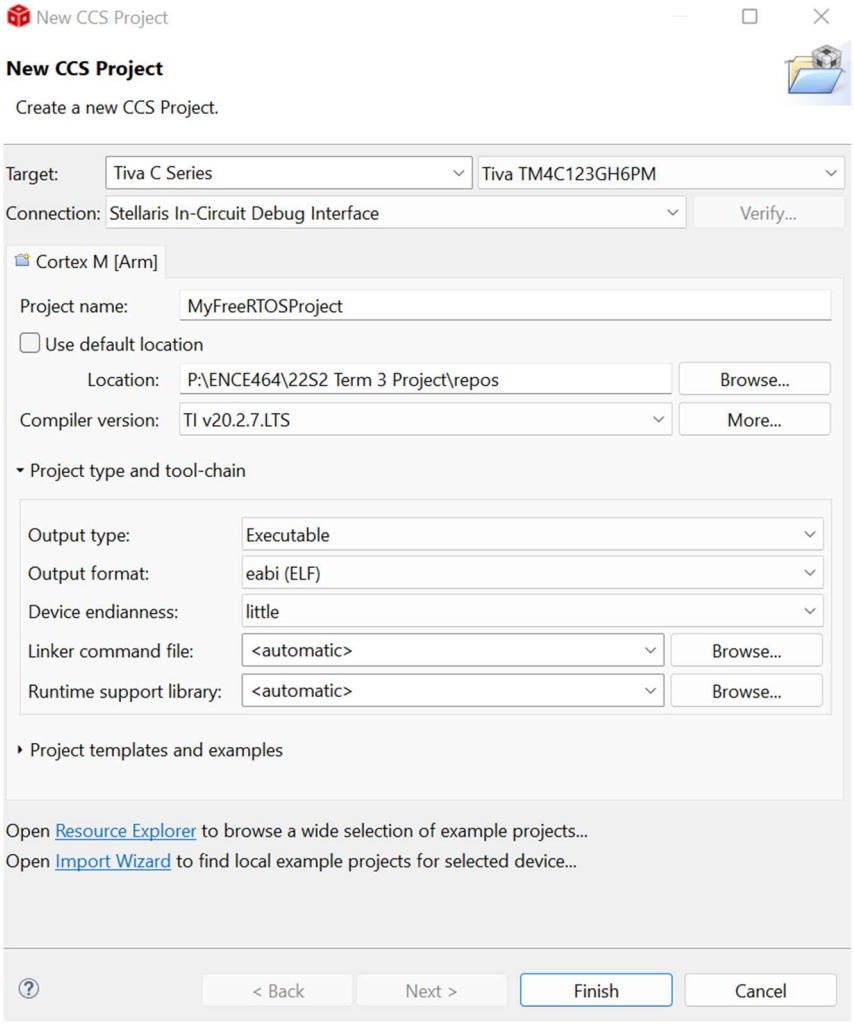
University of Canterbury – ENCE464

FreeRTOS Project Setup with CCS

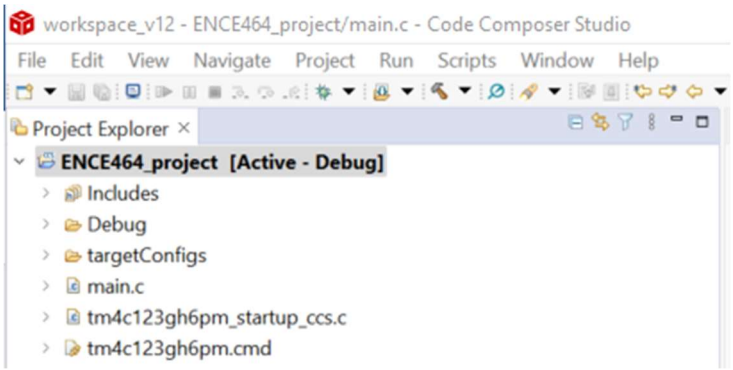
Contributors: Fredy Youssif, Steve Weddell, Harry Mander, Matthew Pike
Last modified: 18 July 2023

1 CREATING A PROJECT FROM SCRATCH

1. Create a Code Composer Studio (CCS) workspace if you haven't already. If you are using UC computers, it is recommended to create the workspace in P drive.
2. Create a new Code Composer Studio (CCS) Project with "Tiva TM4C123GH6PM" as Target and "Stellaris In-Circuit Debug Interface" as Connection.
    a. Give it an appropriate name
    b. Select an appropriate location. **Avoid paths with spaces such as "My Documents".**



3. Click "Finish". You should soon be able to see the newly created project on the Project Explorer pane on the left side of the CCS window.

