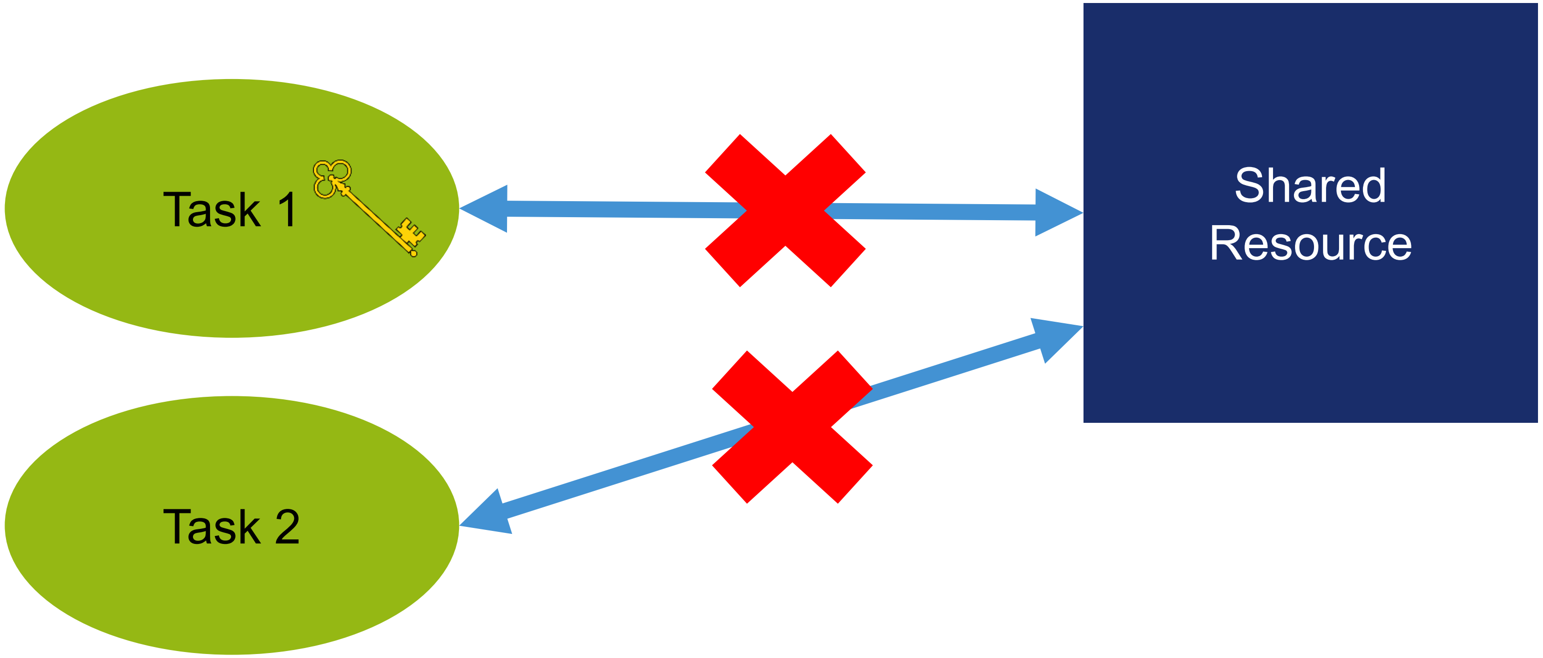
FreeRTOS API:

```
SemaphoreHandle_t xSemaphoreCreateMutex( void )

xSemaphoreTake( SemaphoreHandle_t xSemaphore,
                TickType_t xTicksToWait );
xSemaphoreGive( SemaphoreHandle_t xSemaphore );

xSemaphoreGiveFromISR
        (
          SemaphoreHandle_t xSemaphore,
          signed BaseType_t *pxHigherPriorityTaskWoken
        )
```

*AUTOSAR OS calls this Resource Locks

Mutex Locks

- Analogous to a key that must be held to access data/hardware
- A task must hold the mutex lock to access the hardware



Task 1 Gives up the mutex

Task 2 Takes the mutex

Mutex Lock – Example

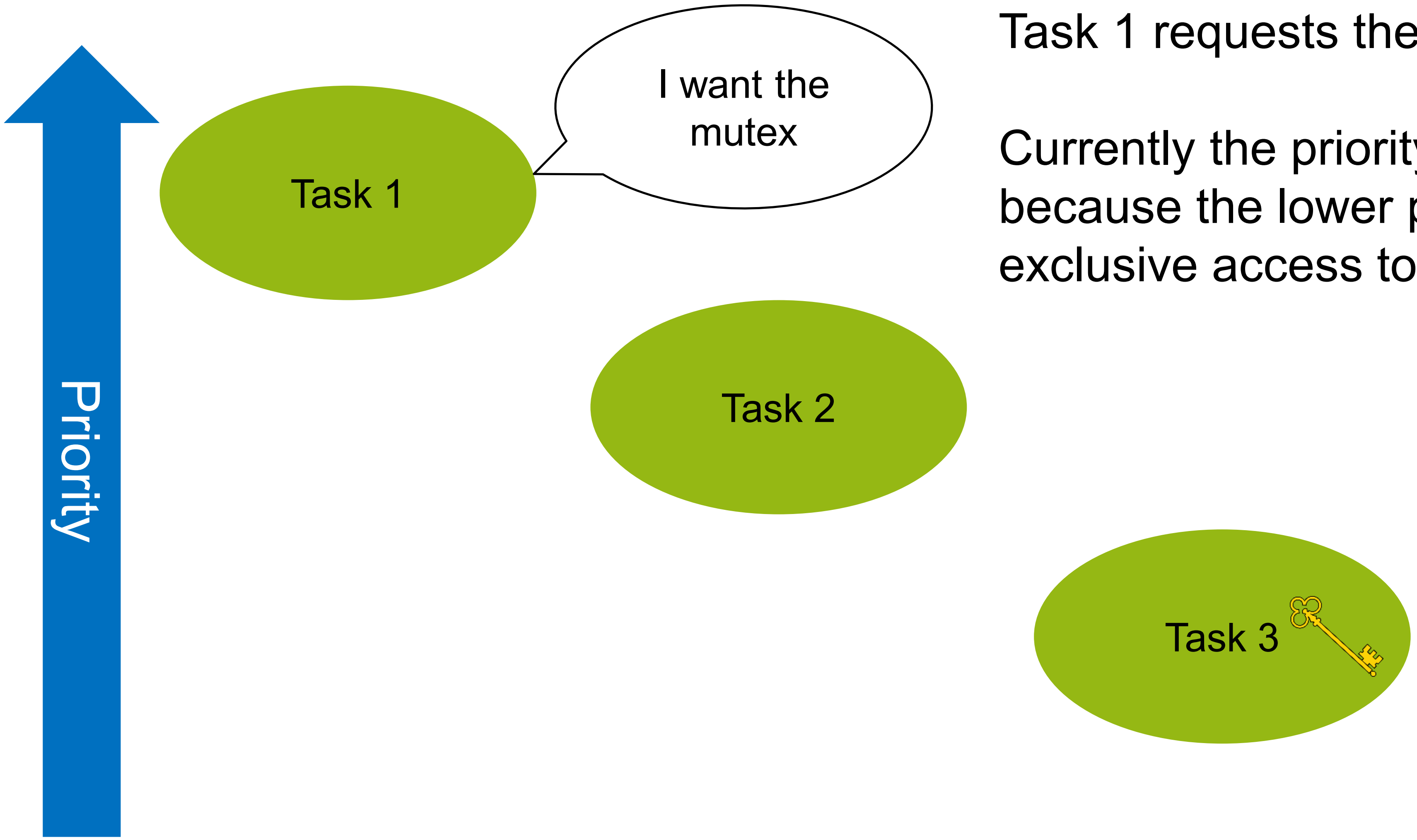
```c
m_spi_lock = xSemaphoreCreateMutex();

void LCD_SendBytes(uint8_t * bytes, uint32_t length) {
    /* Acquire the lock on the spi interface to transmit */
    xSemaphoreTake(m_spi_lock, portMAX_DELAY);

    /* transmit data */
    LPSPI_DRV_MasterTransfer(LPSPICOM1, bytes, NULL, length);
    xEventGroupWaitBits(eg_LCD, //LCD event group
                        eg_LCD_EVENT_XFER_COMPLETE | eg_LCD_EVENT_XFER_ERROR, //look for transfer flags
                        pdTRUE, //clear the flag on return
                        pdFALSE, //do not wait for all flags
                        portMAX_DELAY); //max delay

    /* release lock on the spi */
    xSemaphoreGive(m_spi_lock);
}
```

