User Guide for FreeRTOS on the CY8CKIT-050 (PSoC 5LP Developers Kit)

v1.0

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
Phone (USA): 800.858.1810
Phone (Intl): 408.943.2600
http://www.cypress.com

Copyrights

Disclaimer

Flash Code Protection

Cypress products meet the specifications contained in their particular Cypress PSoC Datasheets. Cypress believes that its family of PSoC products is one of the most secure families of its kind on the market today, regardless of how they are used. There may be methods, unknown to Cypress, that can breach the code protection features. Any of these methods, to our knowledge, would be dishonest and possibly illegal. Neither Cypress nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Cypress is willing to work with the customer who is concerned about the integrity of their code. Code protection is constantly evolving. We at Cypress are committed to continuously improving the code protection features of our products.

Table of Contents

Chapters

Figures

CY8CKIT-050 Kit

These example projects are designed to run on the CY8CKIT-050 (PSoC 5LP Development Kit) from Cypress Semiconductor. A full description of the kit, along with more example programs and ordering information, can be found at http://www.cypress.com/cy8ckit-050.

This kit is a development platform for the PSoC 5LP family of devices. It includes a CY8C5868AXI-035LP part with 256kB on-chip flash memory and 64kB SRAM. The kit hardware provides the following features.

- Power supply system
- Programming interface
- USB communications
- Boost convertor
- PSoC 5LP and related circuitry
- 32-kHz crystal
- 24-MHz crystal
- Port E (analog performance port) and port D (CapSense® or generic port)
- RS-232 communications interface
- Prototyping area
- Character LCD interface
- CapSense buttons and sliders

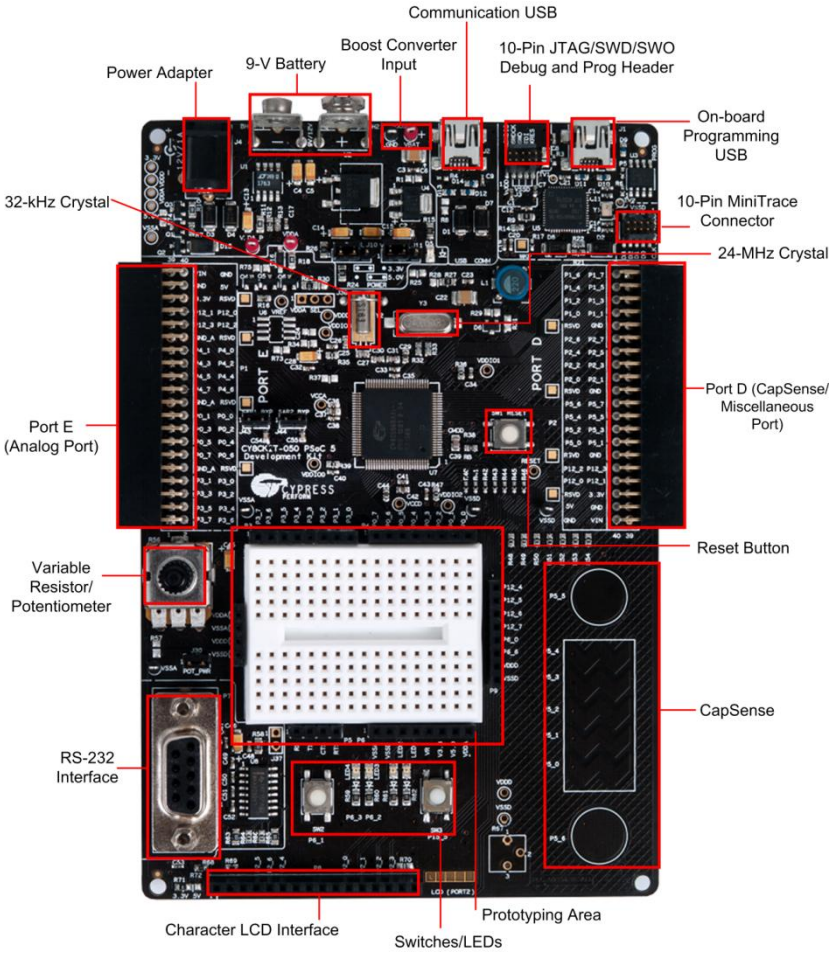


Figure 1: PSoC 5LP Development Kit

Hardware Tools

Programming and debug is available through the following hardware tools.

- PSoC Creator with the on-board debug connection (USB)
- PSoC Creator with the Cypress MiniProg3 kit

Software Tools

The applications are compatible with the following software IDEs.

