1 FreeRTOS on TM4C129ENCPDT

FreeRTOS is a portable open source real-time kernel which operate in preemptive or cooperative mode. It has a small memory footprint (approx. 5-10 kb) and fairly short processing overhead. The features of the kernel are configured at compile time, for instance the tick frequency and which kernel functions to include/exclude. Tasks and semaphores are created at runtime, as well as the priority-based scheduling.

1.1 Create a project including FreeRTOS

1. Create a new project with name *"FreeRTOS_Example"* in a similar way explained in Lab 1 (Section 6 page 14).
2. Add a new folder with name *"FreeRTOS"*.
3. Right click on *"FreeRTOS"* folder and select *Import* → *Import*.
4. Click on *General* → *File System* → *Next.*

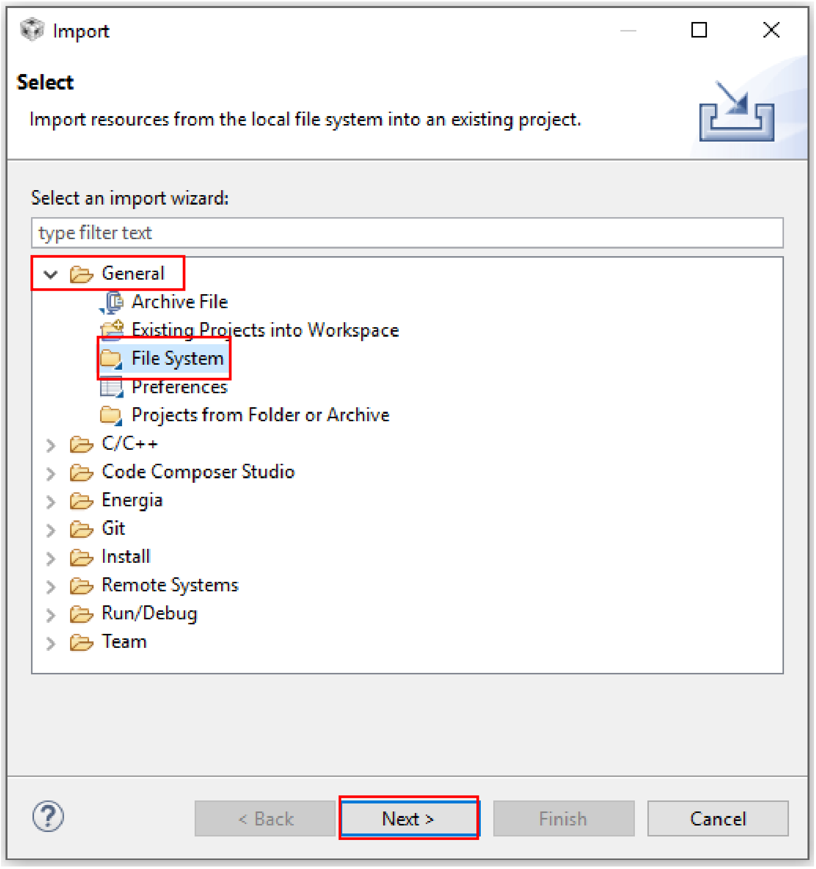
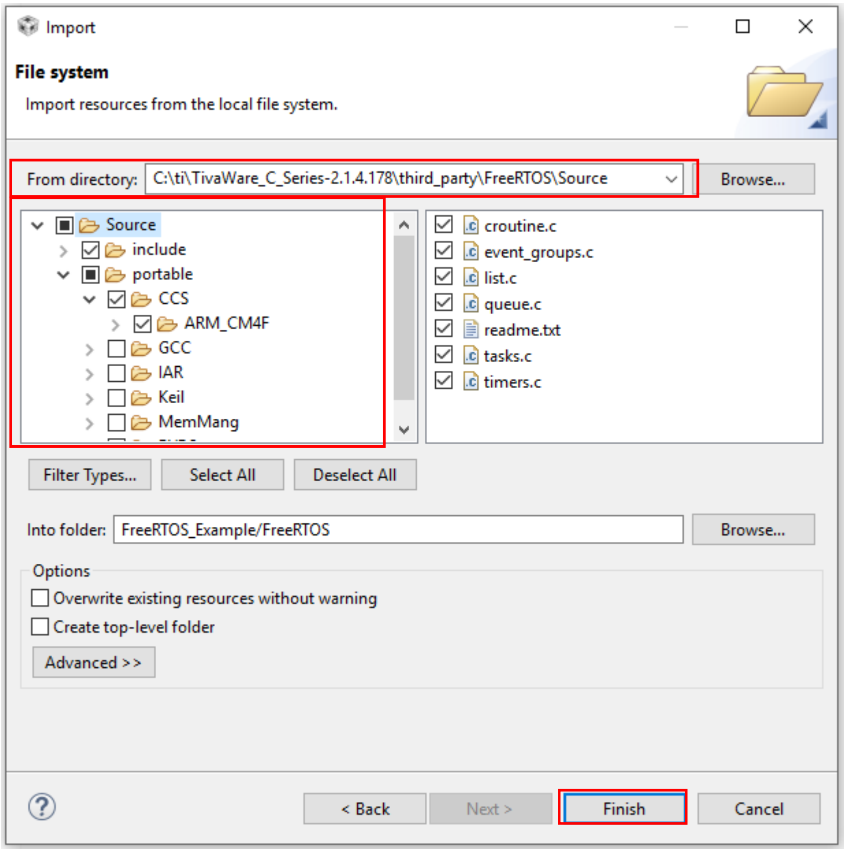