Acquire the lock for SPI hardware

Resource Access

Release the lock allowing another task to take it

Priority Inversion

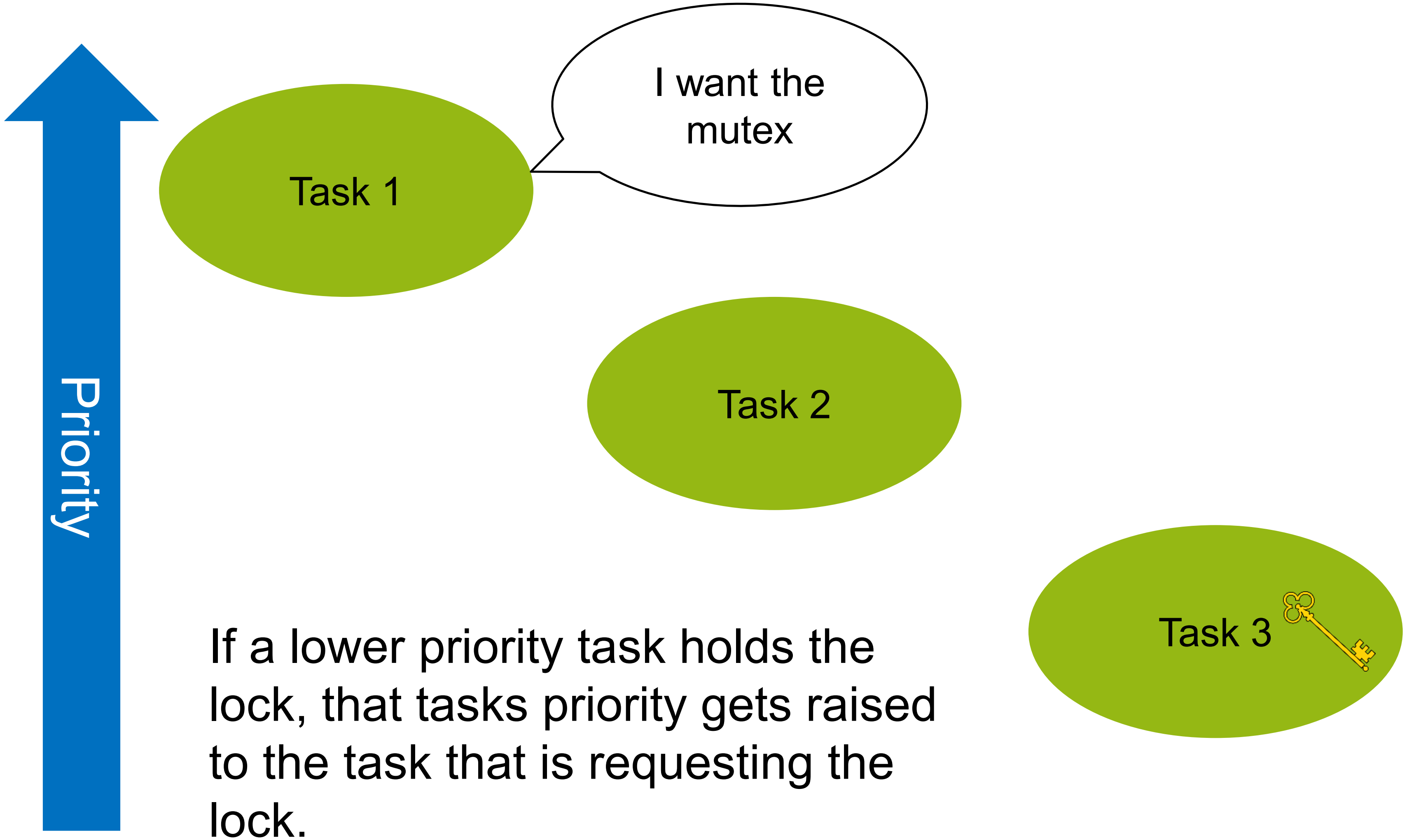


Task 3 holds the lock.

Task 1 requests the lock.