- PSoC Creator 3.0 and newer

Full source code is provided for all projects and you are encouraged to modify the designs to learn and experiment with FreeRTOS on a PSoC 5LP device.

Compiler Toolchain

This document is included in two downloadable images – CY8CKIT-050-GNU and CY8CKIT-050-MDK, which use the ARM GNU GCC 4.7.3 and ARM's Microcontroller Developers Kit compiler toolchain respectively.

CY8CKIT-042-GNU

In this package the ARM GNU GCC 4.7.3 is used to build all projects. This compiler is automatically installed with PSoC Creator 3.0.

CY8CKIT-042-MDK

In this package the Microcontroller Developers Kit (MDK) compiler from ARM Ltd. is used to build all projects. This compiler not included in the PSoC Creator product but is available for download at https://www.keil.com/demo/eval/arm.htm.

Using PSoC Creator

To explore the applications in PSoC Creator open the workspace file "CY8CKIT-050-GCC.cywrk" or "CY8CKIT-042-MDK.cywrk". This workspace includes the following projects.

- Demo.cyprj
- NewDesign.cyprj

Programming/debugging is provided through either the on-board USB connector J1 or the Cypress MiniProg3 debug adapter (not provided in the CY8CKIT-050 kit). Third-party debug adapters cannot be used with PSoC Creator 3.0.

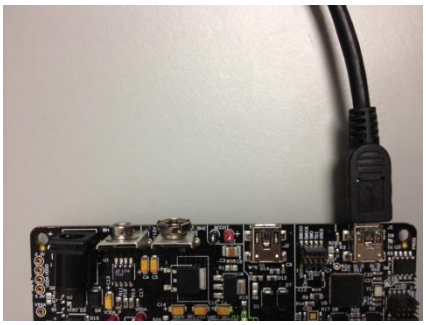


Figure 2: USB programming/debug connector, located next to the 10-pin debug header

Board Support Package

PSoC 5LP is a unique device that combines an ARM Cortex-M3 CPU with arrays of configurable analog and digital functionality. This allows PSoC to be programmed with an amazing range (and amount) of peripheral functionality.

These examples include a generic configuration of the PSoC device that supports the following functions and interfaces.

- 2x SAR (successive approximation) ADCs (12-bit)
- Delta-Sigma ADC (20-bit)
- 1x general purpose Voltage DAC (8-bit)
- 2x Analog Comparators with 2x Voltage DACs (reference sources)
- I2C
- UART
- 2x PWM (16-bit)
- 2x Timers (32-bit)
- Character LCD

All of these functions are supported by a rich set of APIs, which are described in the accompanying Board Support Package (BSP) document.

Board Setup

In order to run the applications, make the following hardware connections.

- Connect the 2x16 character LCD to header J8.
- Attach a jumper wire from P0_0 to the VR header, as shown in Figure 5, below.

Modifying the BSP

The BSP for PSoC devices is generated directly from the configuration of the device. When a design is built in PSoC Creator, the APIs to interface with the peripherals are generated by the tool. For example, if you create a design that uses an ADC, the APIs to start, stop and read the ADC conversion results are automatically added to the project. As a result it is a simple task to extend the BSP provided with these projects or to create your own from scratch.

Launch PSoC Creator and open the NewDesign project. In the NewDesign.cycsh file you will see a number of pages of peripheral functions, all of which are discussed in the BSP document. Feel free to add new functions or change how the existing design works to suit your needs. A simple re-build is all that is needed to re-generate the APIs and create your own, custom BSP.

Example – “Demo”

This is a great place to start learning about FreeRTOS and PSoC. The demo shows off some of the features of PSoC and the kit and how they can be managed and used in a multi-tasking application. In particular, a CapSense monitoring task is used to show how multiple tasks can be CapSense clients without repeating all the update/scan/check code.

Running the Demo

Download the demo and play with it before diving into the code. The program presents a set of five demo applications (implemented as tasks) that can be selected from a simple menu interface on the LCD. Use the CapSense buttons to browse the demos. Use button SW2 to start a demo and SW3 to finish it. Some demos also make use of SW2 and the CapSense buttons and sliders.

Figure 3: CapSense Buttons (pin P5_5 and P5_6) and Slider (pins P5_0..P5_4)

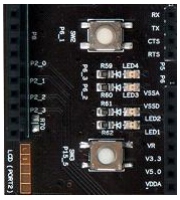


Figure 4: SW2 (pin P6_1) and SW3 (pin P15_5)

Glowing PWMs

This is the simplest demo. It actually requires no application code at all! A pair of PWMs (“PWM_1” and “PWM_2”) are used to drive LED3 (pin P6_2) and LED4 (pin P6_3) such that their intensities vary periodically.