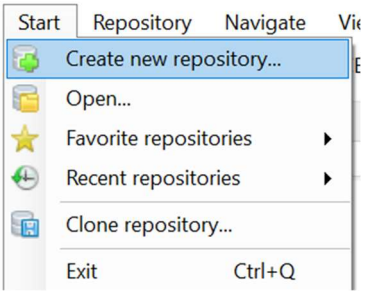


You should be able to build cleanly at this point.

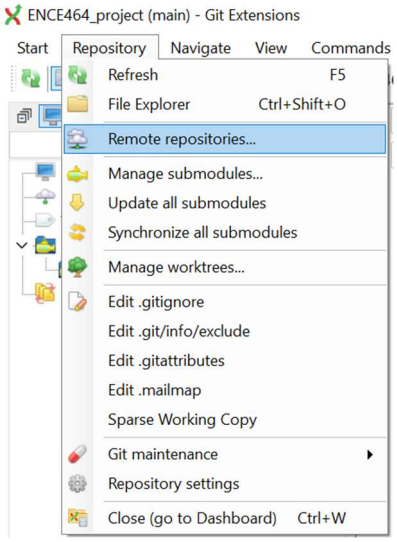
2 INITIALISING YOUR GIT REPO

Note: In this project, you can use your preferred git client[1]. However, in this guide, I will demo the use of Git Extensions[2] (Windows only) as it will be new for most of you. If you are using Mac, a good alternative is SourceTree[3].

1. Initialise a new repository in your project folder (equivalent to `git init`)
    a. In Git Extensions, click "Start" -> "Create new repository…"

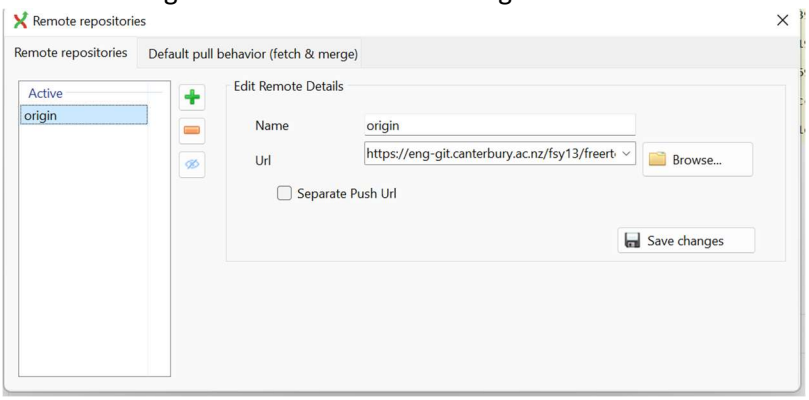


    b. Navigate to the project folder
    c. Keep the Repository type as Personal repository
    d. Hit Create
2. Navigate to your group's remote repository on eng-git and ensure that it is **EMPTY** (has no commits/read me file)[4]
3. Add remote with name "origin" (default) to link to your group's eng-git remote repo (equivalent to `git remote add origin <url>`)
    a. In Git Extensions, navigate to Repository -> Remote repositories…



[1] There are multiple options including git bash (command line), or GUIs such as Git Extensions, Code Composer Studio (a part of the IDE), SourceTree and many more. These GUIs act as wrappers for git bash so you can quickly execute complex git commands without having to memorise them.
[2] I would recommend using git extensions as it is user-friendly, relatively aesthetic and is completely free to use. This is now installed on all ESL computers. A guide for installation (for your PC) and how to use it can be found here: https://git-extensions-documentation.readthedocs.io/
[3] Although you will have to create a free Atlassian account to install it. Download link here: https://www.sourcetreeapp.com/
[4] Otherwise your newly created project will conflict with what's on the remote repo. If you are stuck here, please speak to Fredy or one of the TAs as this is important to get right.