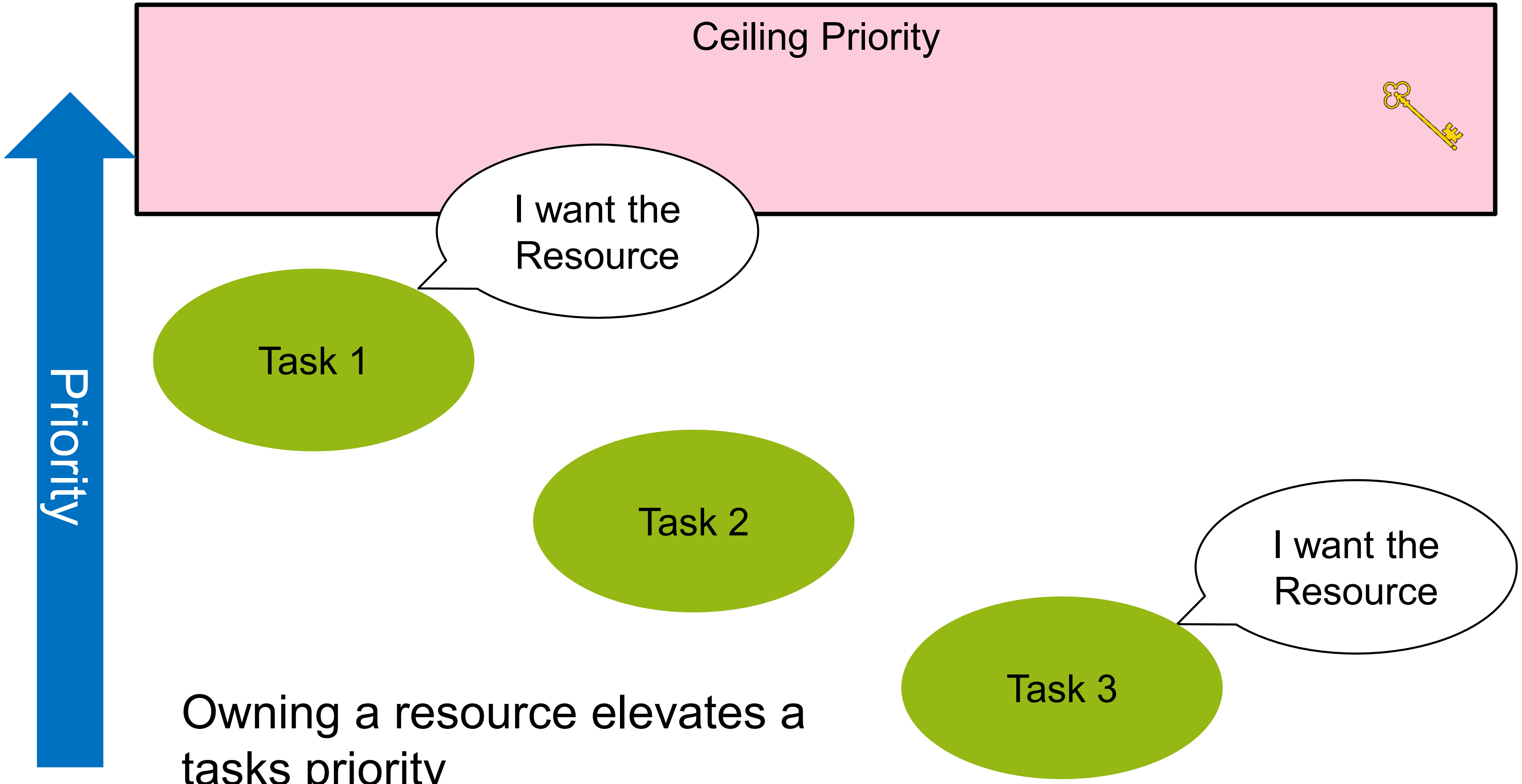
Currently the priority levels are inverted because the lower priority task has exclusive access to that resource.

Priority Inversion with a Mutex Lock – FreeRTOS



If a lower priority task holds the lock, that tasks priority gets raised to the task that is requesting the lock.

Priority Inversion with a Resource Lock – Autosar



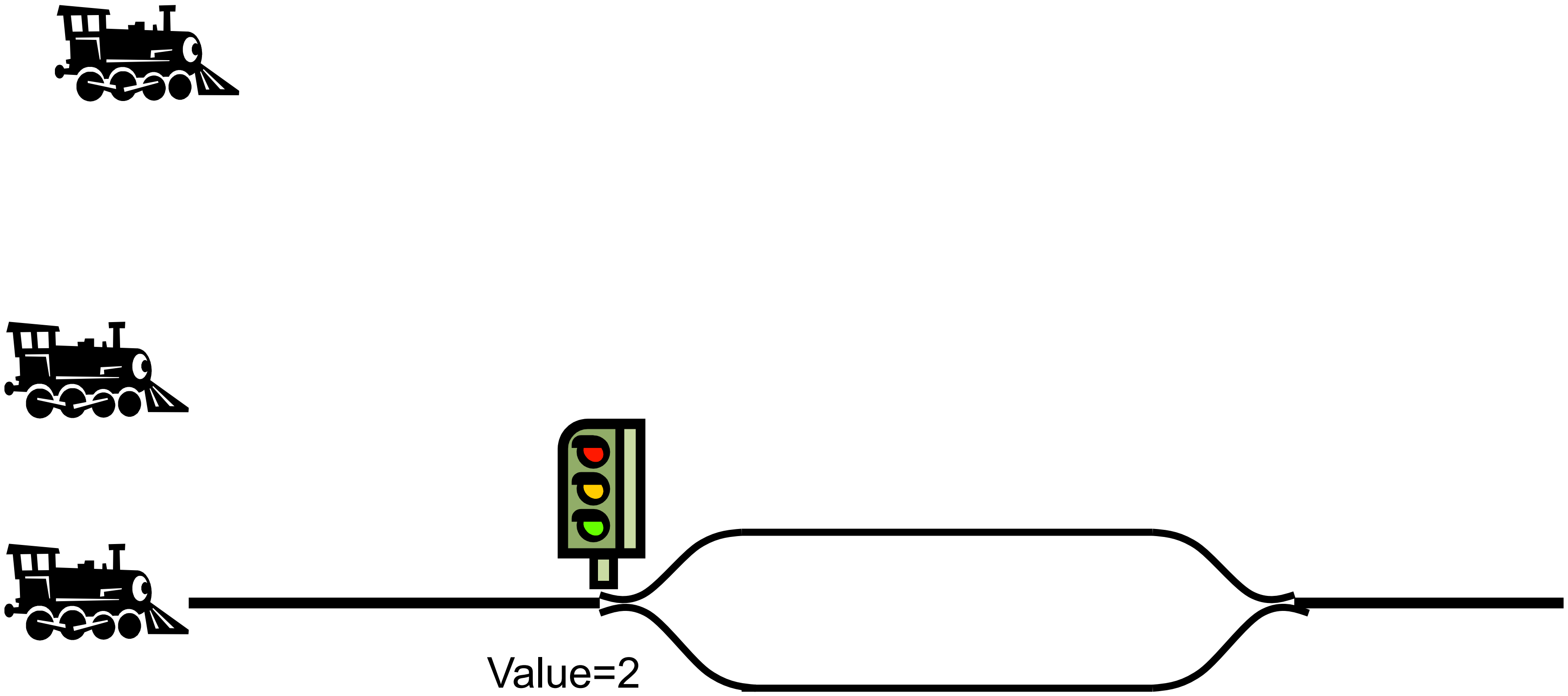
Owning a resource elevates a tasks priority

