RENESAS
BIG IDEAS FOR EVERY SPACE

BASICS OF THE RENESAS SYNERGY™ PLATFORM

Richard Oed

2020.04

CHAPTER 9
INCLUDING A REAL-TIME OPERATING SYSTEM

CONTENTS

9 INCLUDING A REAL-TIME OPERATING SYSTEM

What you will learn in this chapter:

- What threads, semaphores and queues are, and how to use them.
- How to add threads and semaphores using the Synergy Configurator.
- How to toggle an LED with a push-button under RTOS control.

The exercise in the previous chapter already made use of a good part of the Renesas Synergy™ Software Package (SSP). In this chapter you will create a small application using the ThreadX® Real-Time Operating System (RTOS) utilizing a thread for the LED and a semaphore for synchronization with a push-button, and you will experience first-hand that only few steps are actually necessary for this.

You will create the complete project from scratch, so don't worry if you haven't done the previous labs.

9.1 Threads, Semaphores and Queues

Before we dive into actually doing this exercise, let's define some of the terms we will use in this and in the next chapter, to make sure that we all have a common understanding.

First we need to define the term *Thread*. If you are more accustomed to the term *Task*, just think of a thread being a kind of task. Some even use both terms alternately. When using an RTOS, the application running on the microcontroller will be broken down in several smaller, semi-independent chunks of code, with each one typically controlling a single aspect of it. And these small pieces are called *threads*. Multiple threads can exist within one application, but only one can be active at any given time, as Synergy microcontrollers are single core devices. Each one has its own stack space, an assigned priority in respect to the other threads in the application and can be in different states like ready, completed or sleep. In ThreadX®, the control block of a thread contains a member called `tx_thread_state`, from where the application can read the current state. Inter-thread signalling, synchronization or communication is performed by the means of semaphores, queues, mutexes or event flags.

A *semaphore* is a resource of the RTOS, which can be used for signalling events and for thread synchronisation (in a producer-consumer fashion). Using a semaphore allows the application to suspend a thread until the event occurs and the semaphore is posted. Without an RTOS, it would be necessary to constantly poll a flag variable or to create code to perform a certain action inside an interrupt service routine (ISR), blocking other interrupts for quite some time. Using semaphores allows exiting ISRs quickly and to defer the code to the associated thread.

ThreadX® provides 32-bit counting semaphores, and each semaphore has two basic operations associated: `tx_semaphore_get`, which will decrease the semaphore by one, and `tx_semaphore_set`, which increases the semaphore by one. Semaphores in ThreadX® are public resources. It's worth noting that semaphores, like all ThreadX® objects, cannot invoke suspension outside of a thread, for example inside the interrupt service routine. Therefore, all related functions must be called with `TX_NO_WAIT` only when called from an ISR.

The last term we need to talk about, even if we do not use one in this exercise, is the *queue*. But we will do so in the next chapter. Message queues are the primary method of inter-thread communication. One or more messages can reside inside a message queue. There is a special case: A queue with a single message is called *mailbox*. Allowed message sizes are 1 through 16 32-bit words. Larger messages need to be passed by a pointer, something implemented already in the SSP Messaging Framework (`sf_message`) and the size needs to be specified when the queue is created. There is no limit on the number of queues in ThreadX®. Messages are put into the queue using the `tx_queue_send()` function and read from the queue by `tx_queue_receive()`. New messages are placed at the end of a queue and received messages are removed from the front.

9.2 Adding a Thread to ThreadX® using e² studio

This exercise is again based on the Synergy Promotion Kit S5D9. This time, we will use the push-button SW4 on the top right corner of the board to signal an event to the application, which will toggle the green LED1 as response to it. For the implementation, we will use ThreadX® and the handling of the event will take place inside a thread and the notification of this thread will be done through a semaphore.