Figure 1. Import FreeRTOS to the Project (1)

5. Browse to the folder *C:\ti\TivaWare_C_Series-2.1.4.178\third_party\FreeRTOS\Source* and click on *OK*.
6. Select the files shown in Figures 2 and 3 and click on *Finish*.



Import

File system

Import resources from the local file system.

From directory: C:\ti\TivaWare_C_Series-2.1.4.178\third_party\FreeRTOS\Source

- ■ Source
  - ☑ include
  - ■ portable
    - ☑ CCS
      - ☑ ARM_CM4F
    - ☐ GCC
    - ☐ IAR
    - ☐ Keil
    - ☐ MemMang

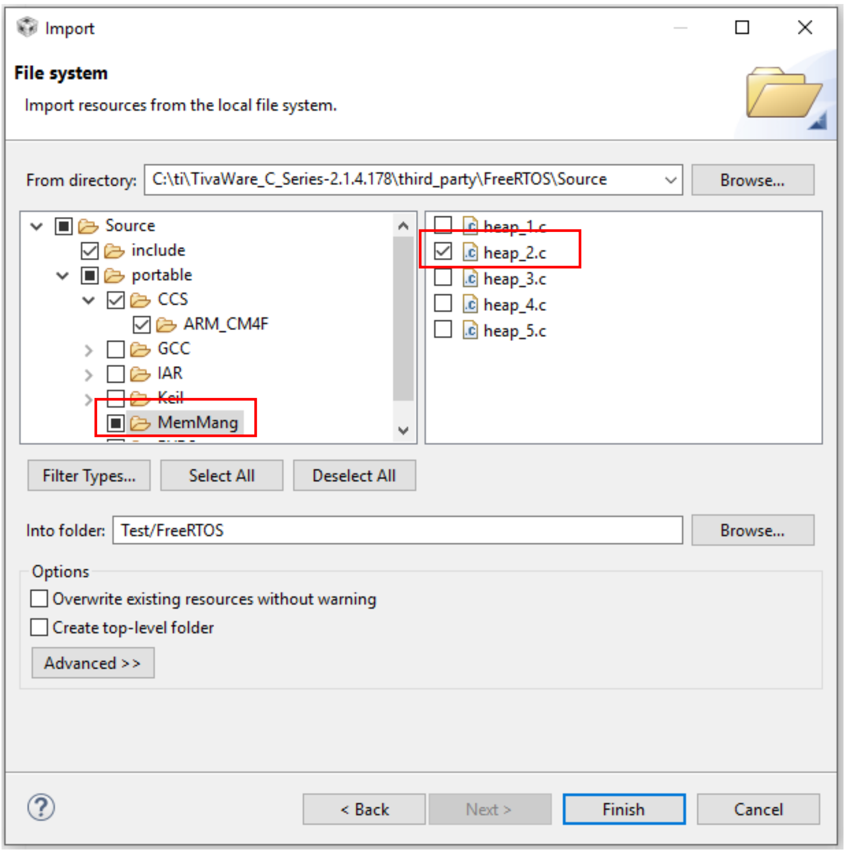
- ☑ croutine.c
- ☑ event_groups.c
- ☑ list.c
- ☑ queue.c
- ☑ readme.txt
- ☑ tasks.c
- ☑ timers.c

Into folder: FreeRTOS_Example/FreeRTOS

Options

- ☐ Overwrite existing resources without warning
- ☐ Create top-level folder

Figure 2. Import FreeRTOS to the Project (2)



Import

File system

Import resources from the local file system.