Semaphores

- Contains a counter that can safely be incremented and decremented.
- Taken and Given similarly to mutex
- Do **NOT** invert priorities like earlier with a mutex lock

Semaphores are similar to parking garages, if there are spots available you can take one but if there is zero you must wait until there is a resource available

Semaphore Railway Analogy



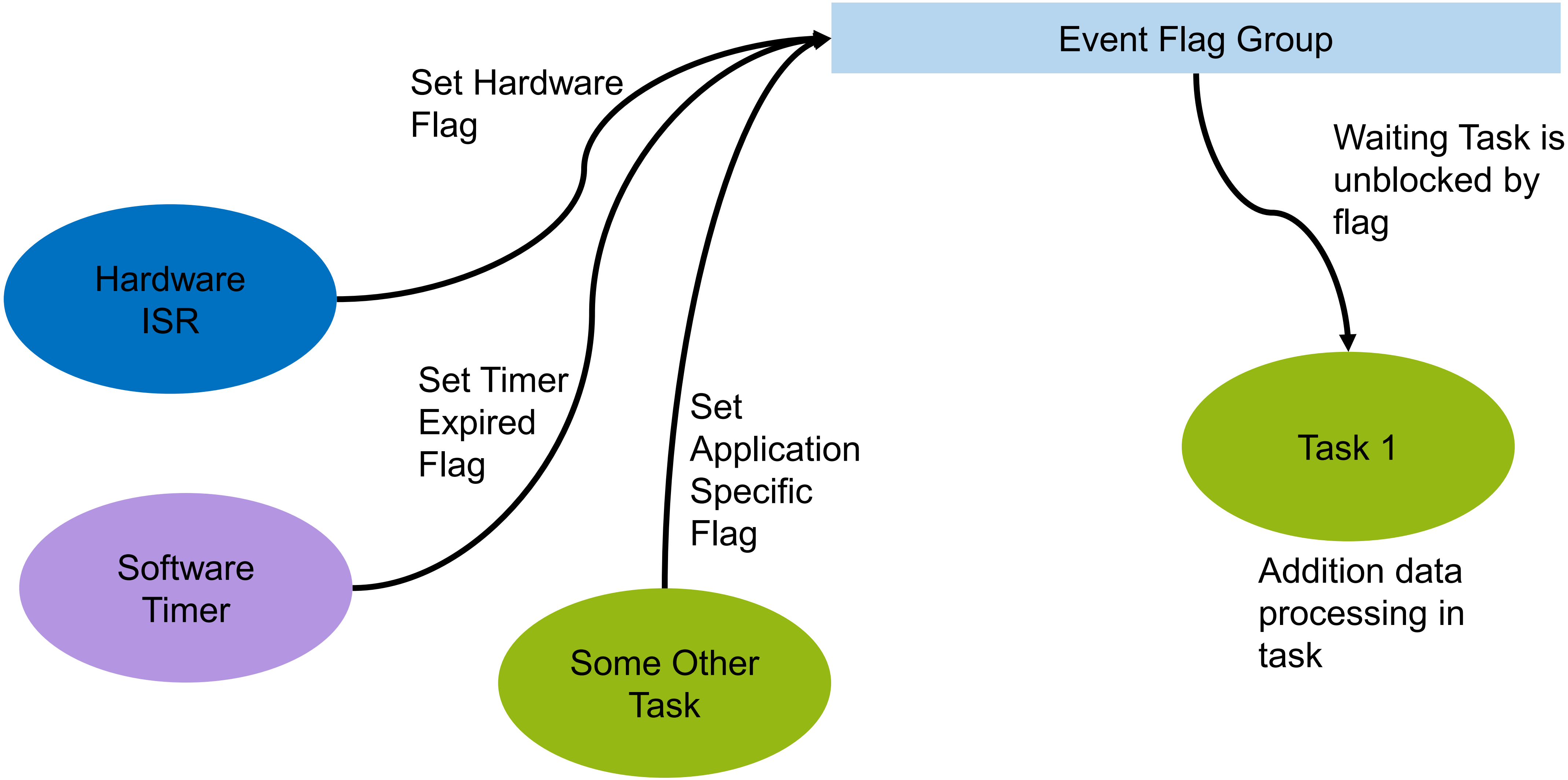
This is a demonstration/example credited Dr. Prof. John Kubiatowicz (Berkeley)

Event Flags

- Flags to signal different events to tasks
- Possible to wait on an event allowing tasks to block
- Task safe way to look for global flags

- FreeRTOS uses a flag groups or bitfields that may contain one or more flags

Example Event Flags



Event Flag APIs with FreeRTOS

```
EventGroupHandle_t xEventGroupCreate( void );

EventBits_t xEventGroupWaitBits(
                        const EventGroupHandle_t xEventGroup,
                        const EventBits_t uxBitsToWaitFor,
                        const BaseType_t xClearOnExit,
                        const BaseType_t xWaitForAllBits,
                        TickType_t xTicksToWait );

EventBits_t xEventGroupSetBits( EventGroupHandle_t xEventGroup,
                                const EventBits_t uxBitsToSet );

EventBits_t xEventGroupClearBits(
                                    EventGroupHandle_t xEventGroup,
                                    const EventBits_t uxBitsToClear );

EventBits_t xEventGroupGetBits( EventGroupHandle_t xEventGroup );
```

All bit manipulation routines have ISR safe versions for use in an ISR

Other Task Communication Methods

- Message Queues – allows messages to be sent to a queue for another task which can be sent and waited on by OS functions

- Stream Buffers – message buffer optimized for a single reader and single writer situation