As usual, the first step is to create a new project using the Project Configurator, something you already exercised in chapters 4 and 8. To get started, go to *File → New → Synergy C/C++ Project*, select the *Renesas Synergy C Executable Project* template. Enter a project name on the next screen, for example *MyRtosProject*. Verify that the other settings are OK and click on *Next*. In the dialog now showing, select the *S5D9 PK* from the drop-down list of supported boards. Move on to the next screen, the *Project Template Selection* and select the *BSP* template.

Click on *Finish* and after the project has been generated by the configurator, e² studio will switch to the *Synergy Configuration* perspective. Go directly to the *Threads* tab. This tab will show one entry for *HAL/Common* in the *Threads* pane, containing the drivers for the Event Link Controller (ELC), the I/O port driver, the Clock Generation Controller (CGC), and the Factory MCU Information (FMI). The FMI-driver includes a generic API for reading the records from the Factory MCU Information Flash Table, providing information about the features and peripherals implemented on the microcontroller used. Click on the *New Thread* icon at the top of the pane, which will add a new thread.

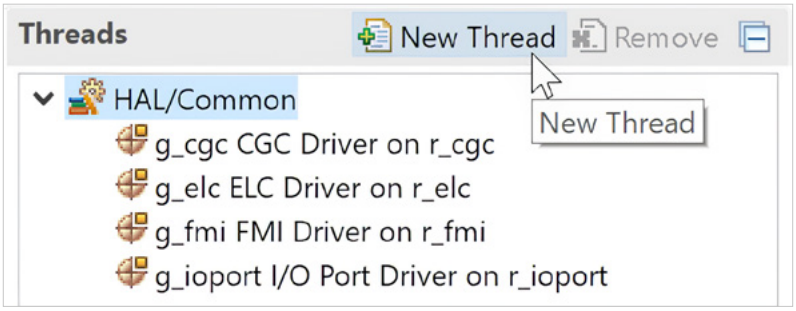

Figure 9-1: After the Synergy Configurator starts, only one thread will be shown. Clicking on the "New Thread" button will add another one

Now change the properties of the new thread in the *Properties* view: Rename the symbol to *led_thread* and the name to *LED Thread*. Leave the other properties at their default value. On the *LED Thread Stack* pane, click on the *New Stack* button icon and select *Driver → Input → External IRQ Driver on r_icu* (see Figure 9-2).

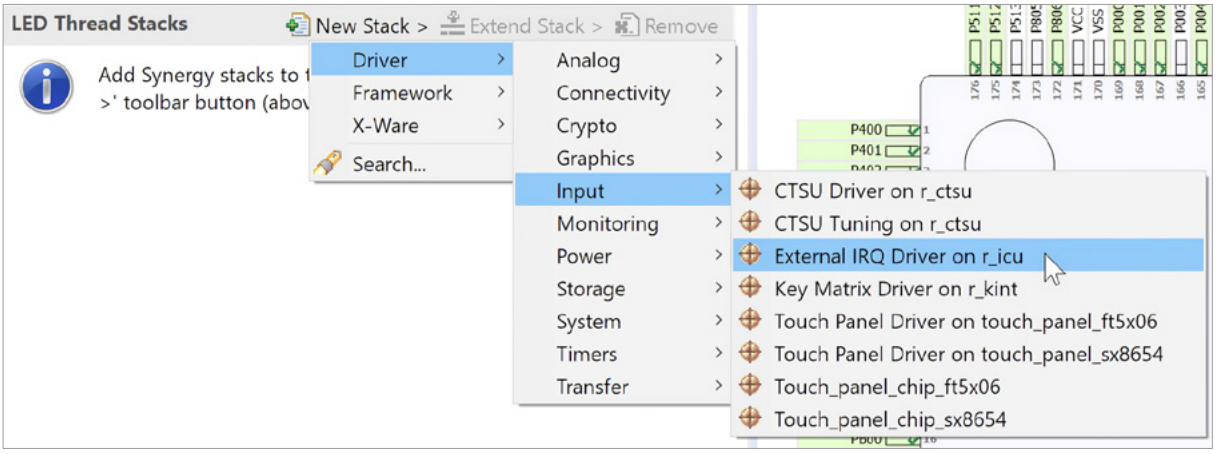

Figure 9-2: Adding a new driver takes only a few mouseclicks

This will add the driver for the external interrupt. Look at the Properties for the new driver and make some changes: First, change the *Channel* from *0* to *11*, as SW4 is connected to *IRQ11*. For the same reason, change the name to *g_external_irq11*, or anything else you like.