The intensity variation is achieved by ANDing the output of a pair of PWMs with slightly different periods. This causes a gradual change in the combined duty cycle of the output signal. Note that for LED3, the results are ANDed and, for LED4, they are actually XORed. These logical functions have truth tables that are the mirror of each other and so the effect you see is that as one LED gets brighter the other dims, and vice versa.

ADC SAR

This demo uses one of the SAR ADCs on the PSoC device to measure the voltage across a variable resistor on the board. The SAR used is “ADC_SAR_1”, which is connected to the potentiometer via pin P6_5. Rotate the potentiometer to see the voltage update on the LCD.

Note that there is a second SAR ADC available in the device, connected to pin P0_1, which is next to the breadboard area. Look for “ADC_SAR_2” in the BSP document for details on how to use this ADC.

ADC DelSig

This is a similar demo to the ADC SAR, except it uses the Delta-Sigma ADC. “ADC_DelSig” is connected to pin P0_0, near the breadboard area, so it is not dedicated to the potentiometer. You can still use the potentiometer for the demo if you connect P0_0 to the “VR” connection with a jumper wire.

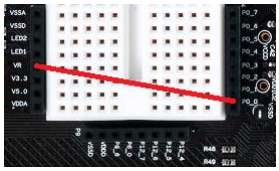


Figure 5: Connect pin P0_0 to VR

CapSense

This demo asks you to “Touch Something!” Press the buttons and move your finger along the slider to see the input displayed on the LCD.

Digital Scope

This is another bit of fun where we pretend the LCD is a digital oscilloscope. Press SW2 to provide input to the scope and watch the display scroll by. You can vary the speed of the scope by moving your finger along the CapSense slider.

Theory of Operation

The Demo project uses a message queue and event groups. The queue is used to share CapSense input with multiple tasks. The events are logically associated with their specific tasks and are an ideal way to schedule the execution of each demo from the task that displays the demo “menu”. The application comprises the following tasks.

Main_Task – creates all the other tasks and displays the menu of demos

CapSense_Monitor_Task – periodically reads the CapSense input and posts it to a queue

<Demo>_Task – the five tasks that implement the actual demos

Main_Task

This task is created in main() with a low priority. It runs when there is no demo active and the CapSense task is not scanning. Main_Task starts by creating the CapSense queue, the event groups, and then all the other tasks. There are two event groups. Main_Task_Event is used to control Main_Task and contains a single event. Demo_Events contains an event for each one of the tasks (the demo number defines the bit position of the event in the group).

Main_Task then prints user instructions to the LCD and enters its forever-loop. In the loop it checks for a CapSense message and, if a button has been pressed, it updates the LCD with the next demo name (moves up or down the list of demos). If the user presses SW2, to start a demo, then the currently-displayed demo is started by signaling its task event.

Note that, to prevent Main_Task from running while a demo is suspended (for example, if the demo executes a delay API), Main_Task immediately suspends itself by waiting on its own task event. This prevents any contention for the LCD resource or the risk of two demos running simultaneously.

<Demo>_Task

Each of the demos is implemented as a medium priority task. When they are created from Main_Task, being a higher priority, they immediately execute. Obviously, the demos should not all run at once and so the task code for each one waits on its own event and is blocked. The demo tasks are, therefore, “released” by Main_Task when it signals the appropriate event.

The demo code is, naturally, found in each of the demo’s forever-loops. Each one includes appropriate start and stop code for the peripheral hardware used in the demo so that other tasks can safely use the same resources (good neighbor policy).

When the user presses SW2, to stop a demo, the demo task responds by signaling the Main_Task event. This makes Main_Task ready but does not let it run just yet because Main_Task is low priority. The demo task then suspends itself by waiting on its own event and Main_Task is finally free to run again. This synchronization of Main_Task and the demo tasks is a classic “bilateral synchronization” where a pair of resources is used to ensure the tasks are never running/ready at the same time.

CapSense_Monitor_Task

Capsense_Monitor_Task is the highest priority of the application tasks. It scans the buttons and slider on the board and, if activity is detected, the event is posted to the queue. Tasks that require input from CapSense simply ask for queue data. The task executes every 20ms, which allows other tasks plenty of time to execute (without the user noticing that the “application code” is not running) and is fast enough to be very responsive to user input on the buttons and slider.

