User Guide for FreeRTOS on the CY8CKIT-042 (PSoC 4 Pioneer 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

Table of Contents

Chapters

Figures

CY8CKIT-042 Kit

These example projects are designed to run on the CY8CKIT-042 (PSoC 4 Pioneer 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/go/cy8ckit-042.

This kit is a development platform for the PSoC 4 family of devices. It includes a CY8C4245AXI-483 part with 32kB on-chip flash memory and 4kB SRAM. The kit hardware provides the following features.

- Arduino™ shield-compatible headers
- Digilent® Pmod™ peripheral module header
- 5-element CapSense™ slider
- RGB LED
- Power and status LEDs
- Reset and user push-button switches
- Multiple power supply options: JTAG/SWD header, USB, external power via LDO, or Arduino shield
- 3.3V or 5.0V operation (jumper)
- JTAG/SWD debug connector 10 pin 0.1”
- On-board PSoC 5LP for serial I/O (I2C or UART) bridge to PC

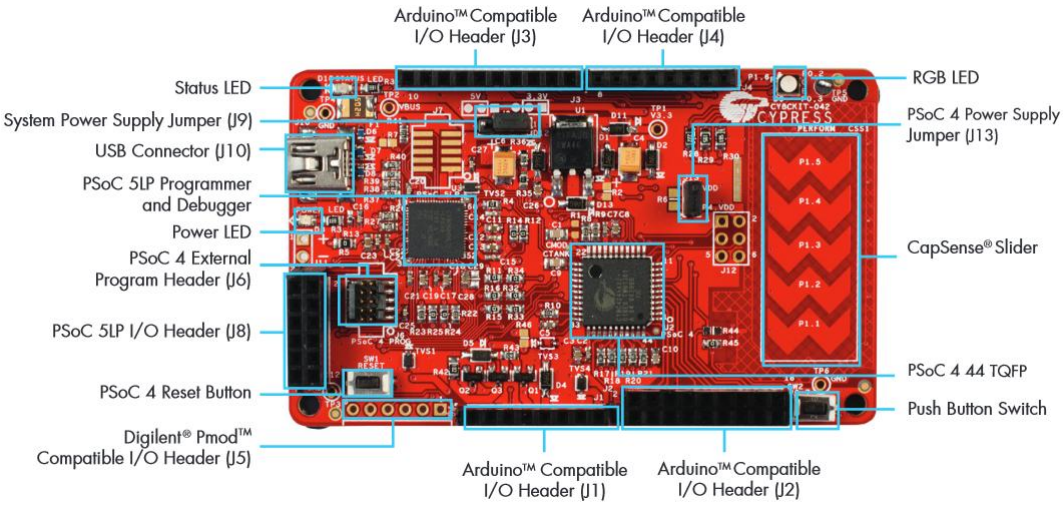


Figure 1: PSoC 4 Pioneer Kit

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 4 device.

Compiler Toolchain

This document is included in two downloadable images – CY8CKIT-042-GNU and CY8CKIT-042-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-042-GCC.cywsp" or "CY8CKIT-042-MDK.cywsp". This workspace includes the following projects.

- Demo.cyprj
- NewDesign.cyprj

Programming/debugging is provided through either the on-board USB connector J10 or the Cypress MiniProg3 debug adapter (not provided in the CY8CKIT-042 kit) on header J6. Third-party debug adapters cannot be used with PSoC Creator 3.0.

Board Support Package

PSoC 4 is a unique device that combines an ARM Cortex-M0 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.

- 3-channel SAR (successive approximation) ADC (12-bit)
- Analog Comparator supporting low-power wakeup
- General purpose current DAC (8-bit) [optional - disabled]
- 5-element CapSense slider [optional - enabled]
- I2C supporting low-power wakeup
- UART
- 3x PWM (16-bit)
- General-purpose Timer (16-bit)
- Hardware-debounced switch supporting low-power wakeup

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

Note: the PSoC project used in these examples can be quickly re-configured (in PSoC Creator) to swap between an IDAC- and a CapSense-supporting setup. This is described more fully in the accompanying BSP document. The projects ship with the DAC disabled and CapSense enabled.

Board Setup

In order to see the UART output, make the following hardware connections.

- Attach jumper wires from J3_09 to J8_10 and J3_10 to J8_09, as shown below.

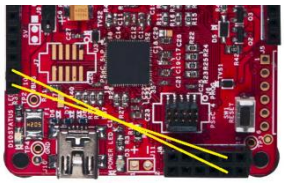


Figure 2: UART Connections for UART-USB Bridge

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 this design, a CapSense task monitors user interaction with the slider and an LED task updates the color and brightness of the RGB LED.

Running the Demo

Download the demo and play with it before diving into the code. The program lets you vary the color and brightness of the LED. The CapSense slider varies the brightness, by modifying the duty cycle of a PWM. The color can be rotated through red, green, blue and white by pressing the user button (SW2).

Theory of Operation

The Demo project uses a message queue and a task event. The queue is used to share CapSense input with the LED task. The task event is signaled from an ISR when SW2 is pressed. The application comprises the following tasks.

CapSense_Task – periodically reads the CapSense input and posts it to the queue

LED_Task – varies the color and brightness of the LEDs

SW2_ISR

This interrupt service routine runs when the SW2 button is pressed or released. It posts to the LED_Task event on a press (releases are ignored).

CapSense_Task

Capsense_Task scans the slider on the board and, if activity is detected, the slider position is posted to the queue. Tasks that require input from CapSense simply ask for data from the queue. The task executes every 10ms, 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.

LED_Task

This task uses the function update_PWMs() to control the brightness and color of the LED.