The interrupt is already enabled and assigned priority 12. Actually, it could be any other priority between 0 and 14 as well, but 12 is a good start, as you will rarely run into interrupt priority collisions, even in larger systems. Please note that priority 15 is reserved for System Tick Timer *(systick)* and hence cannot be used by other peripherals.

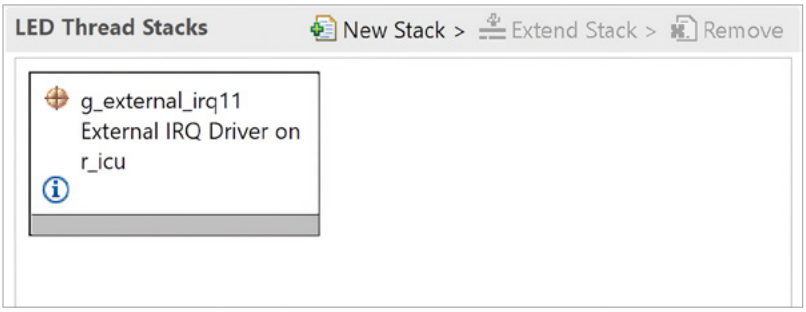

Figure 9-3: The grey color bar of the stack element indicates that this driver is a module instance and that it may be referenced by one other Synergy module instance only

Change the *Trigger* from *Rising* to *Falling*, to catch the pressing of the button, the Digital *Filtering* from *Disabled* to *Enabled* and the *Digital Filtering Sample Clock* from *PCLK / 64* to *PCLK / 32*. This will help to debounce the button. And finally, change the *Callback* from *NULL* to *external_irq11_callback*. This step will create a function, which is called once SW4 is pressed. We will add the code for the callback function itself later on. Figure 9-4 shows a summary of the necessary settings.

Property	Value
Common	
Parameter Checking	Default (BSP)
Module g_external_irq11 External IRQ Driver on r_icu	
Name	g_external_irq11
Channel	11
Trigger	Falling
Digital Filtering	Enabled
Digital Filtering Sample Clock (Only valid when Digital Filt	PCLK / 32
Interrupt enabled after initialization	True
Callback	external_irq11_callback
Pin Interrupt Priority	Priority 12

Figure 9-4: The properties of the IRQ driver needed for our application

Now, there are only a couple of additional steps left you need to perform until you can compile and download your program. The next one is to add a semaphore.

For that, click on the *New Object* button in the *LED Thread Objects* pane. If you do not see this pane, but a *HAL/Common Objects* pane, highlight the *LED Thread* in the *Threads* pane and it will become visible. Add a *Semaphore*, as we will need one to notify the *LED Thread* once the button is pushed. Change the semaphore's *Name* property to *SW4 Semaphore* and the *Symbol* property to *g_sw4_semaphore.*

Leave the count at zero, as we will increment it each time the button SW4 is pressed. Now your *Threads* tab in the Synergy Configurator should look similar to Figure 9-5.