From directory: C:\ti\TivaWare_C_Series-2.1.4.178\third_party\FreeRTOS\Source

- ■ Source
  - ☑ include
  - ■ portable
    - ☑ CCS
      - ☑ ARM_CM4F
    - ☐ GCC
    - ☐ IAR
    - ☐ Keil
    - ■ MemMang

- ☐ heap_1.c
- ☑ heap_2.c
- ☐ heap_3.c
- ☐ heap_4.c
- ☐ heap_5.c

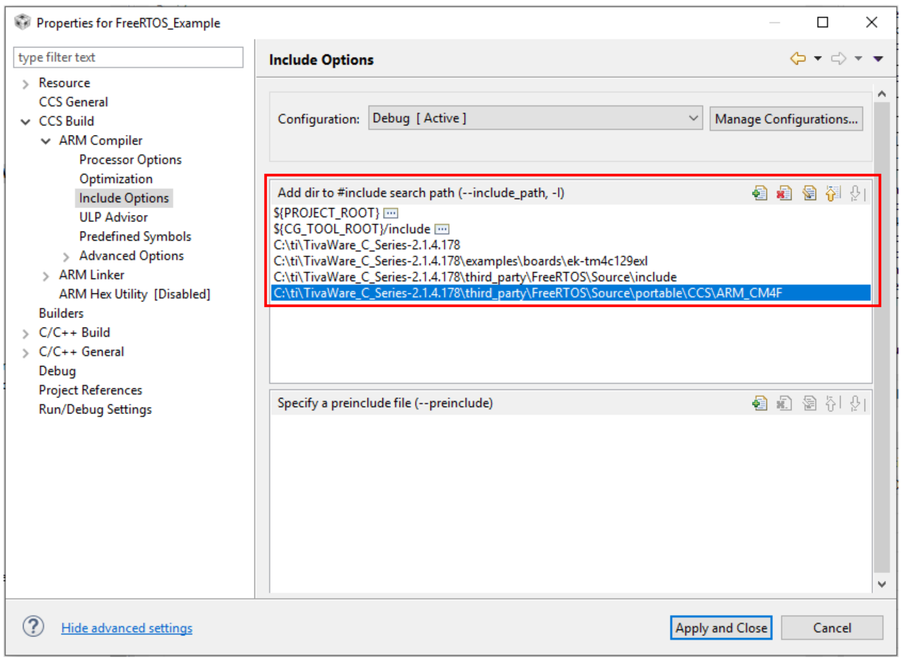
Into folder: Test/FreeRTOS

Options

- ☐ Overwrite existing resources without warning
- ☐ Create top-level folder

Figure 3. Import FreeRTOS to the Project (3)

7. Right click on the project name and select *Properties*.
8. Click on *Include* Options and click on *+ icon*. From the opened window, click on *Browse* and select *C:\ti\TivaWare_C_Series-2.1.4.178\third_party\FreeRTOS\Source\include*.
9. Repeat previous step to add *C:\ti\TivaWare_C_Series-2.1.4.178\third_party\FreeRTOS\Source\portable\CCS\ARM_CM4F*.


Properties for FreeRTOS_Example

Include Options

Configuration: Debug [Active]