Each “packet” of CapSense data (union CS_PACKET defined in OS_Resources.h) is 32 bits in size and so the queue’s payload is the actual CapSense data (rather than a pointer to a buffer). This simplifies the code that has to interpret the data transferred.

Note that the “packet” is contained in a C union. The union comprises four 8-bit fields overlaid with a 32-bit integer. The four 8-bit fields correspond to button0, button1, a flag indicating slider activity, and the slider value. The 32-bit field enables easy extraction of the packet data from the queue, without the need to copy all the 8-bit fields manually.

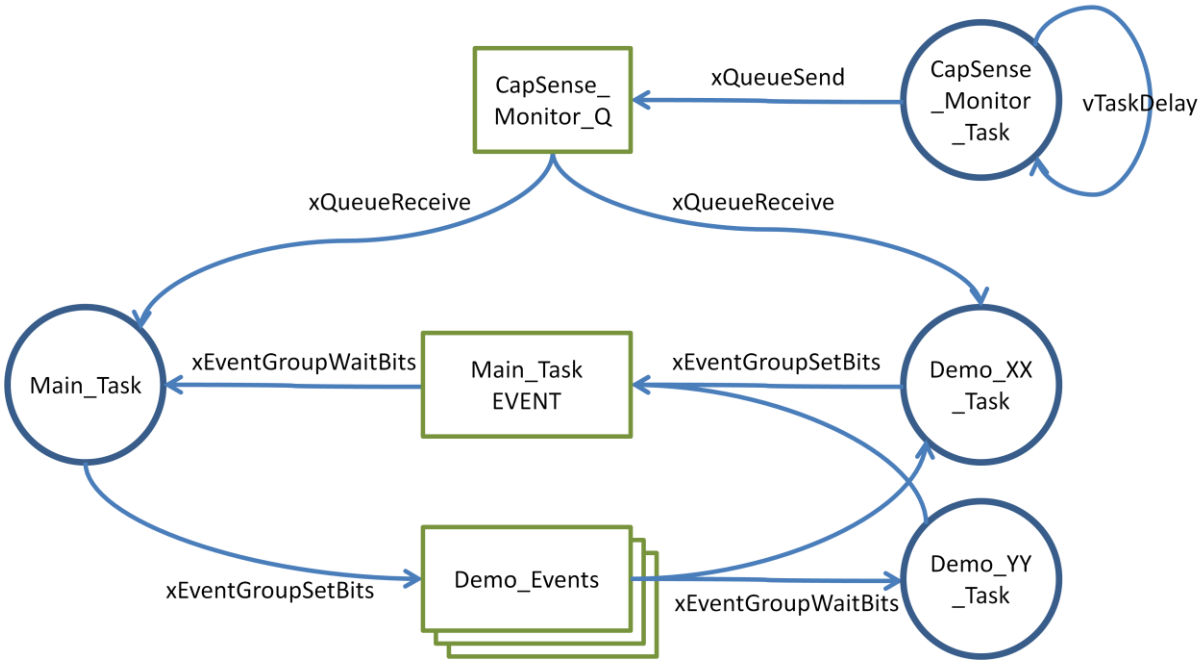


Figure 6: State transitions for Demo project

Example – “NewDesign”

This is a near-empty project that is ready for you to start developing your own applications. It includes all the BSP files and start-up code for the RTOS. The demo code is just a single task (“Main_Task”) that blinks an LED every second. Replace our two-line application with more interesting code of your own to get started today!

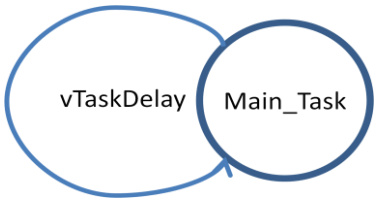


Figure 7: State transitions for NewDesign project

Adding FreeRTOS to a PSoC Creator Project

The instructions above all assume the user starts from a PSoC Creator project with the RTOS already set up. What if you wish to add an RTOS to an existing PSoC design? Follow these instructions to add FreeRTOS to a project in PSoC Creator. Note that you can use either MDK or GCC compilers, PSoC 4 or PSoC 5LP, and DEBUG or RELEASE configurations.

Getting the FreeRTOS Software

These instructions apply to FreeRTOS v8.0.0. It can be downloaded from here: http://sourceforge.net/projects/freertos/. The downloaded file is an executable ZIP – extract it to a convenient location “near” your project directory.

