AIN SHAMS UNIVERSITY
FACULTY OF ENGINEERING
Credit Hours Programs

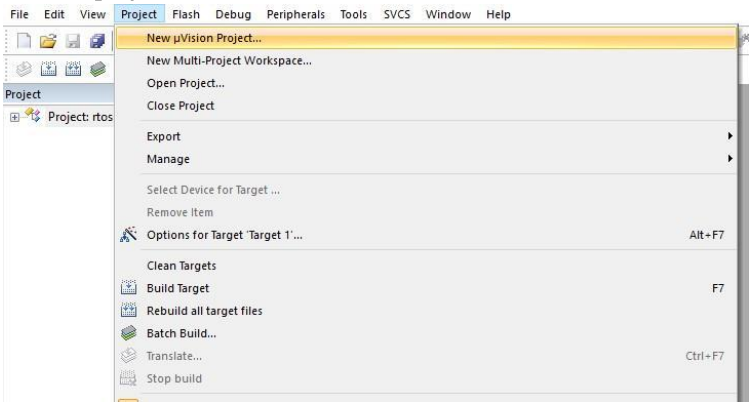
SummerSemester 2024

CSE 411: RT Embedded Systems

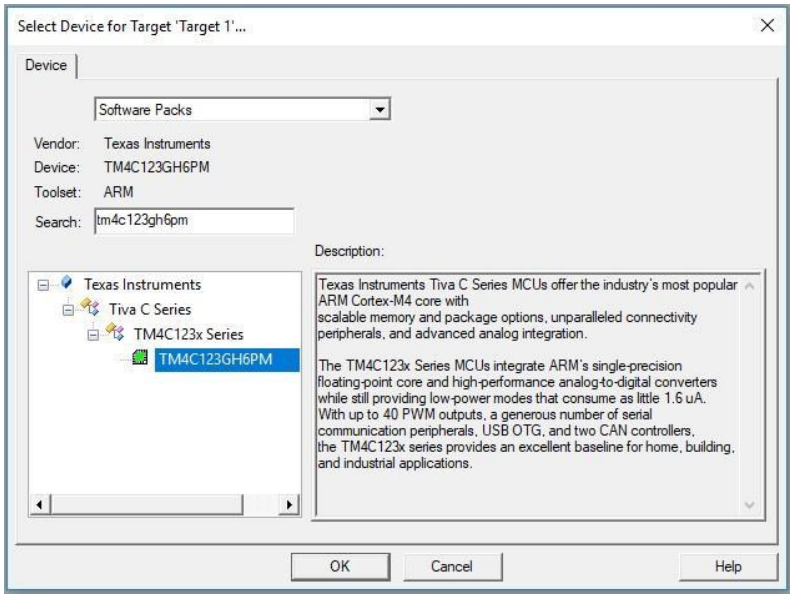
Lab 0: Keil (+Tivaware)

Task 1: Initialize the Micro-controller:

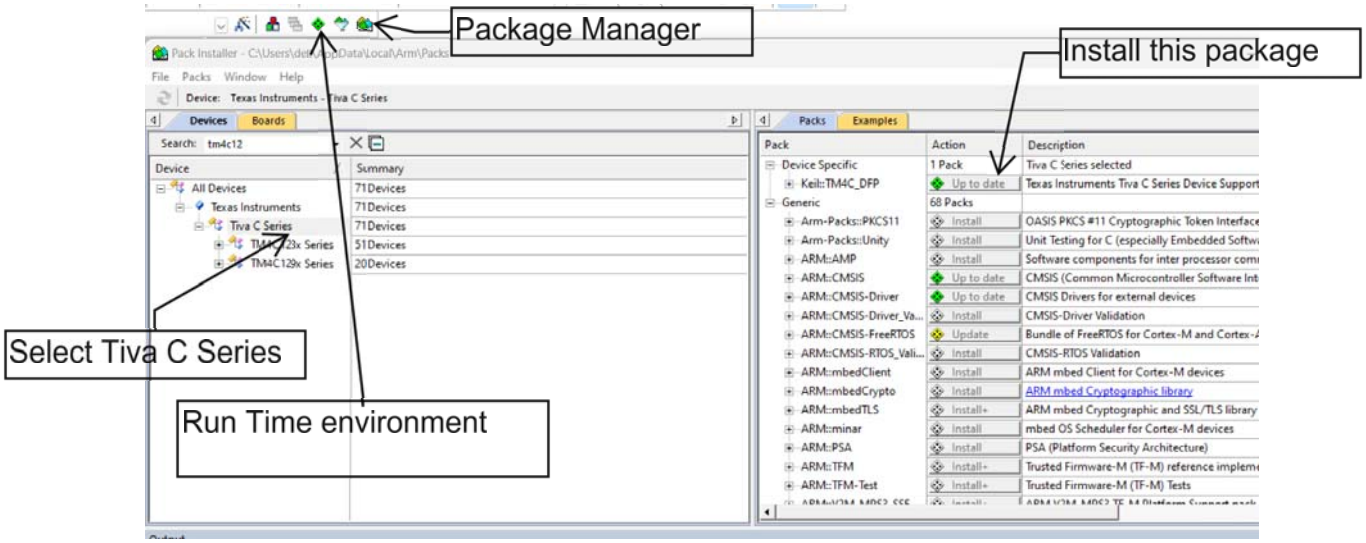
1- Create a new project on Keil

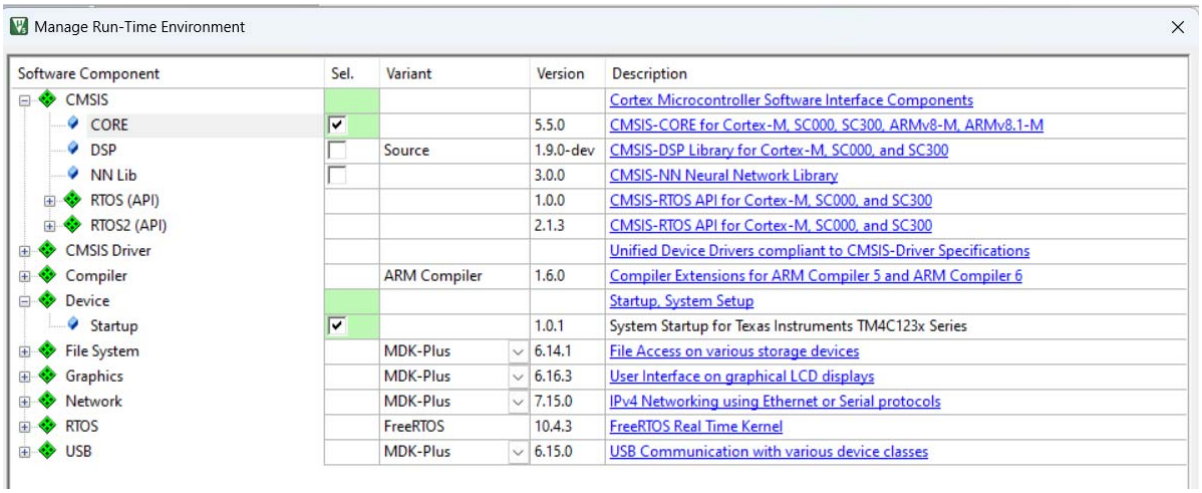


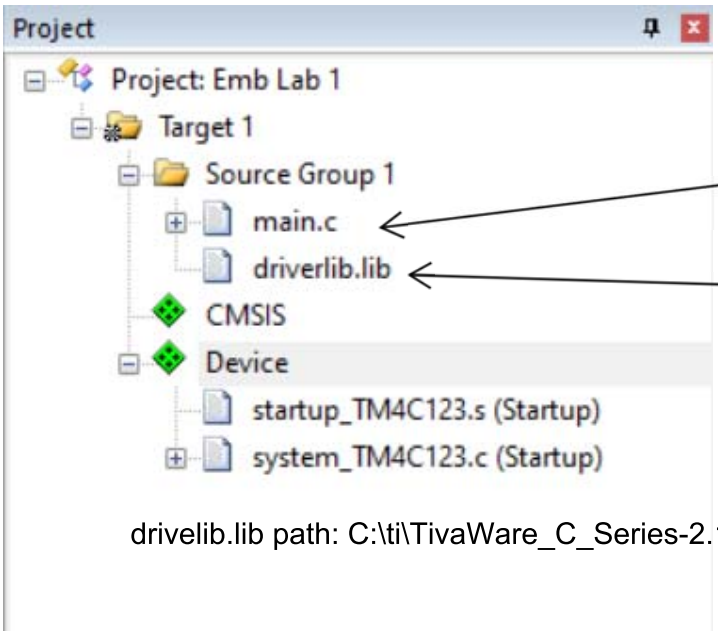
2- Save your project in the desired