- Direct Task Notifications – allows for interacting with a specific task without creating an external object

6. Deadlock

A Caution Using Locks

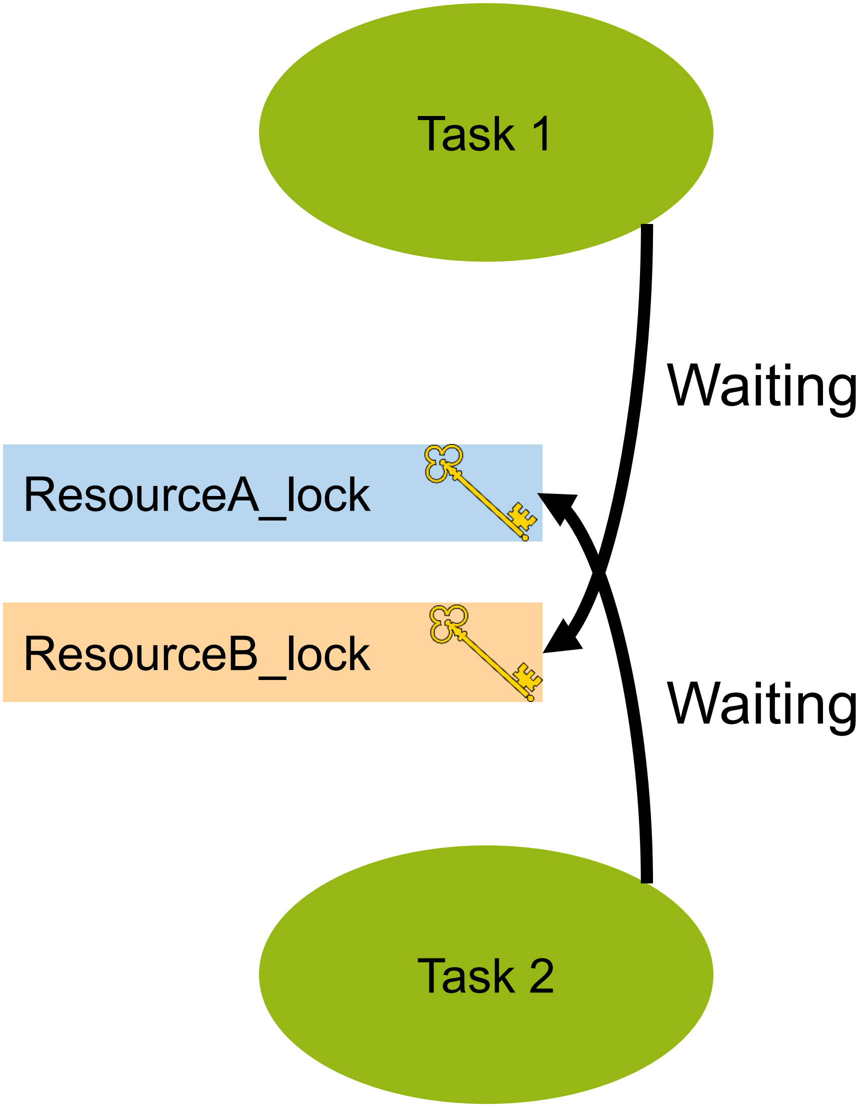
```
SemaphoreHandle_t ResourceA_lock, ResourceB_lock;

static void task1(void *pvParameters) {
    xSemaphoreTake(ResourceA_lock, portMAX_DELAY);
    xSemaphoreTake(ResourceB_lock, portMAX_DELAY);
    //do something
    xSemaphoreGive(ResourceA_lock);
    xSemaphoreGive(ResourceB_lock);
}

static void task2(void *pvParameters) {
    xSemaphoreTake(ResourceB_lock, portMAX_DELAY);
    xSemaphoreTake(ResourceA_lock, portMAX_DELAY);
    //do something
    xSemaphoreGive(ResourceA_lock);
    xSemaphoreGive(ResourceB_lock);
}
```

- See anything that could be problematic?

A Caution Using Locks

Preemptive scheduling means that statements in two separate tasks can execute in any order!

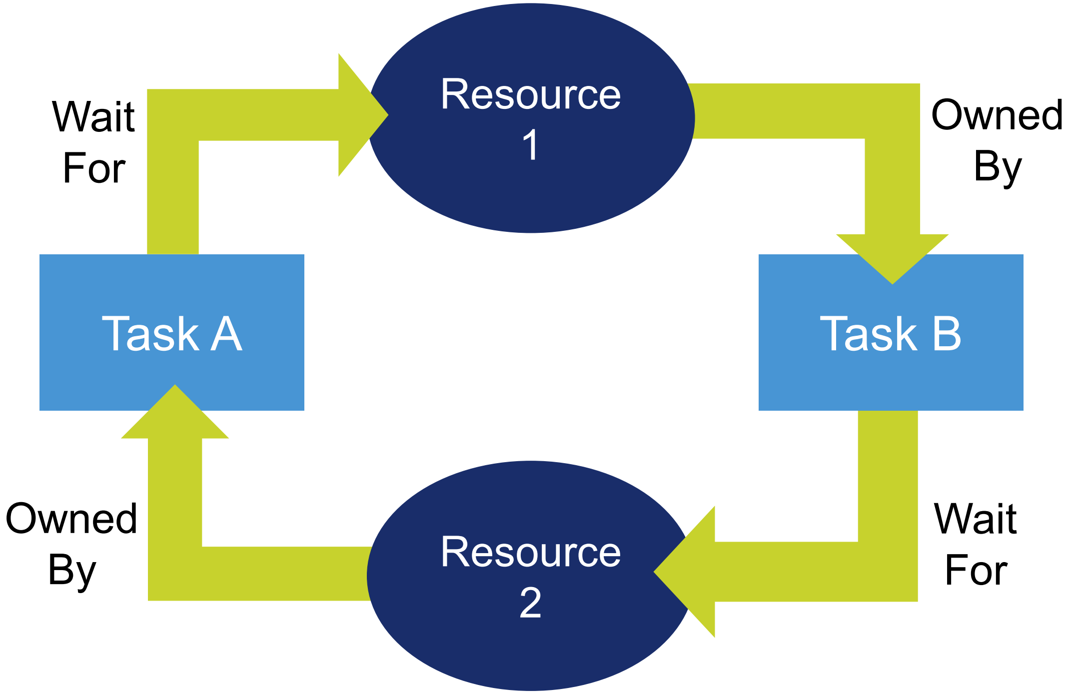
```
SemaphoreHandle_t ResourceA_lock, ResourceB_lock;

static void task1(void *pvParameters) {
    xSemaphoreTake(ResourceA_lock, portMAX_DELAY);
    xSemaphoreTake(ResourceB_lock, portMAX_DELAY);
    //do something
    xSemaphoreGive(ResourceA_lock);
    xSemaphoreGive(ResourceB_lock);
}

static void task2(void *pvParameters) {
    xSemaphoreTake(ResourceB_lock, portMAX_DELAY);
    xSemaphoreTake(ResourceA_lock, portMAX_DELAY);
    //do something
    xSemaphoreGive(ResourceA_lock);
    xSemaphoreGive(ResourceB_lock);
}
```



This is called deadlock!