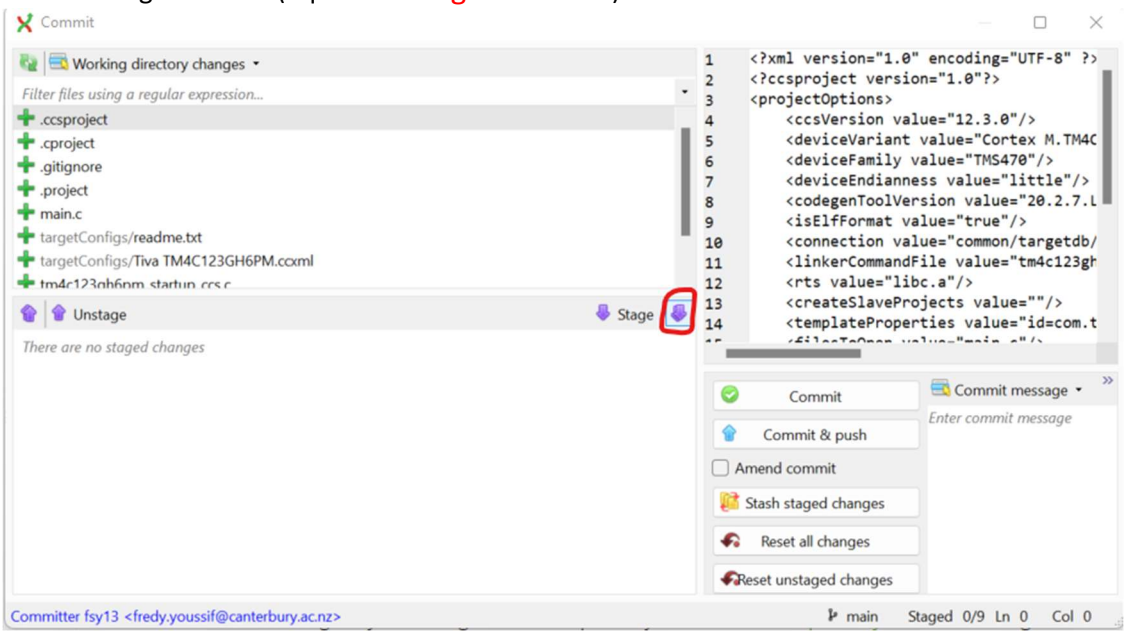
b. In the name field, type “origin”
c. In the URL field, copy your group’s eng-git repository URL
d. Hit Save changes. It should look something like this:



e. Exit the Remote repositories window

4. Add a .gitignore file to the project folder. I suggest the following content:
```
/Debug
/.settings
/.launches
```
5. Once you’ve added the git ignore, you should have a clean “working directory” and you should be able to commit everything that git “sees” as your first commit.
6. Make first commit:
   a. In git extensions, click the “Commit (X)” button at the top

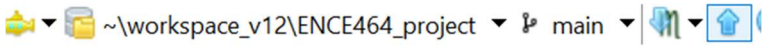
   b. Hit the “stage all” icon (equivalent to `git add -a`)



   c. Type your commit message in the box on the bottom right and hit commit (equivalent to `git commit -m "<msg>"`)
7. Push to origin (equivalent to `git push -u origin <branch>`[5])

[5] The default branch when you initialise a new repo is likely “master”, but you should double check this using git status or by looking at the active branch at the top of the GitExtensions window, next to the project directory (mine is “main”).

a. Click the push icon (up arrow) at the top of Git Extensions, next to the commit button