place, and choose the target (Texas not arm)



if Texas not visible then download it from package manager







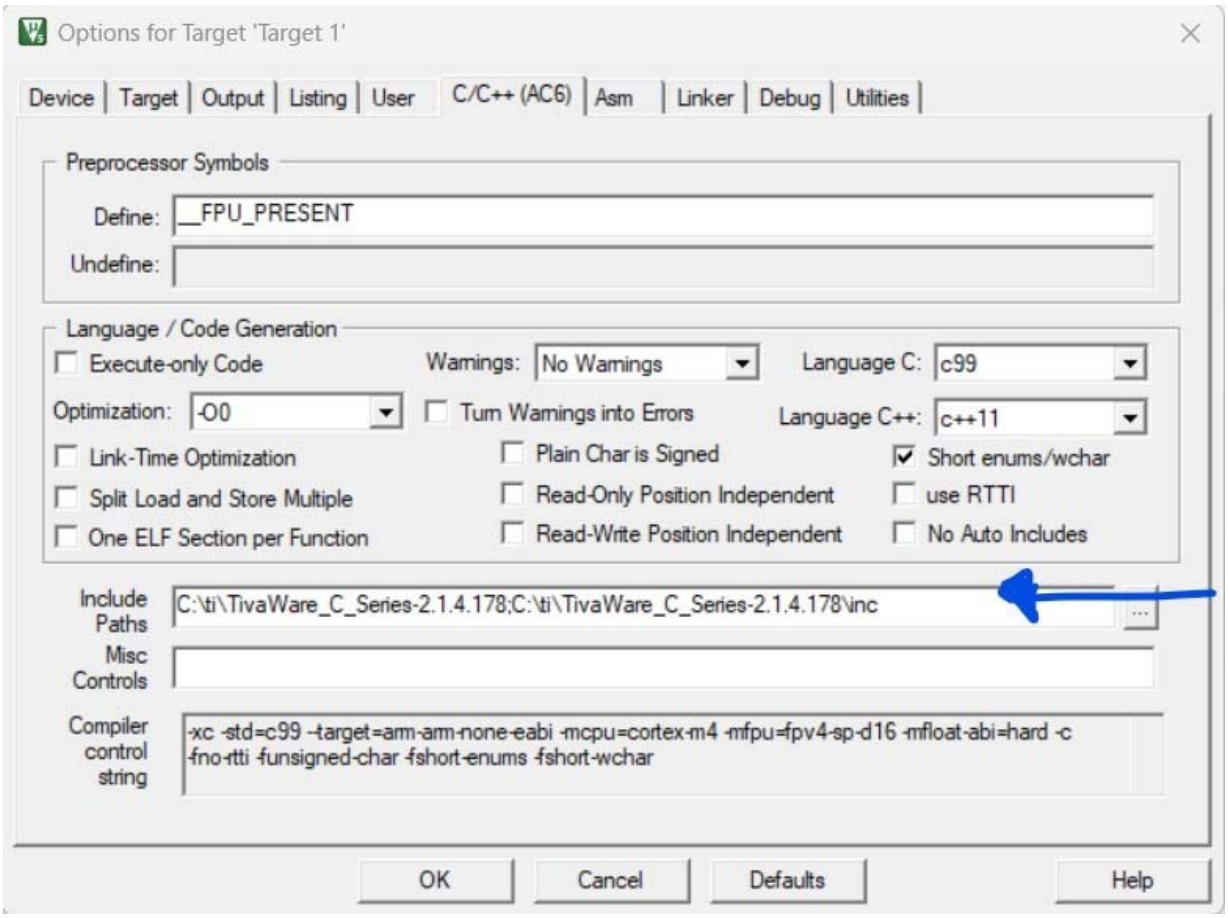
In main #include "TM4C123GH6PM.h"