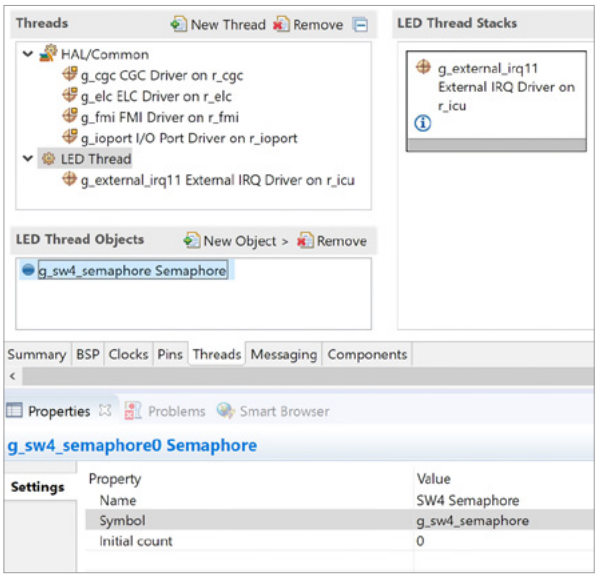

Figure 9-5: This is what the Threads tab should look like after the LED Thread and the semaphore has been added

The final step in the Synergy Configurator is to configure the I/O-pin to which SW4 is connected to as IRQ11 input. For that, activate the *Pins* tab inside the configurator and expand *Ports → P0* and highlight *P006*. On the S5D9 Promotion Kit, this is the pin SW4 is connected to. In the *Pin Configuration* pane at the right give it the *Symbolic Name* SW4, change the mode to *Input mode* and the IRQ to *IRQ11*_DS (if not already set) and the *Chip input/output* to GPIO. Note that the package viewer at the right will highlight pin 163 / P006. The complete configuration is shown in Figure 9-6.

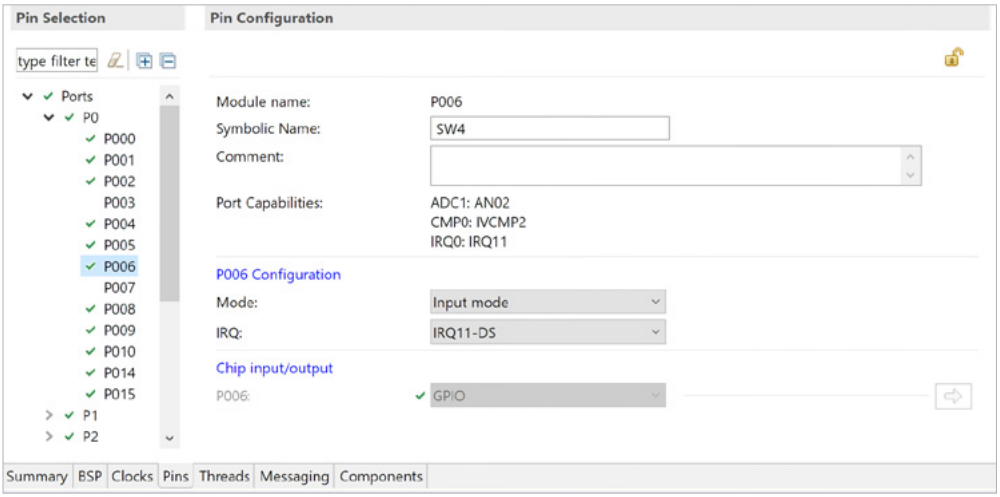

Figure 9-6: Port P006 needs to be configured as input to IRQ11

With this done, you have finished the settings in the configurator. Save the changes and click on the *Generate Project Content* icon at the top of it to generate the necessary files, folders and settings.

The final task you need to execute is to add code for populating the `Leds` structure similar to what you did already in the exercise of chapter 8, write a couple of lines for toggling the LED and to read the semaphore and to create the callback function. The full code for this can be reviewed below.

As we are using a thread for the handling of the push-button and toggling of the LED, we need to add the related code to the *led_thread_entry.c* file this time. Double-click on the file in the *Project Explorer* to open it in the editor. If it is not visible, expand the project folder and then the *src* directory. As with the exercise in chapter 8, you will need to add the structure for the LEDs and initialize it. Another variable for the level of the I/O-pin the LED1 is connected to needs to be defined. Name it `led_level`. It should be of type `ioport_level_t` and should be initialized to `IOPORT_LEVEL_HIGH` (a "high" level corresponds to "off" on the PK-S5D9).