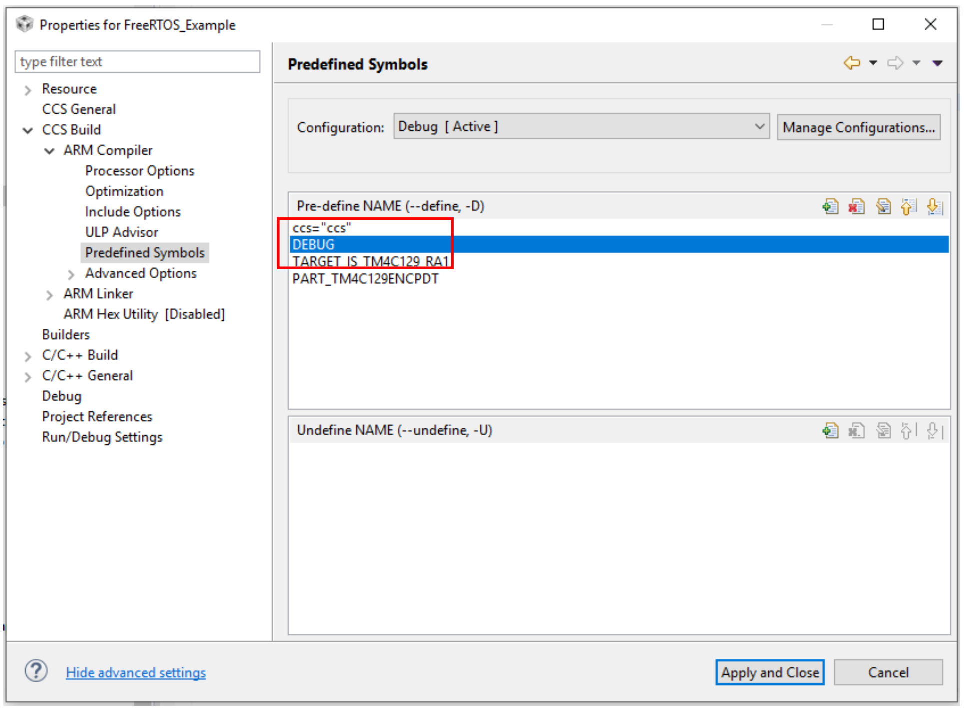
Add dir to #include search path (--include_path, -I)

```
${PROJECT_ROOT}
${CG_TOOL_ROOT}/include
C:\ti\TivaWare_C_Series-2.1.4.178
C:\ti\TivaWare_C_Series-2.1.4.178\examples\boards\ek-tm4c129exl
C:\ti\TivaWare_C_Series-2.1.4.178\third_party\FreeRTOS\Source\include
C:\ti\TivaWare_C_Series-2.1.4.178\third_party\FreeRTOS\Source\portable\CCS\ARM_CM4F
```

Specify a preinclude file (--preinclude)

Figure 4: Configuring the Include Options for FreeRTOS

10. Click on *Predefined Symbols* then click on *+ icon*. From the opened window, write *DEBUG* and click *ok.*


Properties for FreeRTOS_Example

Predefined Symbols

Configuration: Debug [Active]

Pre-define NAME (--define, -D)

```
ccs="ccs"
DEBUG
TARGET_IS_TM4C129_RA1
PART_TM4C129ENCPDT
```

Undefine NAME (--undefine, -U)

Figure 5: Configuring the ROM Functions for FreeRTOS

11. Double click on the file *tm4c129encpdt_startup_ccs.c.*
    a. In the *Include* section add: `#include "FreeRTOS.h"`
    b. In the section *External declarations for the interrupt handlers used by the application*, add:
    ```
    extern void xPortPendSVHandler(void);
    extern void vPortSVCHandler(void);
    extern void xPortSysTickHandler(void);
    ```
    c. In the section *The vector table*, replace the following lines:
    SVCall handler → vPortSVCHandler
    Debug monitor handler → IntDefaultHandler
    The PendSV handler → xPortPendSVHandler
    SysTick handler → xPortSysTickHandler

12. Add to the project the file *FreeRTOSConfig.h* from *Lab5/src* folder.
13. Save the project and build it.

1.2 FreeRTOS API

Detailed information of the FreeRTOS API is found at http://www.freertos.org by clicking the left menu API Reference.

You are free to use all functions available in FreeRTOS. To start with you should look at the functions listed below:

- *xTaskCreate()*
- *vTaskDelay()*
- *vTaskDelayUntil()*
- *xTaskGetTickCount()*
- *vTaskStartScheduler()*
- *taskYIELD()*
- *taskENTER_CRITICAL()*
- *taskEXIT_CRITICAL()*
- *taskENABLE_INTERRUPTS()*
- *taskDISABLE_INTERRUPTS()*

In later labs these _at least_ these functions will be needed:

- *vSemaphoreCreateBinary()*
- *xSemaphoreCreateMutex()*
- *xSemaphoreGive()*
- *xSemaphoreTake()*
- *xQueueCreate()*
- *vQueueDelete()*
- *xQueueReceive()*
- *xQueuePeek()*
- *xQueueSend()*

Good to know

Before reading the details about the functions, here are a collection of things good to know: FreeRTOS uses a prefix to distinguish between different types. You don't have to follow the FreeRTOS convention but knowing what the strange letters are make things easier to understand. Any combination of these letters is valid, e.g. a function/variable named *pvParameters* is most probably of the type *void ** if it follows the FreeRTOS convention.