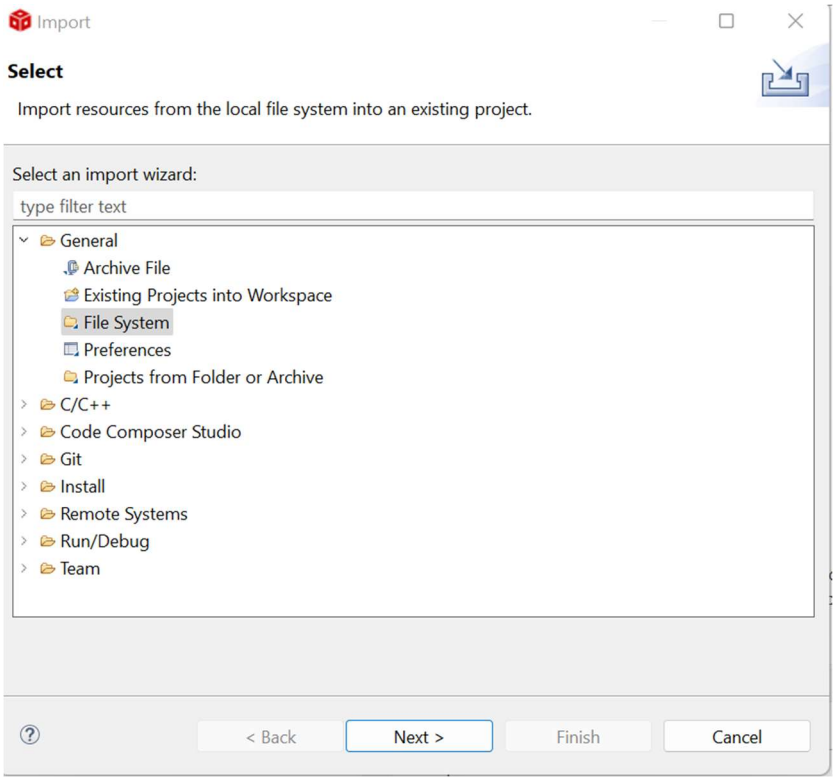
b. Accept all the dialogs that come up – unless it is asking you if you want to do a force push. Speak to a TA if this happened.

8. You should be able to find the project content if you navigate to your repo on eng-git.

3 IMPORTING FREERTOS FILES

1. Download the latest release of the FreeRTOS Kernel files to a different location from the project created above.
   https://github.com/FreeRTOS/FreeRTOS-Kernel/releases/tag/V10.5.1
2. Expand the zip file
3. In the CSS project, make a new folder and call it “FreeRTOS”. The project folder structure should now look something like this:

4. Right click the “FreeRTOS” folder and select “Import” and then “Import..”
5. Under “General”, select “File System” and click “Next”



6. Click “Browse”, navigate to the unzipped FreeRTOS Kernel directory (which contains the “include” and “portable” subfolders, not t) and click “Select Folder.
7. Tick all the C files on the right
8. On the left, expand the top-level directory and tick the 'include' folder
9. Expand the 'portable' folder, expand 'CCS', tick 'ARM_CM4F' folder.
10. In 'portable' folder, click on 'MemMang' and tick ONLY ONE heap_x.c file on the right. I suggest 'heap_4.c' (it is a reasonable generic choice; more info on website)
11. Hit Finish

4 Setting up Project Properties

1. Open project properties by right clicking the project in the “Project Explorer” and choosing “Properties”
2. In Build > ARM Compiler > Include options:
    a. Add directory path:
    ${PROJECT_ROOT}/FreeRTOS/include
    b. Add directory path:
    ${PROJECT_ROOT}/FreeRTOS/portable/CCS/ARM_CM4F
3. OPTIONAL (for possible performance improvement): In Build > ARM Compiler > Predefined Symbols:
    a. Add TARGET_IS_TM4C123_RB2
    This enables use of the ROM version of driverlib calls. Be sure to check the writing on the chip to get correct revision. Mine was 7, which is revision B2 in the datasheet.
4. Add a FreeRTOSConfig.h file to the top-level project folder. You can find an example one[6,7] on this repo by Harry Mander: https://eng-git.canterbury.ac.nz/hma199/tivaware-freertos/
5. You should now be able to build cleanly. If successful, commit and push the changes to git.