The next step will be to open and to configure the external IRQ pin connected to SW4 on the board. For this, use the *open* function of the IRQ HAL driver. With that, the initialization steps necessary are finished.

Inside the `while(1)` loop, you need to add a couple of statements and to remove the `tx_thread_sleep(1)` statement. Start with a function call to write the value of the `led_level` to the output register of the I/O-pin for LED1 followed by statements toggling the pin level. There are several ways to do that. Implement your own, or look up the code at the end of the chapter. Do not forget the Smart Manual feature of e^2 studio, it helps a lot!

The final statement inside the `while(1)` is a call to `tx_semaphore_get()` with the address of the semaphore and the constant TX_WAIT_FOREVER as parameters. This will advise the RTOS to suspend the thread indefinitely until the semaphore is posted from the callback function inside the IRQ 11 interrupt service routine.

And, the last thing to do is to add the callback routine called by the IRQ 11 ISR. This code should be as short as possible, as it will be executed in the context of the interrupt service routine. Writing this function is easy: Simply go to *Developer Assistance → LED Thread → g_external_irq11 External IRQ Driver on r_icu* in the Project Explorer and drag and drop the *Callback function definition* at the end of the list appearing into your source file.

```
void external_irq11_callback(external_irq_callback_args_t *p_args);
```

Inside the callback function, add the following two lines of code:

```
SSP_PARAMETER_NOT_USED(p_args);
tx_semaphore_put(&g_sw4_semaphore);
```

The macro in the first line will tell the compiler that the callback function does not use the parameter `p_args`, avoiding a warning from the compiler, while the second one sets the semaphore each time button SW4 is pressed.

Once all the coding is complete, build your project by clicking on the *Build*-icon (the "hammer"). If it does not compile with zero errors and zero warnings, go back to your code and fix the problems with the help from the feedback of the compiler looking them up in the *Problems* view.

If the project built successfully, click on the small arrow beside the *Debug* icon, select *Debug Configurations* and expand *Renesas GDB Hardware Debugging*. Select *MyRtosProject* or the name you gave your version and click on *Debug*. This will start the debugger. If you need more information on that, please review the related section in chapter 8. Once the debugger is up and running, click on *Resume* twice. Your program is now running and each time you press SW4 on the PK, the green LED1 should toggle.

CONGRATULATIONS!

You successfully finished this exercise.

9.3 Adding a Thread to ThreadX® using IAR Embedded Workbench® for Renesas Synergy™

This exercise is again based on the PK S5D9 Promotion Kit (PK). This time, we will use the push-button SW4 on the top right corner of the board to signal an event to the application, which will toggle the green LED1 as response to it. For the implementation, we will use ThreadX® and the handling of the event will take place inside a thread and the notification of this thread will be done through a semaphore.

As usual, the first step is to create a new project, something you already exercised in chapters 4 and 8. To get started go to *Renesas Synergy → New Synergy Project i*n the IAR Embedded Workbench® for Renesas Synergy™. If there is no active workspace loaded, the *Save Workspace As* dialog will show. Navigate to a directory of your choice and give the workspace a name,for example *MyRtosWorkspace*. Click on *OK*. Another dialog window will show, asking for the locations of the Synergy Standalone Configurator (SSC) installation and the license file for the SSP to use. If you did the previous exercises, all necessary settings should have been already made, otherwise you need to fill in the required information. First, you need to enter the directory, into which the Renesas Synergy™ Standalone Configurator (SSC) has been installed. If you used the Platform Installer, this was *C:\Renesas\Synergy\ssc_v7.5.1_ssp_v1.7.0* by default, or if you decided to go for the Standalone Installer it was *C:\Renesas\Synergy\SSC_v7_5_1_V20190813*. If you used your own paths, enter this one. If you are unsure, refer to chapter 4 for more details.