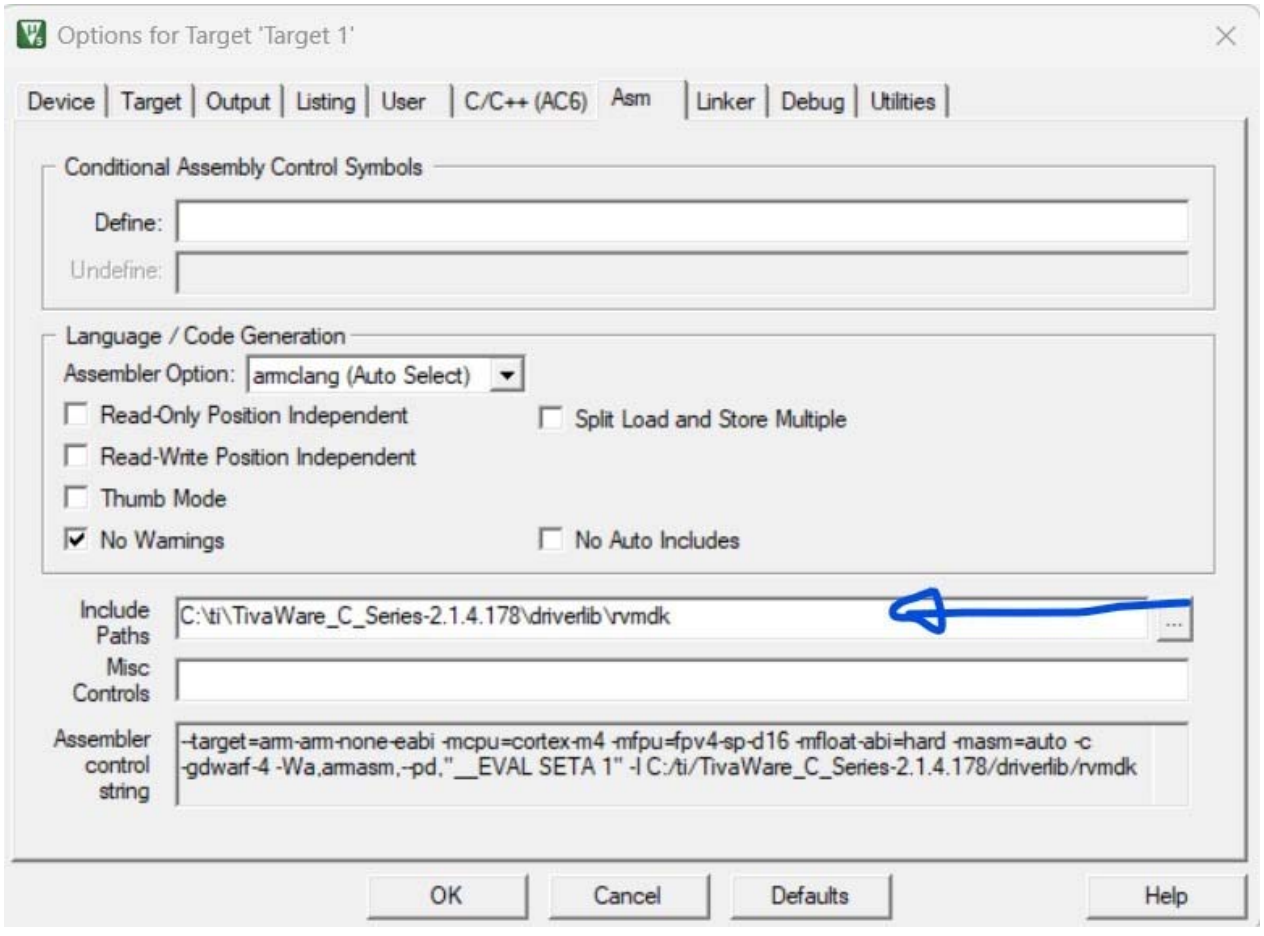
Add new main.c

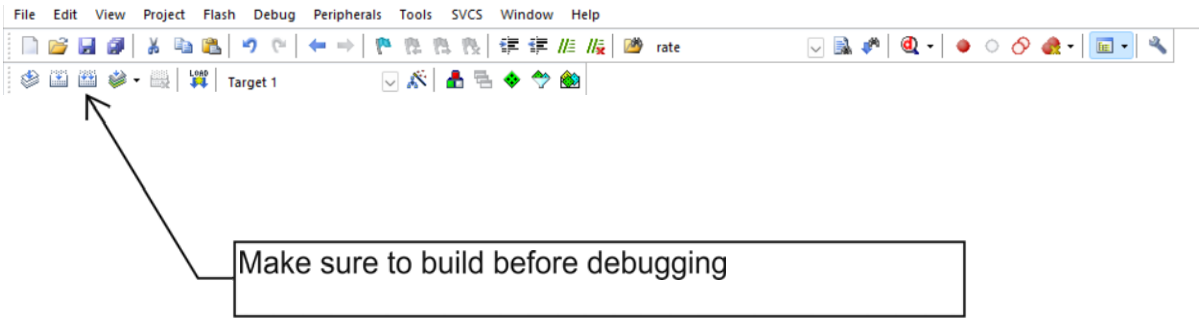
Add Existing drivelib.lib (Tivaware)

drivelib.lib path: C:\ti\TivaWare_C_Series-2.1.4.178\driverlib\rvmdk\driverlib.lib

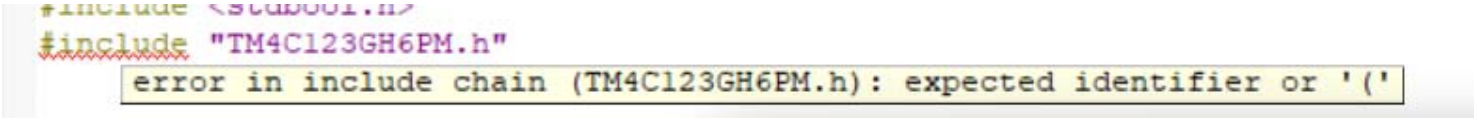
Include Tivaware paths in preprocessor and assembler

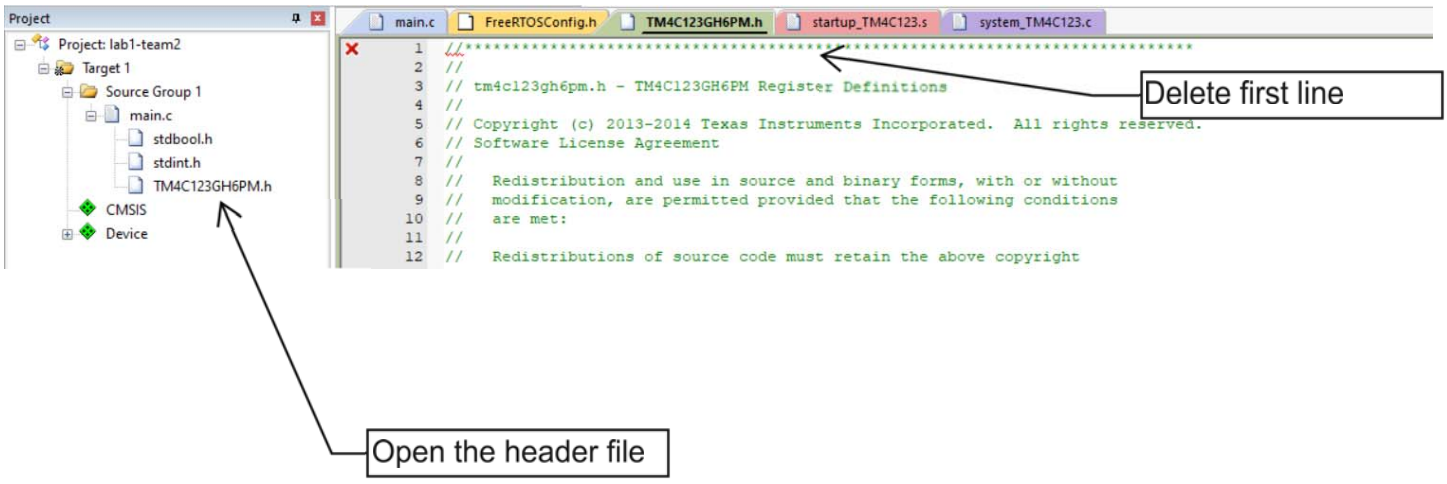






If this error appeared in main.c when including:





AIN SHAMS UNIVERSITY
FACULTY OF ENGINEERING
Credit Hours Programs

Summer Semester **2024**

CSE 411/345/CSE 347: RT Embedded Systems

Lab 1: Manual Tasks Switch

Goals of this Lab:

Real time operating systems are composed of various tasks. Tasks are infinite loop functions that serve certain functionalities. The operating system Kernel switches the processor resources (processing time, hardware peripherals, etc.) between tasks. The goal of this lab is to learn how to manually switch between two tasks.