5 Adding FreeRTOS ISRs into vector table

The inclusion of three interrupts is needed for FreeRTOS. These are:

- vPortSVCHandler
- xPortSysTickhandler
- xPortPendSVHandler

1. Open the 'tm4c123gh6pm_startup_ccs.c' source file
2. At the 'External declarations section for interrupt handlers used by the application', add:

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

[6] One of the FreeRTOSConfig.h defines in this repo references a vAssertCalled function. FreeRTOS highly recommends using configAssert() to pick up errors during development. If using this setup, vAssertCalled function is required to be defined somewhere in your code. Refer to the “apps” directory in the example repo to see how this function is defined.

[7] The FreeRTOSConfig.h in this repo defines a useful macro “pdTM4C_RTOS_INTERRUPT_PRIORITY” to prevent priority clashes when setting up ISRs in the project. This is a common pitfall when using FreeRTOS on ARM Cortex-M3 and -M4. Refer to the “apps” directory in the example repo to see how this macro is used.

```
extern void xPortSysTickHandler(void);
```

3. Scroll down to the vector table and enter the three interrupt names (only) in the corresponding entries. In other words, replace the "IntDefaultHandler" handler, so that your edited version of your vector table in your 'tm4c123gh6pm_startup_ccs.c' source file looks like the following:

```
#pragma DATA_SECTION(g_pfnVectors, ".intvecs")
void (* const g_pfnVectors[])(void) =
{
    (void (*)(void))((uint32_t)&__STACK_TOP),
                                            // The initial stack pointer
    ResetISR,                               // The reset handler
    NmiSR,                                  // The NMI handler
    FaultISR,                               // The hard fault handler
    IntDefaultHandler,                      // The MPU fault handler
    IntDefaultHandler,                      // The bus fault handler
    IntDefaultHandler,                      // The usage fault handler
    0,                                      // Reserved
    0,                                      // Reserved
    0,                                      // Reserved
    0,                                      // Reserved
    vPortSVCHandler,                        // SVCall handler
    IntDefaultHandler,                      // Debug monitor handler
    0,                                      // Reserved
    xPortPendSVHandler,                     // The PendSV handler
    xPortSysTickHandler,                    // The SysTick handler
    IntDefaultHandler,                      // GPIO Port A
    IntDefaultHandler,                      // GPIO Port B
    IntDefaultHandler,                      // GPIO Port C
    IntDefaultHandler,                      // GPIO Port D
```

4. In main.c, add the following includes:

```
#include <FreeRTOS.h>
#include <task.h>
```

5. You should now be able to build cleanly. If successful, commit and push the changes to git.

Note however that if you get a "stack overflow Hook" error at this point, EITHER, go to your 'FreeRTOSConfig.h' file and change the parameter for 'configCHECK_FOR_STACK_OVERFLOW' to 0, OR, add a vApplicationStackOverflowHook(.....) function and place this in you main.c implementation. If needed, Google these key words to find out more.

6 CREATING YOUR FIRST DUMMY TASK

1. In main.c, add the following function prototype on top of the main() function:

```
static void NullTaskFunc(void *);
```

2. In the main function, add the following. Note, you can CTRL + click (windows) or CMD + click (mac) the function on CCS to view its prototype.

```
if (pdTRUE != xTaskCreate(NullTaskFunc, "Null Task", 32, NULL, 4, NULL))
{
      while(1); // Oh no! Must not have had enough memory to create task.
}
vTaskStartScheduler(); // Start FreeRTOS!!

// Should never get here since the RTOS should never "exit".
```

```
while(1);
```

3. Below main(), add:

```
// Our RTOS "task" - does absolute jack squat
static void NullTaskFunc(void *pvParameters)
{
    while(1)
    {
        // With this task always delaying, the RTOS Idle Task runs almost
        // all the time.
        vTaskDelay(10000);
    }
}
```