Next you need to provide the correct path to the license file for the Synergy Software Package (SSP). The navigation button will guide you to the directory where the evaluation license was placed during installation. You can find it in the *\internal\projectgen\Licenses* subdirectory of one of the installation directories given above. Select the file named *SSP_License_Example_EvalLicense_20190725.xml* (or similar). Click on *OK* when finished.

A second *Save As* dialog will show, where you need to provide a location and a name for your project. Navigate to a directory of your choice or create a new one, which you might want to name *MyRtosProject*. It is important that you use a directory where no other Synergy project is located, as the SSC would use the settings of the older project instead the ones of the current one. A new window will tell you that the EW for Synergy will wait for the Synergy Standalone Configurator to finish and the SSC starts. This might take a couple of seconds.

In the Synergy Standalone Configurator look out for the field called *SSP version* under *Device Selection*: It should show the same version of the Synergy Software Package you installed in chapter 4. Under Board select *S5D9 PK*, as this is the hardware we will use for this exercise. Verify that R7FS5D97E3A01CFC is shown beside device; it should have been automatically selected when you changed the board type. If not, navigate through the drop-down list until you found it. Click on *Next*.

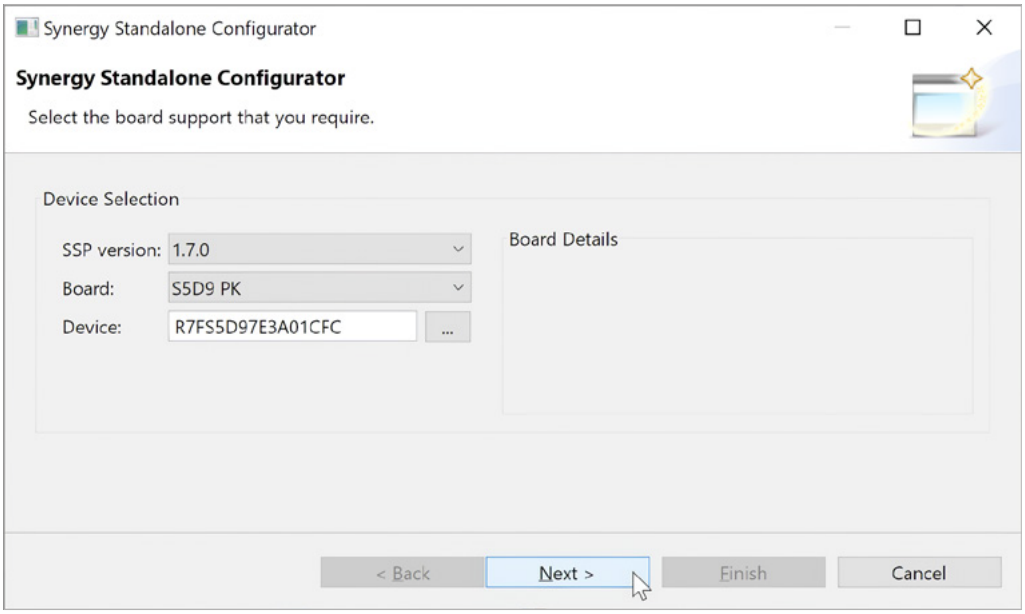

Figure 9-7: Make sure to select the S5D9 PK, as this is the board used for this exercise

In the *Project Template Selection* frame showing, select *BSP*. Click on *Finish* and after the project has been generated by the configurator, the SSC will open the *Synergy Configuration* perspective.

Once this perspective shows, go directly to the *Threads* tab. This tab will show one entry for *HAL/Common* in the *Threads* pane, containing the drivers for the Event Link Controller (ELC), the Clock Generation Controller (CGC) and the Factory MCU Information (FMI). The FMI driver includes a generic API for reading the records from the Factory MCU Information Flash Table, providing information about the features and peripherals implemented on the microcontroller used. Click on the *New Thread* icon at the top of the pane, which will add a new thread.