After starting the PWMs, the task enters a loop that makes non-blocking calls to xEventGroupWaitBits () and xQueueReceive (). The former reports button presses, which LED_Task uses to choose the output color. The latter reports activity on the CapSense slider, which LED_Task uses to vary the compare value of the PWMs on the LEDs.

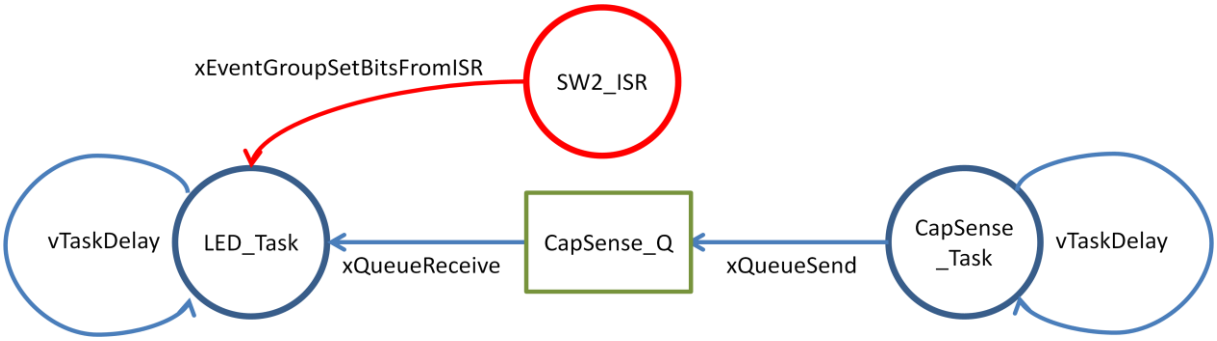


Figure 3: State transitions for Demo project

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 main-loop with more interesting code of your own to get started today!

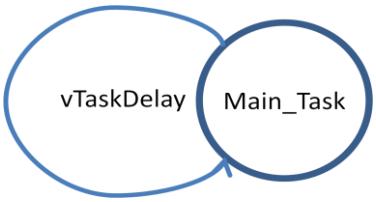


Figure 4: State transitions for NewDesign project

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-CM0”) 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\GCC\ARM_CM0

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 );
```

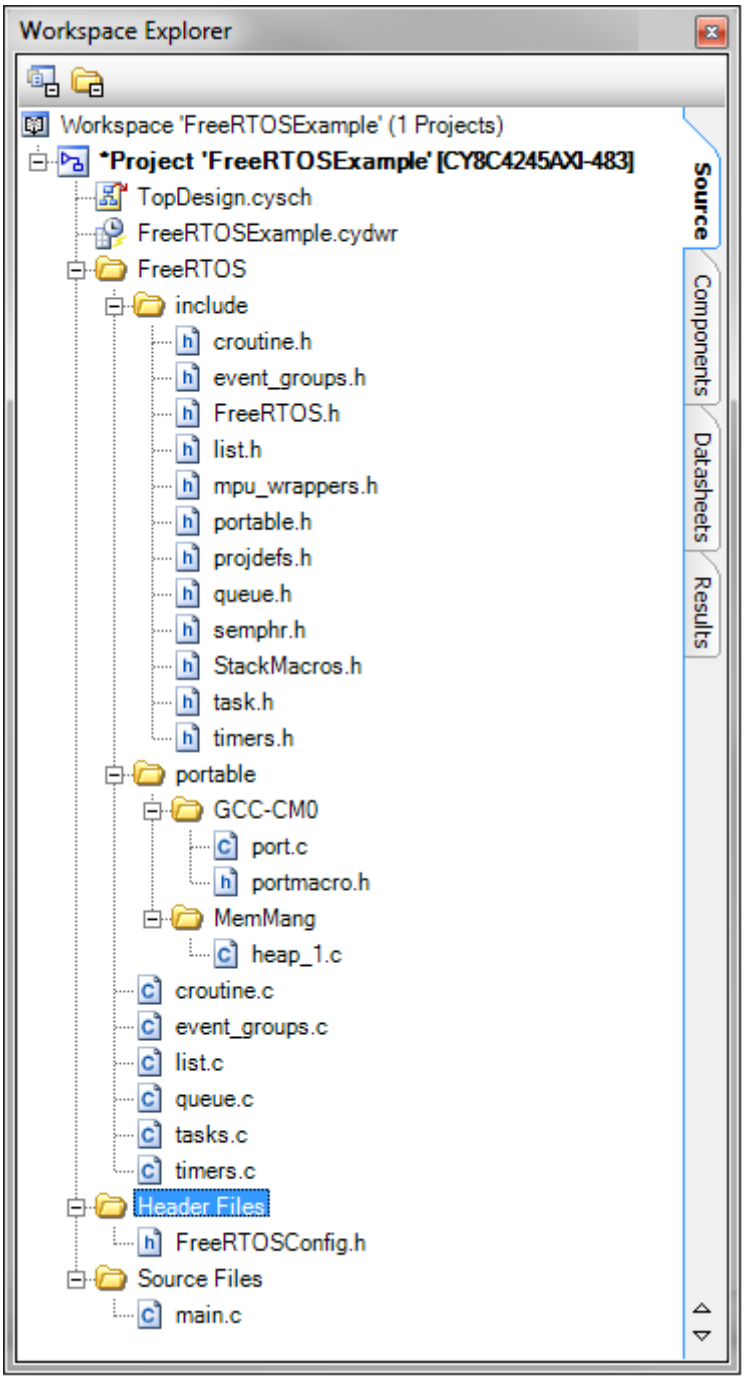
RTOS Configuration

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.