Step 1: Create the tasks:

- Create two infinite loops functions; one toggles the red LED each 500 ms on Port F and the other toggles the blue LED each 1000 ms.
- The tasks code should be written with the help of the TivaWare™ Peripheral Driver Library.
- The structure of a infinite loop function should be:

```
void function_name (void)
{
    //initializing code to be executed only once.
    for (;;)
    {
      // infinite loop code to be executed repeatedly
    }
}
```

Step 2: Write the Systick handler:

- Initialize the Systick to fire an interrupt each 100 ms.
- The Systick interrupt handler should increment a counter variable to keep track of the time passed.
  PS. This counter could be used in toggling the LEDs.

Step 3: Write the program main:

- Write the program main such that it:
  - Calls the functions that initializes the Port F and the Systick.
  - Calls the task that toggle the red LED.
  - Contains an infinite loop that should never be reached.

Step 4: Run the code in debugging mode and manually switches between the tasks:

1. Run the code in debugging mode.
2. Find the memory address of each of the two tasks with the help of the disassembly window.
3. Put a breakpoint in the Systick handler.
4. Find the return stack from the Systick handler with the help of the memory window and the stack pointer in the register window.
5. Change the value of program counter in the return stack to the address of the blue LED task.
6. Witness what happens.

Step 5: Repeat step 4 to switch back to red LED task.

AIN SHAMS UNIVERSITY
FACULTY OF ENGINEERING
Credit Hours Programs

***Summer Semester,* 2024**

CSE411: RT Embedded Systems

Lab 2: Creating task stacks for switching

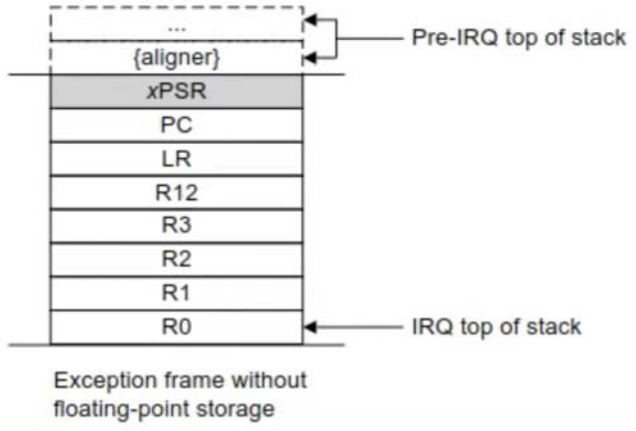
Goals of this Lab:

Real time operating systems are composed of various tasks. Tasks are infinite loop functions that serve certain functionalities. The operating system Kernel switches the processor resources (processing time, hardware peripherals, etc.) between tasks. The goal of this lab is to implement a code that does the context switch. This code has to switch between two tasks.

Introduction:

What we did last lab which is manually switching by changing the value of the PC is not quite legal. It would also not work in larger applications. Even more, it would cause some really serious problems. We also still need to have the code run automatically without interfering every time we need to switch tasks.

To do so we need to look at what happens in the context switching exactly, and what happens exactly in the stacks. As per the TM4C datasheet please find the frames that are added to the stack while doing the context switch of an exception. The screenshot attached is valid if we turned off the Floating point Unit (FPU) from the project settings.



ARM exception frame layout (without the FPU)

NOTE: The stack pointer grows upward in the ARM which means the top of stack would be the address with the lower value. A hint : You may need to flip the image above to match the stack if the values of the stack are put in ascending order.

Task 1: Use the code created in Lab 1

Open the lab 1 progress. Make sure you have the systick timer with the interrupt. Make sure also you have two tasks one of the RED led and the other for the BLUE LED.

Task 2: Create a stack for each task:

Create a stack that could be used to store the values used for each task. You can create the stack as an array of 40 Uint32 elements because the stack is mainly a piece of memory we need to reserve assembled. We also need a pointer that shall point to the top of the stack exactly like what the SP does.

```
/* Example for Task 1 */
uint32_t stack_RedBlinky[40];
uint32_t *sp_RedBlinky = &stack_RedBlinky[40];  /* The stack pointer is initialized to pint one word after the stack because the stack grows down that is from the end of the stack array to its beginning */
```

Now do the same step above also for the second task.

Task 3: In the main function, fill the stacks created with the data to make it look as if it was preempted by an interrupt

According to the ARM exception frame layout attached above. We shall start from the high memory end of the stack because it grows from high to low memory.

Refer to the datasheets for the positions of the bits.

Step 1: Pre decrement the pointer to get to the first address location, we then need to write the THUMB bit in the PSR register

Step 2: Pre decrement the pointer again this time to write the PC. In C you can get the start address of a function using it's name (ex: &fun1)
You need also to cast the pointer to make sure it fits inside the pointer (ex: (uint32_t)&fun1)

Step 3: The rest of the registers would not really matter a lot in the switch so for testing we shall set them to values we know so that we can make sure these values are already copied to the registers.

Step 4: Do steps 1 to 3 again, but this time for the second task. (Blinky LED 2)

Step 5: Create an infinite loop at the end of the main function.

Task 4: Seeing the stacks in the memory view

Step 1: Look at how your stacks are allocated in memory and make sure you have the correct values in each place. (Screenshot_1)

Step 2: Put both of your created stack pointers in the watch window. (Screenshot_1)

Step 3: Put a breakpoint in your systick Interrupt handler. Then wait till the debugger hits the breakpoint.

