AN1260: Integrating v3.x Silicon Labs *Bluetooth*® Applications with Real-Time Operating Systems

This application note provides background information on the system architecture and event-based communication between a real-time operating system (RTOS) and the Bluetooth application. It then discusses user-defined tasks and describes how to customize an application.

KEY POINTS

- Prerequisites
- System architecture, Inter-task communication and task descriptions
- Application integration using specific example tasks
- Customization the application
- Additional resources

1. Introduction

This application note describes how to integrate a v3.x Silicon Labs Bluetooth application with an RTOS (real-time operating system), using the **SOC-Empty** example in Simplicity Studio 5 as an illustration. As of Silicon Labs Bluetooth SDK version 3.1.0, the adaptation layer has been designed to work with Micrium OS RTOS and FreeRTOS, both using the CMSIS-RTOS2 API. To work with any other RTOS, the OS should have the following features:

- Tasks with priorities
- Flags for triggering task execution from interrupt context
- Mutexes

The solution places the handling of Bluetooth stack events into its own task, allowing the application to run other tasks when no Bluetooth events are pending. When no tasks are ready to run, the application will sleep.

1.1 Prerequisites

You should have:

- A general understanding of RTOS concepts such as tasks, semaphores and mutexes.
- A working knowledge of Bluetooth Low Energy communications.
- A Wireless starter kit with an EFR32BG or EFR32MG radio board
- Installed and be familiar with using the following:
  - Simplicity Studio 5
  - IAR Embedded Workbench for ARM (IAR-EWARM) (optional - only use the version that is compatible with the SDK version, as listed in the SDK release notes). May be used as a compiler in the Simplicity Studio development environment as an alternative to GCC (The GNU Compiler Collection), which is provided with Simplicity Studio. Again, use only the GCC version that is compatible with the SDK version, as listed in the SDK release notes.
  - Bluetooth SDK v3.1.0 or above

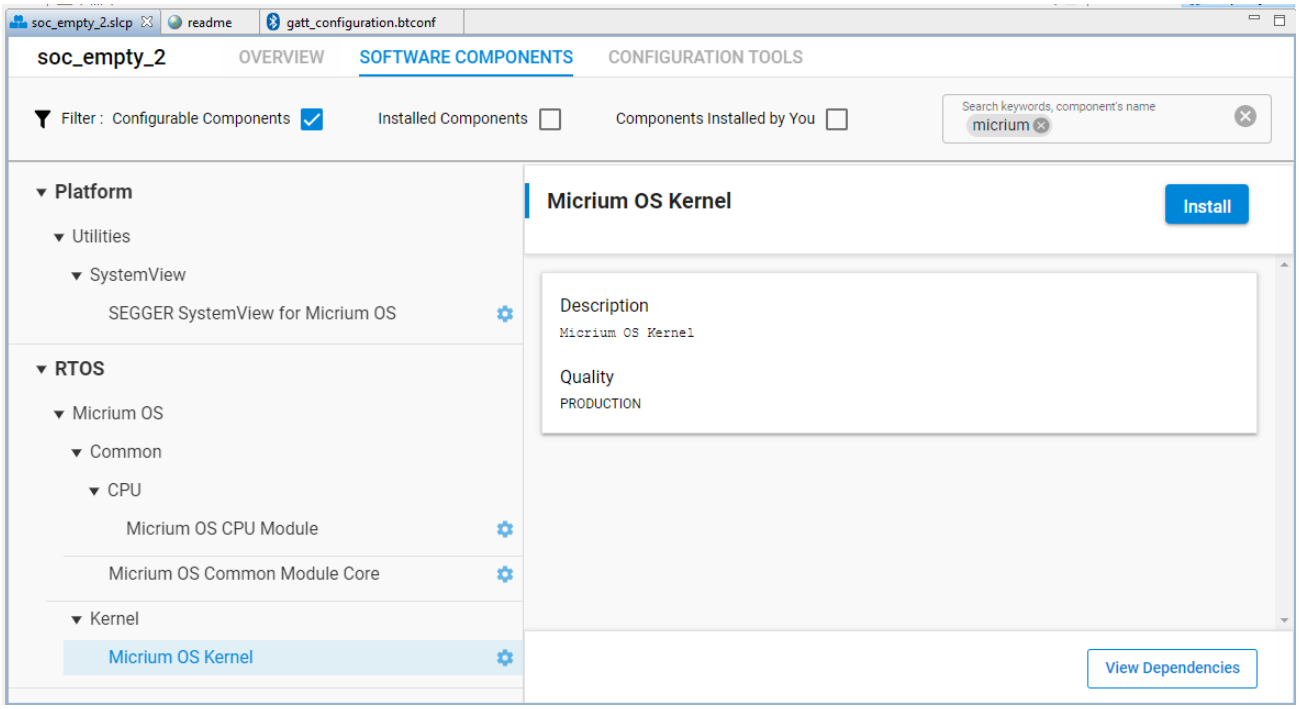
If you need to familiarize yourself with any of these concepts, the following may be useful:

- *UG103.14: Bluetooth® LE Fundamentals*
- *QSG169: Bluetooth® SDK v3.x Quick Start Guide*
- µC/OS-III User Manual for an overview of RTOS fundamentals
- FreeRTOS
  - RTOS Fundamentals
  - FreeRTOS Kernel Developer Docs

1.2 Micrium OS Configuration

To have Micrium RTOS run in your application, the **Micrium OS Kernel** component needs to be added to your application project. The following procedure illustrates this using the **SOC-Empty** example project.

1. Double-click the `soc_empty.slcp` file in the Simplicity Studio Project Explorer to open the Project Configurator, and click the **SOFTWARE COMPONENTS** tab.
2. Enter **micrium** in the search field in the top right corner. Optionally, select the **Configurable Components** checkbox to shorten the component list.
3. Select **Micrium OS Kernel** in the left pane and click **Install** .



The **Micrium OS Kernel** component adds Micrium RTOS support to the **SOC-Empty** example project and configures the project to run the Bluetooth stack in multiple tasks. These tasks are discussed in section 2. System Architecture.

The following components are added automatically when adding the **Micrium OS Kernel** component:

Generic RTOS related:

- Platform - CMSIS - CMSIS-RTOS2
- Platform - CMSIS - CMSIS-RTOS2 Headers
- Platform - Common - Common APIs for CMSIS-Compliant Kernels

Micrium OS related:

- RTOS - Micrium OS - Common - CPU - Micrium OS CPU Module
- RTOS - Micrium OS - Common - Micrium OS Common Module Core
- RTOS - Micrium OS - Common - RTOS Description
- RTOS - Micrium OS - Kernel - Micrium OS Kernel

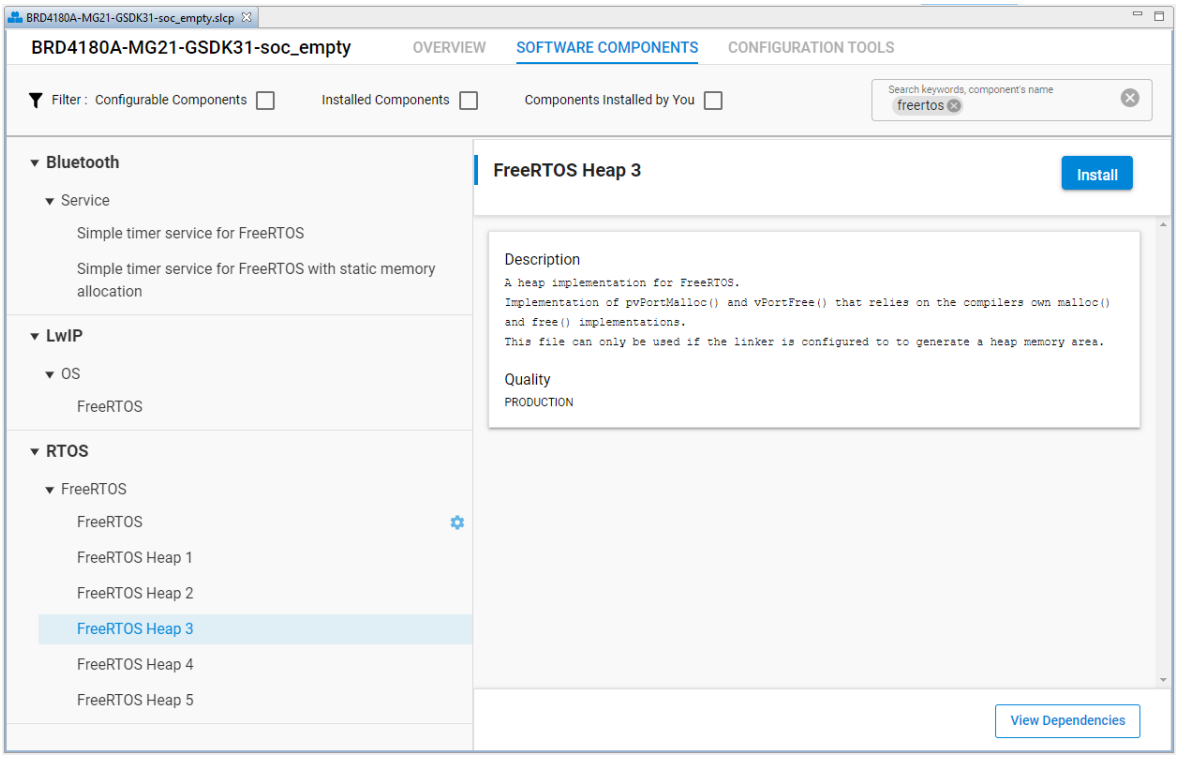
Note: If the component **Simple timer service** is used in the application, a separate version for the Micrium OS called **Simple timer service for Micrium RTOS** should be used instead.