Note that the FreeRTOS author(s) have done an excellent job of separating user and OS files in their distribution. Also, they keep device-specifics (like communications drivers and LEDs) to a minimum. As a result the “Source” directory in the installed SW contains all you need to get FreeRTOS working (plus the configuration file discussed below) and there should never be a need to modify those files. Indeed, if disk space is an issue, you might choose to delete unnecessary compiler and architecture support.

Instructions on writing applications can be found here: http://www.freertos.org/RTOS.html

Add FreeRTOS Files

The following are guidelines, not hard requirements. You do need to add the source files to the project or they will not build but include files are found through the build settings, so they are optional. Cypress recommends adding everything, as described, so you can see all the files (and learning about the OS becomes easier). You may also safely modify the structure of the folders to suit your taste.

1. Create a “FreeRTOS” folder
    - Add the C source files from FreeRTOS\Source
        - croutine.c
        - event_groups.c
        - list.c
        - queue.c
        - tasks.c
        - timers.c
2. Create an “include” folder under ”FreeRTOS”
    - Add the C header files from FreeRTOS\Source\include
        - croutine.h
        - event_groups.h
        - FreeRTOS.h
        - list.h

- mpu_wrappers.h
- portable.h
- projdefs.h
- queue.h
- semphr.h
- StackMacros.h
- task.h
- timers.h

3. Create a “portable” folder under ”FreeRTOS”
4. Create a “MemMang” folder under “portable”
   - Add “heap_1.c” file[1] from FreeRTOS\Source\MemMang
5. Create a compiler-architecture[2] (e.g. “GCC-CM3” or “MDK-CM3”) folder under “portable”
   - Add the C source and header files from FreeRTOS\compiler[3]\architecture[4]
     - port.c
     - portmacro.h

Build Settings

6. Add the following directories to the compiler include path (use the appropriate compiler and architecture choices for your project)
   - <relative path>\FreeRTOS\Source\include
   - <relative path>\FreeRTOS\Source\portable\<compiler>\<architecture>

Stack and Heap Settings

FreeRTOS defines its own heap and allocates task stacks from that. As a result it is usually best to reduce the stack and heap settings in the PSoC Creator project. The stack is only used until the RTOS starts (100 bytes is usually sufficient stack) and the heap is typically unnecessary.

7. Set the Stack size to 100 bytes in the CYDWR file
8. Set the Heap size to 0 bytes in the CYDWR file

---

[1] There are 4 heap_?.c files. Each offers increasingly sophisticated support for memory allocation. For a simple demo the heap_1.c file is adequate, but note that it does not support the reuse of heap memory after the deletion of tasks or resources.

[2] The port of FreeRTOS can support GCC or the MDK compiler as well as PSoC 4 and PSoC 5LP. It is recommended to use the following folder names: GCC-CM0, GCC-CM3, MDK-CM0 and MDK-CM3. Note that the MDK files are actually in the RVDS directory.

[3] The compiler directories are “GCC” and “RVDS” (for the MDK compiler).

[4] The architecture directories are “ARM-CM0” and “ARM-CM3”.

Initialization Code

In the main() function – or any other file / function that runs before you start the OS with vTaskStartScheduler() – you need to add some initialization code.

9. Add a handler function for malloc fails (heap full)

```
void vApplicationMallocFailedHook( void )
{
      /* The heap space has been exceeded. */
      taskDISABLE_INTERRUPTS();
      while( 1 )
    {
        /* Do nothing - this is a placeholder for a breakpoint */
    }
}
```

10. Add a handler function for stack overflow

```
void vApplicationStackOverflowHook( xTaskHandle pxTask, signed char
*pcTaskName )
{
      /* The stack space has been exceeded for a task */
      taskDISABLE_INTERRUPTS();
      while( 1 )
    {
        /* Do nothing - this is a placeholder for a breakpoint */
    }
}
```

11. Add defines and declarations for the OS exception Handlers

```
/* Declaration of NVIC base vector for FreeRTOS exception handling */
#define CORTEX_INTERRUPT_BASE           (16)

/* Declarations of the exception handlers for FreeRTOS */
extern void xPortPendSVHandler(void);
extern void xPortSysTickHandler(void);
extern void vPortSVCHandler(void);
```