Step 4: Manually change the value in the SP register to the stack pointer of Blinky1,remove the breakpoint, and watch as the debugger goes to the Blinky1 function. (Screenshot_2)

Step 5: Now to switch to Blinky2, put back the breakpoint in the ISR. Now, we first need to take the current value in the SP register and put it inside the created stack pointer for Blinky 1 because that value is now the top of stack for Blinky1. After that we can put the value of Blinky2 stack pointer in the SP register. (Screenshot 3)

AIN SHAMS UNIVERSITY
FACULTY OF ENGINEERING
Credit Hours Programs

SummerSemester 2024

CSE 411: RT Embedded Systems

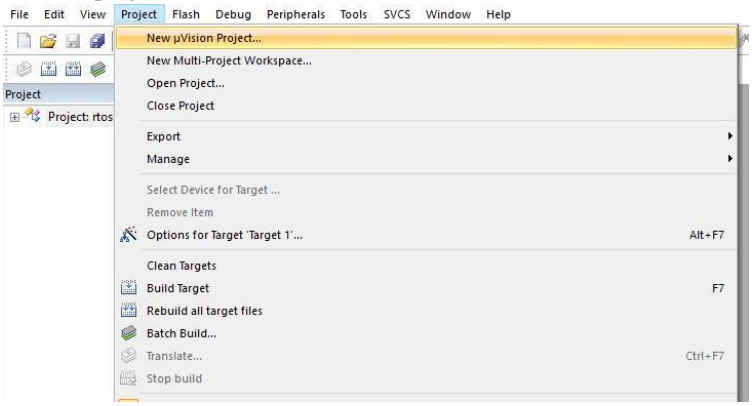
Lab. 3: Introduction to FreeRTOS

Goals of this Lab:

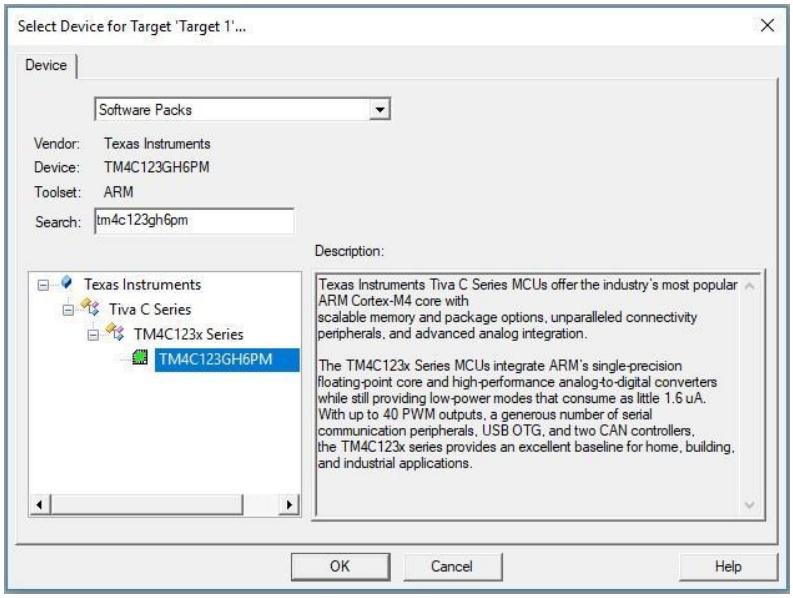
- Create a new project on kiel with FreeRTOS integrated in the project
- Adding tasks in FreeRTOS

Task 1: Initialize the Micro-controller:

1- Create a new project on Keil

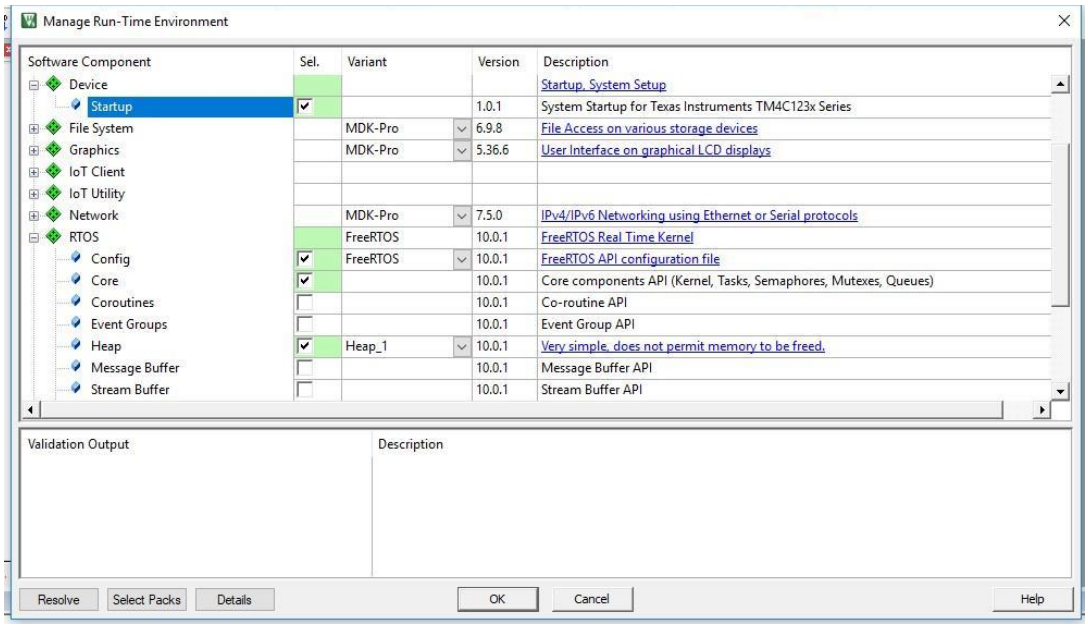


2- Save your project in the desired place, and choose the target

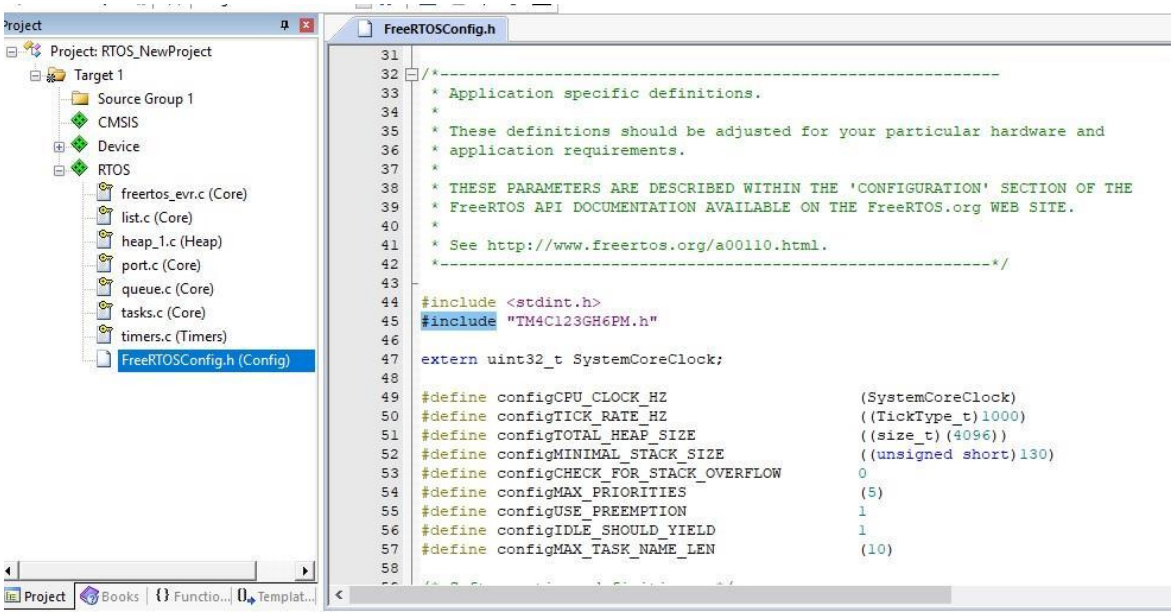


3- In the Manage Run Time Environment window, check the following boxes:
   - CMSIS -> Core
   - Device -> startup - RTOS -> Config Core

Heap (select Heap_1 from the drop down box)
Timers



4- In the Project menu on the left side Open the RTOS folder and open the file FreeRTOSConfig.h and include “TM4C123GH6PM.h” inside it.



5- Create a new main.c file, and start your code.

Task 2: Initialize Tasks

- Create a function that contains a for loop without conditions, ie: for(; ;), make sure that this function is implemented outside the main function.
- Inside the for Loop, implement a function that every time it get executed it toggles the RED LED
- Using the xTaskCreate API create a task and assign the function to it as per section 1.4 in the attached document

Richard Barry-Using the FreeRTOS Real Time Kernel - A Practical Guide - Cortex-M3 Edition.pdf

- After creating the task start the task by calling the scheduler function “vTaskStartScheduler();”
- Just before you start running the code, call the vTaskDelay() after calling the function that toggles the RED LED. Assign a 1000ms delay in the task

AIN SHAMS UNIVERSITY
FACULTY OF ENGINEERING
Credit Hours Programs

Summer Semester **2024**

CSE 411: RT Embedded Systems

Lab. 4: Multiple Tasks

Goals of this Lab:

- Creating multiple tasks using FreeRTOS.
- Drawing the expected timing diagram of the tasks.
- Validating the timing expected timing diagram.

Task 1: Create a new project on Keil and set it up:

Refer to lab sheet 2 to setup a new FreeRTOS project using Keil.

Task 2: Initialize the Tasks:

- Create three tasks; one toggle the red LED on the Tiva c launchpad each second, one toggle the blue LED each two seconds and the final task toggle the green LED each three seconds.
- Draw by hand the expected timing diagram of the tasks.
- Draw by hand the expected timing diagram of the output color of the LEDs.
- Upload the code on the Tiva c launchpad and validate the timing diagrams.

Task 3: Lab Report:

A lab report is required from each of you, the report should include:

- The expected timing diagram of the tasks.
- The expected timing diagram of the output color of the LEDs.
- If the actual LED color was matching the expected timing diagram, and if not suggest what may have caused the mismatch.
- Would giving the three tasks different priority level have changed the output of the system or not, and why?
- While implanting the tasks did you use vTaskDelay() API or vTaskDelayUntil() API, would it have made a difference and why?