1.3 FreeRTOS Configuration

To have FreeRTOS run in your application, add the **FreeRTOS** component and the preferred **FreeRTOS Heap** component (Heap 3 is used in the examples) to your application project.

To add the **FreeRTOS** component to the **SOC-Empty** example project:

1. Double-click the `soc_empty.slcp` file in the Simplicity Studio Project Explorer to open the Project Configurator, and click the **SOFTWARE COMPONENTS** tab.
2. Enter **freertos** in the search field in the top right corner.
3. Select **FreeRTOS Heap 3** in the left pane and click **Install**.



The **FreeRTOS Heap 3** component adds FreeRTOS support to the **SOC-Empty** example project and configures the project to run the Bluetooth stack in multiple tasks. These tasks are discussed in section 2. System Architecture.

Note: If the **FreeRTOS** component is added instead of the **FreeRTOS Heap 3** component, it will default to FreeRTOS Heap 4. To see more information about the FreeRTOS heap implementations, see the FreeRTOS documentation.

The following components are added automatically when adding the **FreeRTOS Heap 3** component:

Generic RTOS related:

- Platform - CMSIS - CMSIS-RTOS2
- Platform - CMSIS - CMSIS-RTOS2 Headers
- Platform - Common - Common APIs for CMSIS-Compliant Kernels

FreeRTOS related:

- RTOS - FreeRTOS - FreeRTOS
- RTOS - FreeRTOS - FreeRTOS Heap 3

Note: If the component **Simple timer service** is used in the application, there are special versions for the FreeRTOS called **Simple timer service for FreeRTOS** and **Simple timer service for FreeRTOS with static memory allocation**.

2. System Architecture

The **SOC-Empty** example application with Micrium RTOS or FreeRTOS support requires several tasks in order to operate

- Link Layer Task
- Bluetooth Host Task
- Event Handler Task
- Idle Task

Silicon Labs has implemented these tasks for the Micrium RTOS and FreeRTOS.