Now change the properties of the new thread in the *Properties* view: Rename the symbol to *led_thread* and the name to *LED Thread*. Press the *Return key*. Leave the other properties at their default value. On the *LED Thread Stack* pane on the right, click on the *New Stack* button and select *Driver → Input → External IRQ Driver on r_icu* (see Figure 9-2).

This will add the driver for the external interrupt. Look at the Properties for the new driver and make some changes: First, change the *Channel* from *0* to *11*, as SW4 is connected to IRQ11. For the same reason, change the name to *g_external_irq11*, or anything else you like.

The interrupt is already enabled and assigned priority 12. Actually, it could be any other priority between 0 and 14 as well, but 12 is a good start, as you will rarely run into interrupt priority collisions, even in larger systems. Please note that priority 15 is reserved for System Tick Timer (systick) and hence cannot be used by other peripherals.

Change the *Trigger* from *Rising* to *Falling*, to catch the pressing of the button, the *Digital Filtering* from *Disabled* to *Enabled* and the *Digital Filtering Sample Clock* from *PCLK / 64* to *PCLK / 32*. This will help to debounce the button. And finally, change the *Callback* from *NULL* to *external_irq11_callback*. This step will create a function, which is called once SW4 is pressed. We will add the code for the callback function itself later on. Refer to Figure 9-4 for a summary of the necessary settings. Now, there are only a couple of additional steps left you need to perform until you can compile and download your program. The next one is to add a semaphore.

For that, click on the *New Object* button in the *LED Thread Objects* pane. If you do not see this pane, but a *HAL/Common Objects* pane, highlight the *LED Thread* in the *Threads* pane and it will become visible. Add a *Semaphore*, as we will need one to notify the *LED Thread* once the button is pushed. Change the semaphore's *Name* property to *SW4 Semaphore* and the Symbol property to *g_sw4_semaphore*.

Leave the count at zero, as we will increment it each time the button SW4 is pressed. Now your *Threads* tab in the Synergy Standalone Configurator should look similar to Figure 9-5.

The final step in the Synergy Configurator is to configure the I/O-pin to which SW4 is connected to as IRQ11 input. For that, activate the *Pins* tab inside the configurator and expand *Ports → P0* and highlight *P006*. On the S5D9 Promotion Kit, this is the pin SW4 is connected to. In the *Pin Configuration* pane at the right give it the *Symbolic Name SW4*, change the *mode* to *Input mode* and the *IRQ* to *IRQ11_DS* (if not already set) and the *Chip input/output* to *GPIO*. Note that the package viewer at the right will highlight pin 163 / P006. The complete configuration is shown in Figure 9-6.

With this done, you have finished the settings in the configurator. Save the changes and click on the *Generate Project Content* icon at the top of it to generated the necessary files, folders and settings. Once this process has finished, close the SSC by clicking on the close button of the window (the 'X' at the top right side). If asked if you want to save the changes, answer with Yes. This will get you back to the IAR Embedded Workbench® for Renesas Synergy™ and will update the workspace of the IDE with the additional files and directories. You can open the SSC anytime again by clicking on the *Synergy Configuration* icon on the toolbar, by selecting *Renesas Synergy → Configurator* in the menu or by right clicking on the *Synergy entry* in the project tree and selecting *Open Renesas Synergy Configurator*.

The final task you need to execute is to add code for populating the `Leds` structure similar to what you did already in the exercise of chapter 8, write a couple of lines for toggling the LED and to read the semaphore and to create the callback function. The full code for this can be reviewed on page 12.

As we are using a thread for the handling of the push-button and toggling of the LED, we need to add the related code to the *led_thread_entry.c* file this time. Double-click on the file in the *Project Explorer* to open it in the editor. If it is not visible, expand the project folder and then the *src* directory. As with the exercise in chapter 8, you will need to add the structure for the LEDs and initialize it. Another variable for the level of the I/O-pin the LED1 is connected to needs to be defined. Name it `led_level`. It should be of type `ioport_level_t` and should be initialized to `IOPORT_LEVEL_HIGH` (a "high" level corresponds to "off" on the PK-S5D9).