1. **Variables**
   - Variables of type char are prefixed *c*
   - Variables of type short are prefixed *s*
   - Variables of type long are prefixed *l*
   - Variables of type float are prefixed *f*
   - Variables of type double are prefixed *d*
   - Enumerated variables are prefixed *e*
   - Other types (e.g. structs) are prefixed *x*
   - Pointers have an additional prefixed *p*, for example a pointer to a short will be prefixed *ps*
   - Unsigned variables have an additional prefix *u*, for example an unsigned short have prefixed *us*
   - void functions are prefixed *v*
2. **Functions**
   - Private functions are prefixed with *prv*
   - API functions are prefixed with their return type, as per the convention defined for variables
   - After the type prefix, function names start with the file in which they are defined. For example *vTaskDelay()* is defined in the file *tasks.c*
3. **malloc() and free()**
   - When the real time kernel requires RAM, instead of calling *malloc()* it makes a call to *pvPortMalloc()*.
   - When RAM is being freed, instead of calling *free()* the real time kernel makes a call to *vPortFree()*.
   - The *malloc()* and *free()* calls are not deterministic.
4. **portBASE_TYPE** is the base type of the TM4C129ENCPDT port, which is 32-bit signed long.
5. **portTickType** is the type of real-time ticks and is 32-bit unsigned long.

1.3 FreeRTOSconfig.h

Configuration of FreeRTOS is done through the *FreeRTOSconfig.h* file found in the project directory. Important configuration settings are briefly described below:

- *configUSE_PREEMPTION*: If set to 0 this configuration option chooses the cooperative RTOS kernel. If set to 1 the preemptive kernel is chosen.
- *configCPU_CLOCK_HZ*: This need to be set to the internal MCU clock, 120 MHz.
- *configTICK_RATE_HZ*: This is the frequency at which the RTOS tick will operate. As the tick frequency goes up, the kernel will become less efficient since it must service the tick interrupt service request (ISR) more often.
- *configMAX_PRIORITIES*: The total number of priority levels that can be assigned to a task. Each new priority level consumes some memory. It is good to keep the priority levels at a minimum. If you have five tasks that run at different priorities, set this value no less than 6 to include the idle task. If you try to create a task with higher priority than you have configured the FreeRTOS kernel for, that task will be set to the maximum priority available to the kernel without warning.

- *configMAX_TASK_NAME_LEN*: The maximum number of characters that can be used to name a task. The name is mostly used for debugging and visualization of the system.
- *configIDLE_SHOULD_YIELD*: This determine if a task set at priority 0 (the idle task priority) should yield or not, to protect it from starvation.
- *configUSE_TRACE_FACILITY*: The FreeRTOS core has trace functionality built in. This item is set to 1 if a kernel activity trace is desired. The trace log is created in RAM, so a buffer needs to be allocated.
- *configMINIMAL_STACK_SIZE*: This is the stack size used by the idle task. An analysis of the optimal value should be possible. Sticking to the default value should be good enough for our purposes.
- *configUSE_CO_ROUTINES*: Defaults to 0. We do not use co-routines in these labs.

1.4 Tasks and the Real-Time Scheduler

The scheduler algorithm of FreeRTOS is simply "highest priority goes first". A low priority task does not execute until there are no higher priority tasks ready for execution. If a higher priority task is always ready, the lower priority task never executes, this is called starvation.

When more than one task has the same priority, tasks are executed in a round robin fashion. Preemption is only allowed if the *configUSE_PREEMPTION* is set to 1 otherwise the cooperative scheduler will be used.

Before starting the real-time scheduler at least one task must be created. This is done by the *xTaskCreate()* function. The idle task, which mostly does not do anything, is created automatically at the same time. It has a priority of *tskIDLE_PRIORITY*, which is set to 0, no other task can have lower priority than that.