12. Install the exception handlers

```
 /* Handler for Cortex Supervisor Call (SVC, formerly SWI) - address 11 */
 CyIntSetSysVector( CORTEX_INTERRUPT_BASE + SVCall_IRQn,
     (cyisraddress)vPortSVCHandler );

 /* Handler for Cortex PendSV Call - address 14 */
 CyIntSetSysVector( CORTEX_INTERRUPT_BASE + PendSV_IRQn,
     (cyisraddress)xPortPendSVHandler );

 /* Handler for Cortex SYSTICK - address 15 */
 CyIntSetSysVector( CORTEX_INTERRUPT_BASE + SysTick_IRQn,
           (cyisraddress)xPortSysTickHandler );
```

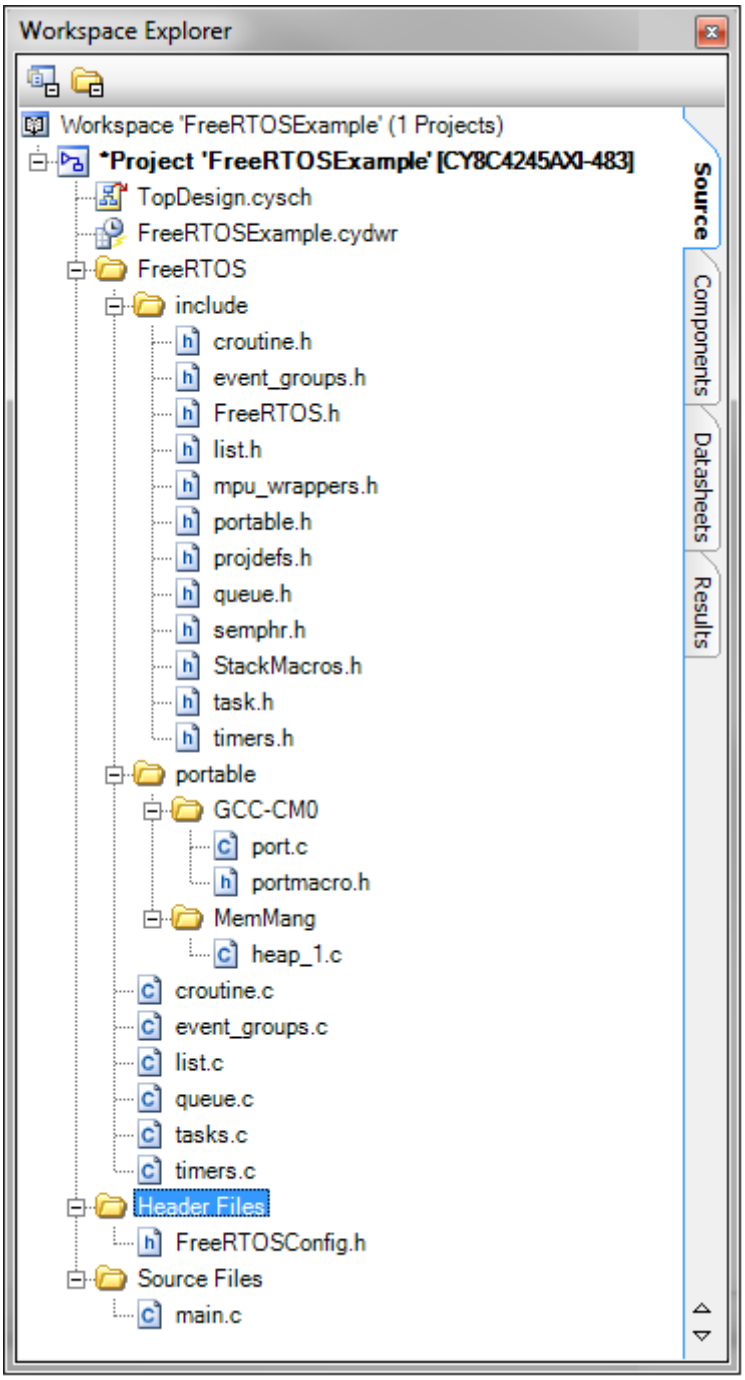
RTOS Configuration

The configuration of the RTOS is handled from a single header file that, unlike all other FreeRTOS files, is unique to the project (so you are free to modify it to your project's needs).

13. Add (or create) the attached FreeRTOSConfig.h file to the project under Header Files

Workspace Explorer with FreeRTOS

After the above steps your project should look a lot like this and be ready for you to launch the RTOS from main() by creating tasks and calling vTaskStartScheduler().



Revision History

Version	Changes	Reason for Changes / Impact
1.0	New document	Initial release

Note: the version number refers to the version of the demo package itself (the BSP, RTOS port code and applications). Letter-level revisions – e.g. from 1.0 to 1.0a – are document-only changes.