4. You should be able to build cleanly and RUN at this point. If successful, commit and push the changes to git.

7 ADDING TIVAWARE

1. Tivaware C series files should be found in the ESL labs in the following directory:
   C:\ti\TivaWare_C_Series-X.X.X.XXX
   If you are using your own PC, the easiest thing to do is to copy this folder across to the same location on your PC.
2. Project -> Properties:
   a. In Resource > Linked Resources (Path Variables tab):
      Add a new path variable named TIVAWARE with location:
      ${TI_PRODUCTS_DIR}\TivaWare_C_Series-X.X.X.XXX[8,9]
      (or simply use the browse to locate the TIVAWARE version you use)
   b. In Build > ARM Compiler > Include options:
      Add directory path: ${TIVAWARE}
   c. In Build > ARM Linker > File Search Path:
      Add include library file: ${TIVAWARE}\driverlib\ccs\Debug\driverlib.lib
3. In main.c, add the folling includes:

```
#include <stdbool.h>
#include <stdint.h>

#include "inc/hw_memmap.h"
#include "inc/hw_types.h"

#include "driverlib/adc.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/pwm.h"
#include "driverlib/sysctl.h"
```

[8] You will need to replace the Xs with whatever version number you have
[9] If "TI_PRODUCTS_DIR" path is not defined in the linked resources, add a new path variable with that name and give it the value of the location you've placed the TivaWare folder in i.e. "C:\ti".

4. In main() function, add to the start:

```
// Set the clock rate to 80 MHz
SysCtlClockSet (SYSCTL_SYSDIV_2_5 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
                SYSCTL_XTAL_16MHZ);
```

5. You should be able to build cleanly and run at this point. If successful, commit and push the changes to git.

8 BLINK AN LED

As you know by now, blinking an LED is the hello world of embedded systems. You can optionally keep this task running as you develop your project to provide physical feedback that the RTOS is running.

1. In main.c, add the following function prototype on top of the main() function:

```
static void BlinkLED(void *);
```

2. In main() function, add the following.

```
// For LED blinky task - initialize GPIO port F and then pin #1 (red) for
output
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
while (!SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOF)) ; // busy-wait until
GPIOF's bus clock is ready
GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1); // PF_1 as output
// doesn't need too much drive strength as the RGB LEDs on the TM4C123
launchpad are switched via N-type transistors
GPIOPadConfigSet(GPIO_PORTF_BASE, GPIO_PIN_1, GPIO_STRENGTH_4MA,
GPIO_PIN_TYPE_STD);
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 0); // off by default
if (pdTRUE != xTaskCreate(BlinkLED, "Blinker", 32, (void *)1, 4, NULL))
{ // (void *)1 is our pvParameters for our task func specifying PF_1
    while (1) ; // error creating task, out of memory?
}
```

3. Below main(), add:

```
//Blinky function
//
void BlinkLED(void *pvParameters) {
    /* While pvParameters is technically a pointer, a pointer is nothing
    * more than an unsigned integer of size equal to the architecture's
    * memory address bus width, which is 32-bits in ARM. We're abusing
    * the parameter then to hold a simple integer value. Could also have
    * used this as a pointer to a memory location holding the value, but
    * our method uses less memory.
    */
    const unsigned int whichLed = (unsigned int)pvParameters;

    // TivaWare GPIO calls require the pin# as a binary bitmask,
    // not a simple number. Alternately, we could have passed the
    // bitmask into pvParameters instead of a simple number.
```

```
    const uint8_t whichBit = 1 << whichLed;
    uint8_t currentValue = 0;
    while (1)
    {
        // XOR toggles the bit on/off each time this runs.
        currentValue ^= whichBit;
        GPIOPinWrite(GPIO_PORTF_BASE, whichBit, currentValue);

        // Suspend this task (so others may run) for 125ms
        // or as close as we can get with the current RTOS tick setting.
        // (vTaskDelay takes scheduler ticks as its parameter, so use the
        // pdMS_TO_TICKS macro to convert milliseconds to ticks.)
        vTaskDelay(pdMS_TO_TICKS(125));
    }
    // No way to kill this blinky task unless another task has an
    // xTaskHandle reference to it and can use vTaskDelete() to purge it.
}
```

You should now be able to build, run and visually test this program. All going well, you are now ready to start development! As always, don't forget to commit and push the changes to git.

You can optionally keep the LED flashing as you develop your code. It can be used to indicate whether your RTOS is running.