1.4.1 Task Creation

A task should have the following structure, particularly it should never be allowed to return:

```
void vTaskFunction( void * pvParameters )
{
...
for(;;) {
/* Task application code below. */
...
}
}
```

The task is created by a call to the *xTaskCreate()* function:

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

Parameters:

- *pvTaskCode* - Pointer to the task entry function. Tasks must be implemented to never return (i.e. continuous loop).

- *pcName* - A descriptive name for the task. This is mainly used to facilitate debugging. Max length defined by *configMAX_TASK_NAME_LEN*.
- *usStackDepth - The size of the task stack specified as the number of variables the stack can hold - not the number of bytes. For example, if the stack is 16 bits wide and usStackDepth is defined as 100, 200 bytes will be allocated for stack storage. The stack depth multiplied by the stack width must not exceed the maximum value that can be contained in a variable of type size_t.*
- *pvParameters* - Pointer that will be used as the parameter for the task being created.
- *uxPriority* - The priority at which the task should run.
- *pvCreatedTask* - Used to pass back a handle by which the created task can be referenced.

Returns:

pdPASS if the task was successfully created and added to a ready list, otherwise an error code defined in the file *projdefs.h*

1.4.2 Starting the Scheduler

The requirement for starting the scheduler is at least one task have previously been created.

```
void vTaskStartScheduler(void);
```

Example:

```
// Task to be created .
void vTaskCode ( void * pvParameters )
{
    for (;;) {
        // Task code
        ...
        }
}

// Function that creates a task .
void vOtherFunction ( void )
{
    static unsigned char ucParameterToPass ;
    xTaskHandle xHandle ;
    // Create the task , store the handle .
    xTaskCreate ( vTaskCode ,
                  " NAME ",
                  MINIMAL_STACK_SIZE ,
                  & ucParameterToPass ,
                  tskIDLE_PRIORITY + 1,
                  & xHandle );
    vTaskStartScheduler ();
    ...
}
```

2 Serial I/O API

2.1 Introduction

The Universal Asynchronous Receiver/Transmitter (UART) API provides a set of functions for using the Tiva UART modules. Functions are provided to configure and control the UART modules, to send and receive data, and to manage interrupts for the UART modules.

The Tiva UART performs the functions of parallel-to-serial and serial-to-parallel conversions. It is very similar in functionality to a 16C550 UART but is not register-compatible. Some of the features of the Tiva UART are:

- A 16x12 bit receive FIFO and a 16x8 bit transmit FIFO.
- Programmable baud rate generator.
- Automatic generation and stripping of start, stop, and parity bits.
- Line break generation and detection.
- Programmable serial interface:
    - 5, 6, 7, or 8 data bits.
    - even, odd, stick, or no parity bit generation and detection.
    - 1 or 2 stop bit generation.
    - baud rate generation, from DC to processor clock/16.
- Modem control/flow control.
- IrDA serial-IR (SIR) encoder/decoder.
- uDMA interface.
- 9-bit operation.

This driver is contained in *driverlib/uart.c*, with *driverlib/uart.h* containing the API declarations for use by applications.

2.2 The UART API

The UART API provides the set of functions required to implement an interrupt-driven UART driver. These functions may be used to control any of the available UART ports on a Tiva microcontroller and can be used with one port without causing conflicts with the other port.

The UART API is broken into three groups of functions: those that deal with configuration and control of the UART modules, those used to send and receive data, and those that deal with interrupt handling (a full description of these functions can be found in Chapter 30 of the document *"TivaWare Peripheral Driver Library"*, file no. 5).

- The clock source for the baud rate generator is handled by the *UARTClockSourceSet()* and *UARTClockSourceGet()* functions.
- Configuration and control of the UART are handled by the *UARTConfigGetExpClk()*, *UARTConfigSetExpClk()*, *UARTDisable()*, *UARTEnable()*, *UARTParityModeGet()*, and *UARTParityModeSet()* functions. The DMA interface can be enabled or disabled by the *UARTDMAEnable()* and *UARTDMADisable()* functions.
- Sending and receiving data via the UART is handled by the *UARTCharGet()*, *UARTCharGet-NonBlocking()*, *UARTCharPut()*, *UARTCharPutNonBlocking()*, *UARTBreakCtl()*, *UARTCharsAvail()*, and *UARTSpaceAvail()* functions.
- Managing the UART interrupts is handled by the *UARTIntClear()*, *UARTIntDisable()*, *UARTIntEnable()*, *UARTIntRegister()*, *UARTIntStatus()*, and *UARTIntUnregister()* functions.