The next step will be to open and to configure the external IRQ pin connected to SW4 on the board. For this, use the *open* function of the IRQ HAL driver. With that, the initialization steps necessary are finished.

Inside the `while(1)` loop, you need to add a couple of statements and to remove the `tx_thread_sleep(1)` statement. Start with a function call to write the value of the `led_level` to the output register of the I/O-pin for LED1 followed by statements toggling the pin level. There are several ways to do that. Implement your own, or look up the code at the end of the chapter. Do not forget the auto-completion feature of the IDE; it helps a lot!

The final statement inside the `while(1)` is a call to `tx_semaphore_get()` with the address of the semaphore and the constant `TX_WAIT_FOREVER` as parameter. This will advise the RTOS to suspend the thread indefinitely until the semaphore is posted from the callback function inside the IRQ 11 interrupt service routine.

And, the last thing to do is to add the callback routine called by the IRQ 11 ISR. This code should be as short as possible, as it will be executed in the context of the interrupt service routine. You can find the prototype for this function in the *HAL/Thread* code generated in the *synergy_gen* folder. In our case, it can be copied over from the *led_thread.h* file residing there:

```
void external_irq11_callback(external_irq_callback_args_t *p_args);
```

Inside the callback function, add the following two lines of code:

```
SSP_PARAMETER_NOT_USED(p_args);
tx_semaphore_put(&g_sw4_semaphore);
```

The macro in the first line will tell the compiler that the callback function does not use the parameter `p_args,` avoiding a warning from the compiler, while the second one sets the semaphore each time button SW4 is pressed.

Once all the coding is complete, build your project by clicking on the *Make*-icon on the toolbar or by hitting the *F7*-key. If it does not compile with zero errors, go back to your code and fix the problems with the help from the feedback of the compiler looking them up in the *Build* window.

If the project built successfully, the next step is to make sure that your Promotion Kit is connected to your Windows® workstation. If not do so by inserting an USB-cable into the port named J19 on the PK and the other end into a free USB-port of your workstation. Now start the debugging session by selecting either *Project → Download and Debug* or by clicking on the *Download and Debug* icon on the toolbar. The IAR Embedded Workbench® for Renesas Synergy™ will download your code, switch to the debugger and stop at the entry point of the program, the call to main(). If you get warnings about missing source files, just click on *Skip*. This is because some of the source files are protected from viewing when using an evaluation license.

Now run the program. For that, click on the *Go* icon on the debugger toolbar and the program continues to execute. Your program is now running and each time you press SW4 on the Promotion Kit, the green LED1 should toggle.

CONGRATULATIONS!

You successfully finished this exercise.

```
#include “led_thread.h”
void led_thread_entry(void);
bsp_leds_t Leds;
/* LED Thread entry function */
void led_thread_entry(void)
{
    ioport_level_t led_level = IOPORT_LEVEL_HIGH;
    R_BSP_LedsGet(&Leds);
    g_external_irq11.p_api->open(g_external_irq11.p_ctrl,
                                 g_external_irq11.p_cfg);
    while (1)
    {
        g_ioport.p_api->pinWrite(Leds.p_leds[BSP_LED_LED1],
                                 led_level);
        if (led_level == IOPORT_LEVEL_HIGH)
        {
            led_level = IOPORT_LEVEL_LOW;
        }
        else
        {
            led_level = IOPORT_LEVEL_HIGH;
        }
        tx_semaphore_get(&g_sw4_semaphore, TX_WAIT_FOREVER);
    }
}
void external_irq11_callback(external_irq_callback_args_t * p_args)
{
    SSP_PARAMETER_NOT_USED(p_args);
    tx_semaphore_put(&g_sw4_semaphore);
}
```

Points to take away from this chapter:

- Using HAL-driver is simple through the use of comprehensive APIs.
- The Synergy Software Package will take care of most non-user code related things.
- Using ThreadX® is straightforward and adding threads and semaphores is not much of an effort.