Deadlock

- A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause



Necessary Conditions for Deadlock

1. Mutual Exclusion Condition
   - Resources are either locked by a process or available
2. Hold and Wait Condition
   - Process can request additional resources while holding a resource
3. No Resource Preemption Condition
   - Resources that are locked cannot be forcibly taken away
4. Circular Wait Condition
   - A circular chain of 2 or more processes are waiting on resources held by another member in the chain

Dealing With Deadlock

- Just ignore it (unsafe) – Done by most operating systems (UNIX and Windows)

- Deadlock Avoidance – Monitor free resources and refuse to allocate a resource if it could potentially cause a deadlock

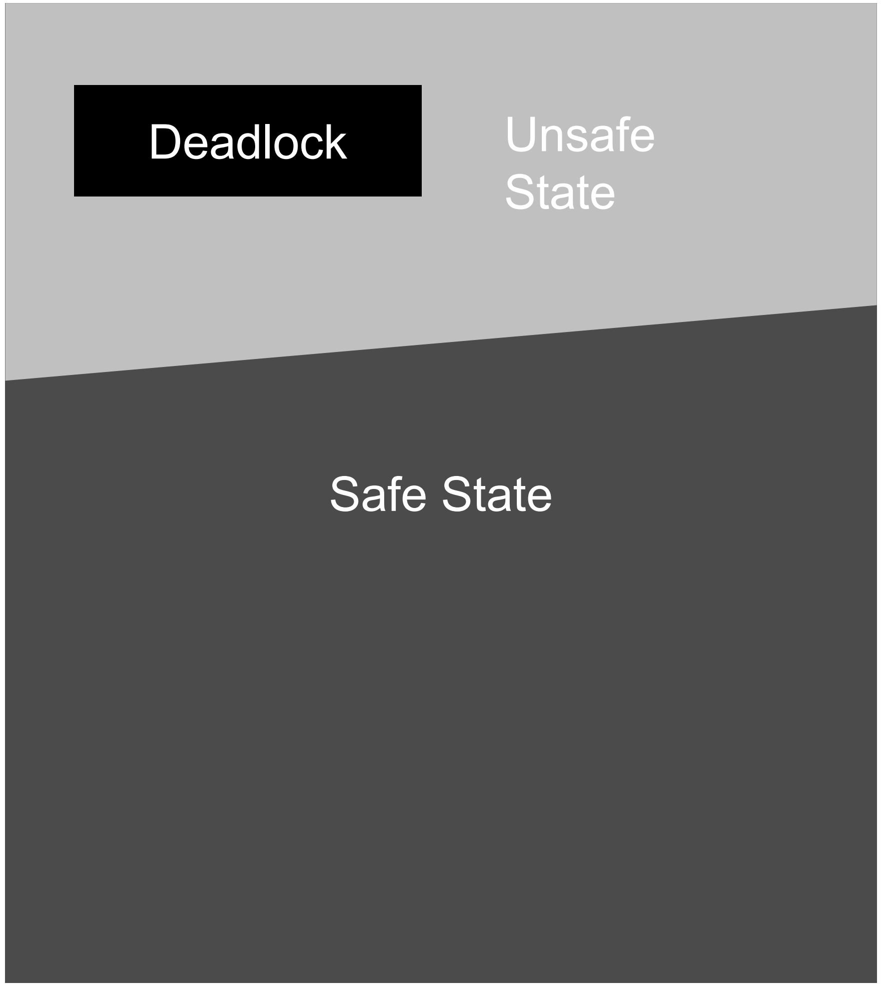
- Deadlock Prevention – Attack one of the 4 necessary conditions for deadlock
  - No Hold and Wait Condition, No Deadlock
  - No Circular waiting, No Deadlock

Deadlock Avoidance

- Keep the system in a safe state
- Monitor maximum resources for each task

- All Deadlocks are unsafe states, but not all unsafe states result in deadlock

- For more reading, see Banker's Algorithm



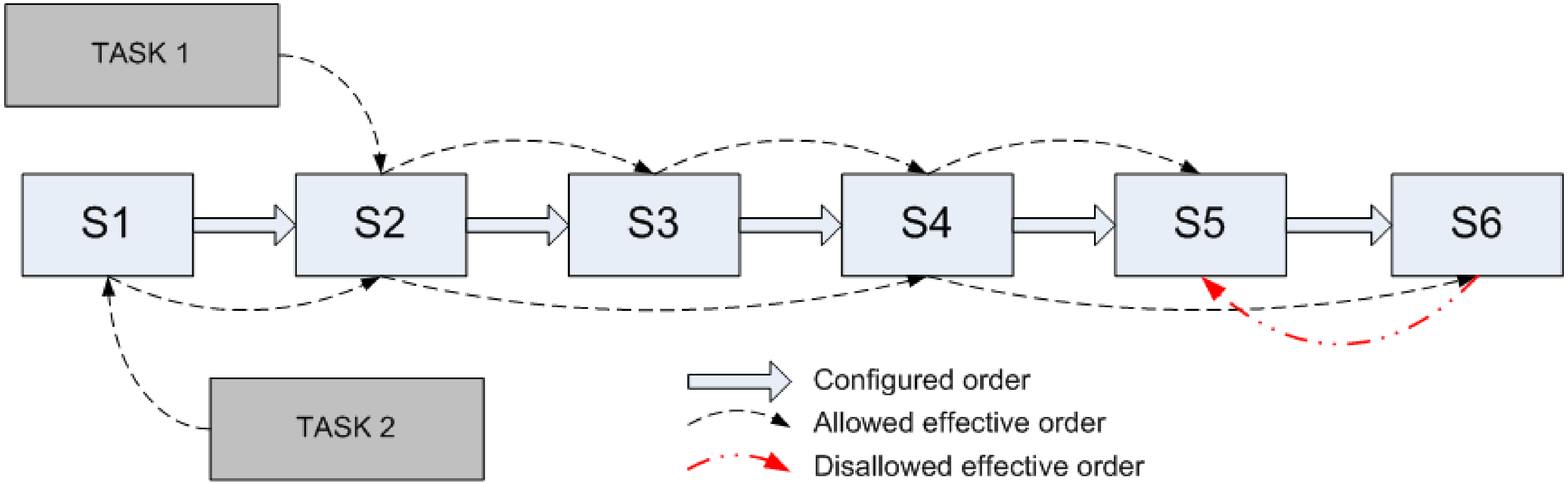
Deadlock Prevention – Attacking No Preemption