- The 9-bit operation mode is handled by the *UART9BitEnable(), UART9BitDisable(), UART9BitAddrSet(),* and *UART9BitAddrSend()* functions.
- The *UARTConfigSet(), UARTConfigGet(), UARTCharNonBlockingGet(),* and *UARTCharNonBlockingPut()* APIs from previous versions of the peripheral driver library have been replaced by the *UARTConfigSetExpClk(), UARTConfigGetExpClk(), UARTCharGetNonBlocking(),* and *UARTCharPutNonBlocking()* APIs, respectively. Macros have been provided in *uart.h* to map the old APIs to the new APIs, allowing existing applications to link and run with the new APIs.

2.3 A Programming Example

The following example shows how to use the UART API to initialize the UART, transmit characters, and receive characters on a TM4C129x device.

```
//
// Enable the UART0 module.
//
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
//
// Wait for the UART0 module to be ready.
//
    while(!SysCtlPeripheralReady(SYSCTL_PERIPH_UART0))
    {
    }
// Initialize the UART. Set the baud rate, number of data bits, turn off
// parity, number of stop bits, and stick mode. The UART is enabled by the
// function call.
//
    UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 38400,
                        (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
                                UART_CONFIG_PAR_NONE));
//
// Check for characters. Spin here until a character is placed
// into the receive FIFO.
//
    while(!UARTCharsAvail(UART0_BASE))
    {
    }
//
// Get the character(s) in the receive FIFO.
//
    while(UARTCharGetNonBlocking(UART0_BASE))
    {
    }
//
// Put a character in the output buffer.
//
    UARTCharPut(UART0_BASE, ’c’));
//
// Disable the UART.
//
    UARTDisable(UART0_BASE);
```

3 Assignment (2 bonus points if completed together with the report before the deadline, which is specified in Canvas for each assignment)

1. Write a program using FreeRTOS that blinks the LEDs at different intervals, LED1 = 1 second, LED2 = 2 second, LED3 = 3 second, LED4 = 4 second. Pressing a button (1-2) should make the corresponding LED (1-2) stay lit for 10 seconds while the others keep blinking. If both buttons are pressed, one after another, both LEDs should light up. When 10 seconds has passed, the affected LED should go back into its original state, i.e., pressing the button should **NOT** make the LEDs blink out of sync. Make the various task output their actions to the serial port.

2. Firstly, you will implement three different tasks that have periods, release times and priorities in such a manner that the priority inversion (see RTOS lecture slides) takes place. If done properly, the execution of a correctly implemented priority inversion will result in a deadline miss of the tasks. To visualize the execution of the different tasks, you are required to output certain keypoints to the UART listed below:
   When a task is released for execution – "Task x started"
   When a task finishes its execution – "Task x finished"
   When a task successfully takes a semaphore – "Task x sem take"
   When a task releases a semaphore – "Task x sem give"
   When a task starts its workload – "Task x started its workload"

3. Write a software function to detect the deadline miss when the task set in (part 2 the assignment) is executed. You should write the following information via the UART: which task misses its deadline and by how much time the deadline is missed. All tasks should have this deadlines miss detection.

Note 1: A good idea is to discuss the design of the priority inversion with the Lab assistant and get approval before starting to code. It is not mandatory to discuss, but encouraged, as it is quite easy to misinterpret priority inversion. The design must also be attached with the report.
Note 2: VtaskDelay and VtaskDelayUntil are not an acceptable workloads for priority inversion since they enforce a context switch between tasks.

Important Tip: Read the study material provided in the corresponding RTOS lecture before starting the design, there is an simple example of priority inversion that you can follow as well.

4 Report

The report should include the following.

1. What is the main difference between preemptive and cooperative scheduling?
2. What is the difference between *vTaskDelay()* and *vTaskDelayUntil()*?
3. Design of Assignment 2 (priority inversion).

5 Optional Assignments (2 bonus point if completed before the deadline, which is specified in Canvas for each assignment)

Measure the context switching time using a timer. How does it vary with the number of active tasks? Plot context switch time as a function of number of active tasks in a graph.

Note: The bonus points collected after successfully completing the compulsory assignments and optional assignments before the corresponding deadlines will be added to your score in the final exam. These bonus points could help you in improving your final grade in the course.