AIN SHAMS UNIVERSITY
FACULTY OF ENGINEERING
Credit Hours Programs

Summer Semester **2024**

CSE411: RT Embedded Systems

Lab. 5: Adding Muliplte Tasks

Goals of this Lab:

- Create multiple tasks
- Adding the Idle Task for the freeRtos

Task 1: Idle Task:

- When the OS is not doing any task, it falls back to this task when there are no tasks being executed at the time
- Start by adding #define configUSE_IDLE_HOOK in the FreeRTOSConfig.h, this will allow the idle task hook.
- Write the Idle Task function according to the following prototype:
  - void vApplicationIdleHook();
- In the Idle task, toggle the blue LED
- This task would be the same as the other tasks but you will not need to add xTaskCreate

Task 2: Task Creation

- Start by enabling all LEDs on
- Create 2 tasks that has the same priority
- Task 1 shall toggle red LED only - 1000 ms
- Task 2 shall toggle green LED only - 2000 ms
- Draw the expected time line for the tasks
- Draw the observed time line for the LEDs

Ain Shams University Faculty of Engineering

CSE411: RT Embedded Systems

Lab 6 - Queues in FreeRTOS

Lab Objective:

- In this lab, you should get introduced to the usage of the queue
- Know the mechanism of the queue work
- Implement the queue in FreeRTOS.

Lab Mission:

1) Create an Init Task to Initizalize the UART0 and 2 push buttons.

```
void InitTask(void *){
SYSCTL_RCGCUART_R|=0X0001; //enable clock to uart0
SYSCTL_RCGCGPIO_R |= 0x00000001; //enable clock to port A (A0 Rx A1 Tx)
UART0_CTL_R = 0x0; //disable uart rx and tx
UART0_CC_R=0X0; //use SystemCoreClock (50MHZ)
UART0_IBRD_R=27; //115200 baud rate integer part
UART0_FBRD_R=8;  //fraction part
UART0_LCRH_R=(0x3<<5); //length 8 bit
GPIO_PORTA_AFSEL_R|=0X03;  //alternate func for A0 A1
GPIO_PORTA_PCTL_R=0X011; //use a uart
GPIO_PORTA_DEN_R|=0X03; //digital enable
UART0_CTL_R=0x0301; //enable uart rx and tx
//115200 1 stopbit no parity no fifo
}
```

FreeRtosConfig.h

```
extern uint32_t SystemCoreClock;
#endif

/* Constants that describe the hardware and memory usage. */
#define configCPU_CLOCK_HZ                      (SystemCoreClock)
#define configTICK_RATE_HZ                      ((TickType_t)1000)
```

System_Tm4c123.h

```
//---------------------- Clock Configuration ----------------------------------
//
//  <e> Clock Configuration
//          <i> Uncheck this box to skip the clock configuration.
//
// The following controls whether the system clock is configured in the
// SystemInit() function.  If it is defined to be 1 then the system clock
// will be configured according to the macros in the rest of this file.
// If it is defined to be 0, then the system clock configuration is bypassed.
//
#define CLOCK_SETUP 1
```

2) Create a Queue using the FreeRTOS APIs

3) Create a Task that checks the first Push Button and increments a counter

```
void BTN1_CHK_TASK(void *){
....
....
}
```

4) Create a Task that checks the Second Push Button and send the counter to the created queue, and then sets the counter back to 0

```
void BTN2_CHK_TASK(void *){
....
....
}
```

5) Create a UART Task that periodically checks data in the queue, and send that data if available via UART to PC.

```
void UART_TASK(void *){
....
....
}
```

Ain Shams University Faculty of Engineering

CSE411: RT Embedded Systems

Lab 7 - Semaphores in FreeRTOS

Lab Objective:

- In this lab, you should get introduced to the usage of semaphores
- Know the working mechanism of the semaphore
- Implement the semaphore in FreeRTOS.

Lab Mission:

1) Create an Init Task to Initizalize the UART0 and 1 push buttons.

```
void InitTask(void *){

....
....
}
```

See Lab 6 for Uart Initialization

2) Create a Queue using the FreeRTOS APIs
3) Create a Counting Semaphore using the FreeRTOS APIs

4) Create a Task that checks the Push Button and gives the semaphore after sending a value to the queue

```
void BTN1_CHK_TASK(void *){
        static uint8 IncrementingCounter;
....
....
}
```

5) Create a UART Task that periodically tries to get the semaphore, and send that data via UART to PC if the semaphore was taken successfully.

```
void UART_TASK(void *){
....
....
}
```

Ain Shams University Faculty of Engineering
CSE411: RT Embedded Systems

Lab 8 - Mutex in FreeRTOS

Lab Objective:

- In this Lab we will cover the following points.
  - Mutex and how they are implemented
  - Difference between mutex and semaphores
  - Uses of mutex and semaphores

Lab Mission:

- In this lab we aim to know the difference between Mutex and Semaphores and how to use Mutex.

1) Create an Init Task to Initizalize the UART0 , 1 push button and LED

```
void InitTask(void *)
{
        ....
}
```

2) Configure the push button to generate interrupt when pressed
3) Create a Mutex using the FreeRTOS API " xSemaphoreCreateMutex "
4) Create a Binary Semaphore
5) Create a continuous Task called " CounterTask " that prints "This is the CounterTask " then counts from 0 – 10 and prints each count on the console.

6) Create a periodic Task with higher priority called " LedTogglerTask " which is unlocked by Binary Semaphore Given from ISR of the push button.
   The Task should Write on UART " This is LedToggler Task " when given the semaphore then toggles the Led then sleeps for 500 ms.
7) The ISR should give the binary semaphore.
8) Expected behavior:
   This is the Counter task
   0
   1
   2
   3
   4
   5
   6
   7
   8
   9
   10
   This is the LedToggler task
   0
   1
   2
   ..

Task 2:

Mimic Deadlock and priority inheritance examples shown in lab...and show in debugging session (using breakpoint) what happened.