- Resources whose state can be easily restored can be preempted
  - CPU registers are saved when a task stops running and restored later

- Not always a viable option
- Consider UART or SPI hardware
  - Halfway through sending a message
  - The hardware is now given to another process to send a different message

Deadlock Prevention – Attacking Circular Waiting

- Locks must have a global order in which they are acquired in to prevent deadlock



Deadlock Prevention – Attacking Hold and Wait

- FreeRTOS APIs allows you to specify a timeout for blocking functions
- Return error if the timeout expires.
- Approach: release any held resources and try again

```
xSemaphoreTake(m_ResourceB_lock, portMAX_DELAY);
```

Timeout in ticks which can be specified based on the application

Lab 4: Task Synchronization and Deadlock

- Event Flags are used to communicate between an ISR and the task that processes those events
- 2 Running tasks are trying to both lock the same two resources resulting in deadlock

7. Application Hooks

Application Hooks

- User defined functions that are called during particular operating system events

- FreeRTOS contains 4 operating system:
  - **Idle Hook:** Called when the system is idle in the idle task
  - **Tick Hook:** Called during the system time tick
  - **Malloc Failed Hook:** Called in the event of a failed allocation
  - **Stack Overflow Hook:** Called in the event of a stack overflow detection

FreeRTOS IDLE Hook

- May be a good place to enter low power mode

```
void vApplicationIdleHook(void) {
  /* Called whenever the RTOS is idle (from the IDLE task ) */
  CPU_EnterLowPowerMode(); /* wait for interrupt */
  /* here an interrupt woke us up */
}
```

- FreeRTOS also has an more advanced feature that can turn off the tick interrupt while the idle task is running allowing long periods of sleep without periodic interrupts

Malloc Failed Hooks

- Example: Stop if allocating memory fails

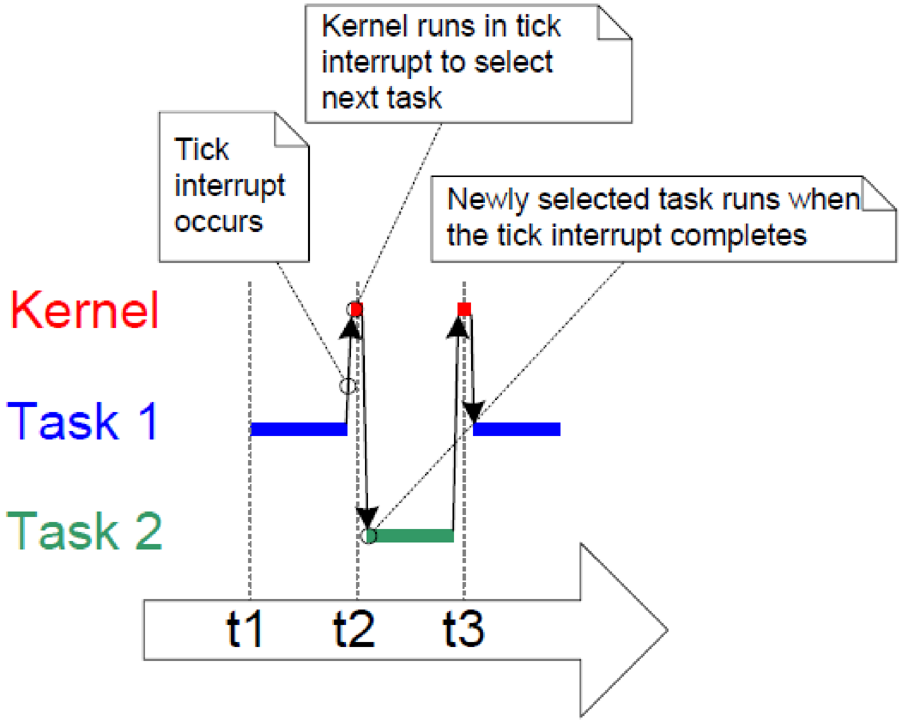
```
void vApplicationMallocFailedHook(void) {
  /* Called if a call to pvPortMalloc() fails because
      there is insufficient free memory available in the
      FreeRTOS heap.  pvPortMalloc() is called internally
       by FreeRTOS API functions that create tasks,
      queues, software timers, and semaphores. The size
      of the FreeRTOS heap is set by the
      configTOTAL_HEAP_SIZE configuration constant in
      FreeRTOSConfig.h. */
  taskDISABLE_INTERRUPTS();
  for(;;) {} /* stop for debugging */
}
```

8. Timing and Software Timers

Timing Based on System Tick

```
/* pdMS_TO_TICKS() takes a time in milliseconds as its only parameter, and evaluates
to the equivalent time in tick periods. This example shows xTimeInTicks being set to
the number of tick periods that are equivalent to 200 milliseconds. */
TickType_t xTimeInTicks = pdMS_TO_TICKS( 200 );
```

- **configTICK_RATE_HZ** to set the system tick rate
- Timing can now be done using software based on system ticks.



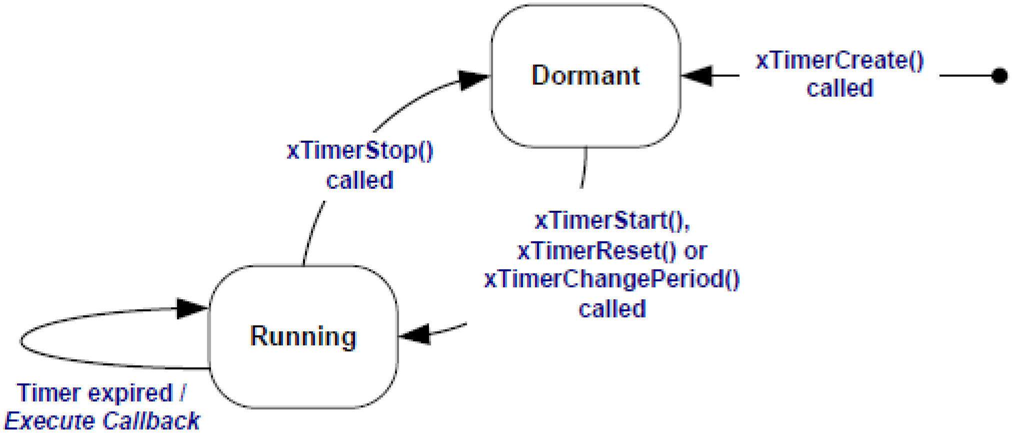
Software Timers

- Used to execute a function sometime in the future based on system ticks
- Can be configured to run once (one-shot) or periodically
- Run outside of an interrupt context
- In FreeRTOS, managed by a timer daemon that is started with the scheduler

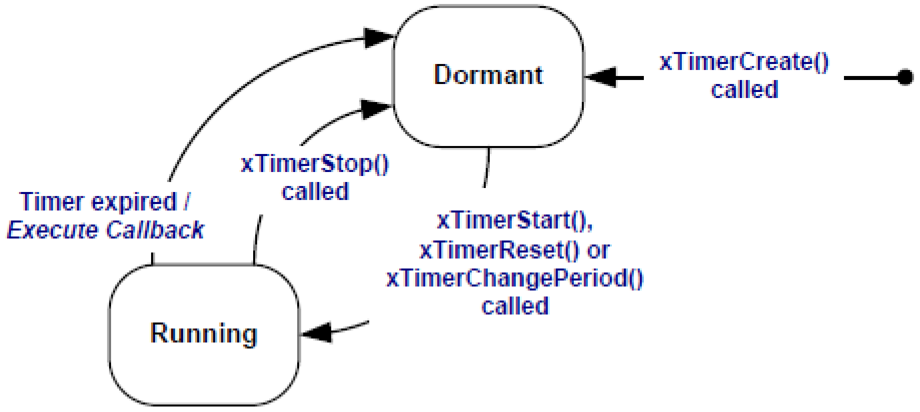
*AUTOSAR has alarms which function similarly

Software Timer States

- A software timer can be in one of the following two states:
  - Dormant: A Software timer that is not running and will not call its callback function. The timer must be started before it is used
  - Running: A Running software timer will execute its associated callback once the timer expires and either reload or transition to the dormant state once the counter has expired.



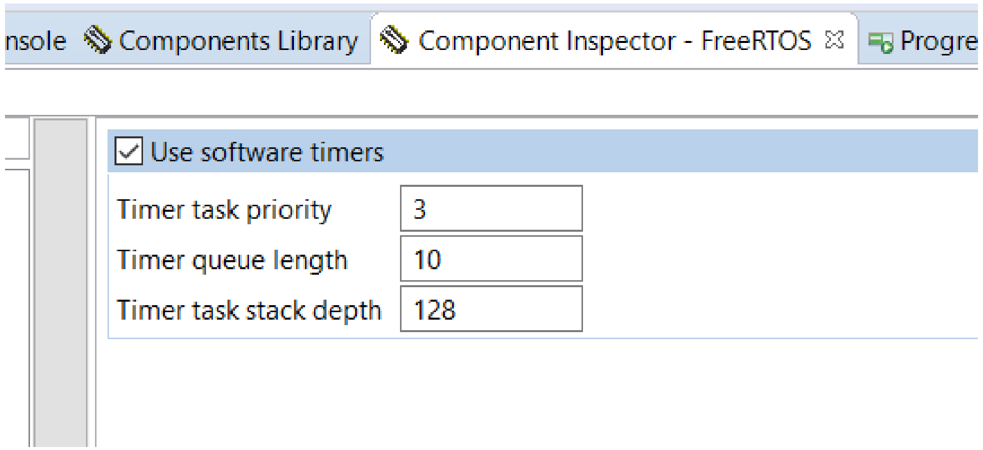
Auto-reload software timer states and transitions



One-short software timer states and transitions

Software Timer Configuration with FreeRTOS

- Can be configured using Processer Expert
- Alternatively done by using FreeRTOSConfig.h




```c
/* Software timer related definitions. */
#define configUSE_TIMERS                1
#define configTIMER_TASK_PRIORITY       ( 3 )
#define configTIMER_QUEUE_LENGTH        10
#define configTIMER_TASK_STACK_DEPTH    128
```

Timer Creation

```
TimerHandle_t xTimerCreate(
  const char *const pcTimerName,
  const TickType_t xTimerPeriodInTicks,
  const UBaseType_t uxAutoReload,
  void *const pvTimerID,
  TimerCallbackFunction_t pxCallbackFunction);
```

```
statsTimerHandle = xTimerCreate("LCD Timer",     /* Human Readable Name of the timer (for debug) */
                                1000/portTICK_PERIOD_MS, /* Timeout length in ticks (1000 ms) */
                                pdTRUE,  /* Enable/Disable auto reload of the timer  (Enabled)*/
                                displayTaskHandle, /* Identifier for the timer ( Using the task it is associated
                                vTimer_callback_display_statistics) /* timer expiration callback function */;
```

Timer Creation

- Once a timer has been created it needs to be started and managed by the timer APIs

```
BaseType_t xTimerStart(TimerHandle_t xTimer,
  TickType_t xTicksToWait);
BaseType_t xTimerStop(TimerHandle_t xTimer,
  TickType_t xTicksToWait);
BaseType_t xTimerReset(TimerHandle_t xTimer,
  TickType_t xTicksToWait);
BaseType_t xTimerDelete(TimerHandle_t xTimer,
  TickType_t xTicksToWait);
```

Lab 5 – Using Timers

- The lab 5 demo contains a simple clock based on software timers.
- 4 timers to track time
  - AM/PM
  - Hours
  - Minutes
  - Seconds
- Software timers are useful tools when scheduling time based events

Lab 5 – Using Timers

- Accuracy is based off of the system clock because ticks happen based on systick.

Summary – FreeRTOS Workshop

- Demonstrated Setup of FreeRTOS and debug tools built into S32K Design Studio
- How to change the scheduling policy and the explored the different behaviors of each policy
- Learned about the dangers of using global memory in an operating system context, and safe ways to work around this
- Learned about strategies to avoid deadlock
- Used software timers to defer processing of a task

SECURE CONNECTIONS
FOR A SMARTER WORLD