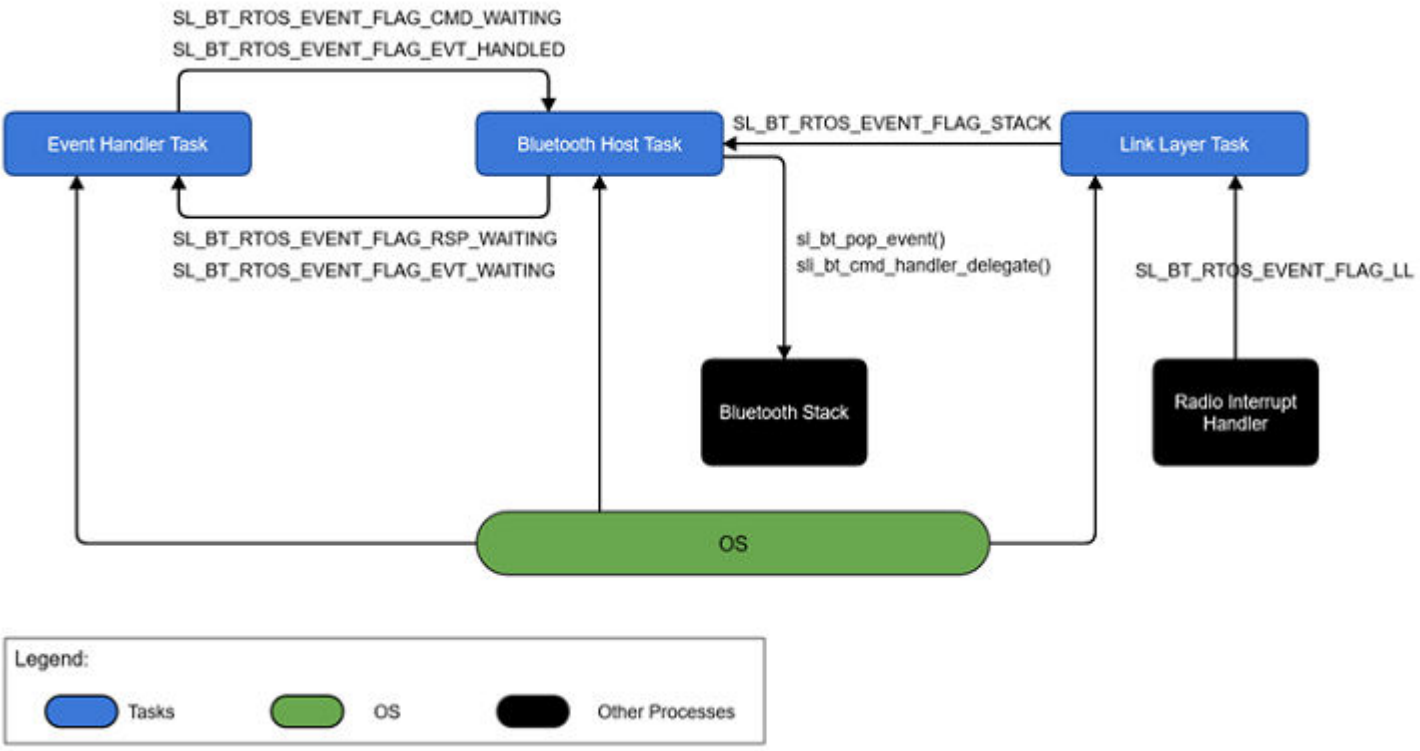
2.1 Inter-Task Communication

Before describing the tasks, it is important to understand how the tasks communicate with each other. The tasks in this application synchronize with each other through the use of a number of flags. The flags are internal of the sl_bt_rtos_adaptation layer. These flags are summarized in the following table:

FLAG	Sender	Receiver	Purpose
SL_BT_RTOS_EVENT_FLAG_STACK	Link Layer Task	Bluetooth Host Task	Bluetooth stack needs an update, call sl_bt_pop_event()
SL_BT_RTOS_EVENT_FLAG_LL	Radio Interrupt	Link Layer Task	Link Layer needs an update, call sl_bt_priority_handle()
SL_BT_RTOS_EVENT_FLAG_CMD_WAITING	Event Handler and Application Tasks	Bluetooth Host Task	Command is ready in shared memory, call sli_bt_cmd_handler_delegate()
SL_BT_RTOS_EVENT_FLAG_RSP_WAITING	Bluetooth Host Task	Event Handler and Application Tasks	Response is ready in shared memory.
SL_BT_RTOS_EVENT_FLAG_EVT_WAITING	Bluetooth Host Task	Event Handler Task	Event is ready in shared memory.
SL_BT_RTOS_EVENT_FLAG_EVT_HANDLED	Event Handler Task	Bluetooth Host Task	Event is handled and shared memory is free to use for next event.

The following diagram illustrates how these flags are used in synchronizing the tasks.

In addition to these flags, a mutex is used by the gecko command handler to make it thread-safe. This makes it possible to call BGAPI commands from multiple tasks.



2.2 Link Layer Task

The purpose of this task is to update the upper link layer. The link layer task waits for the `SL_BT_RTOS_EVENT_FLAG_LL` flag to be set before running. The upper link layer is updated by calling `sl_bt_priority_handle()`. The `SL_BT_RTOS_EVENT_FLAG_LL` flag is set by `sli_bt_rtos_ll_callback()`, which is a callback function specified to `scheduler_callback` in the stack configuration. The callback is called from a kernel-aware interrupt handler (lower link layer). This task is given the highest priority.

2.3 Bluetooth Host Task

The purpose of this task is to update the Bluetooth stack, issue events, and handle commands. This task waits for any of the `SL_BT_RTOS_EVENT_FLAG_STACK`, `SL_BT_RTOS_EVENT_FLAG_CMD_WAITING` and `SL_BT_RTOS_EVENT_FLAG_EVT_HANDLED` flags to be set before running. The `SL_BT_RTOS_EVENT_FLAG_STACK` flag is set by `sli_bt_rtos_stack_callback()`, which is a callback function specified to `stack_schedule_callback` in the stack configuration. This task has higher priority than the Event Handler Task and any of the Application Tasks, but lower than the Link Layer Task.

Before this task starts running, it prepares the application to run the Bluetooth stack. This task calls `sl_bt_init()` to initialize and configure the Bluetooth stack, and then calls `sl_bt_rtos_create_tasks()` to create the Link Layer Task and Event Handler Task.

2.3.1 Updating the Stack

The Bluetooth stack must be updated periodically. The Bluetooth Host Task updates the stack by calling `sl_bt_event_pending()` and reads the next stack event from the stack by calling `sl_bt_pop_event()`. This allows the stack to process messages from the link layer as well as its own internal messages for timed actions that it needs to perform.

2.3.2 Issuing Events

The Bluetooth Host Task sets the `SL_BT_RTOS_EVENT_FLAG_EVT_WAITING` flag to indicate to the event handler task that an event is ready to be retrieved. Only one event can be retrieved at a time. The `SL_BT_RTOS_EVENT_FLAG_EVT_WAITING` flag is cleared by the Event Handler Task when it has retrieved the event. The `SL_BT_RTOS_EVENT_FLAG_EVT_HANDLED` flag is set by the Event Handler Task to indicate that event handling is complete.

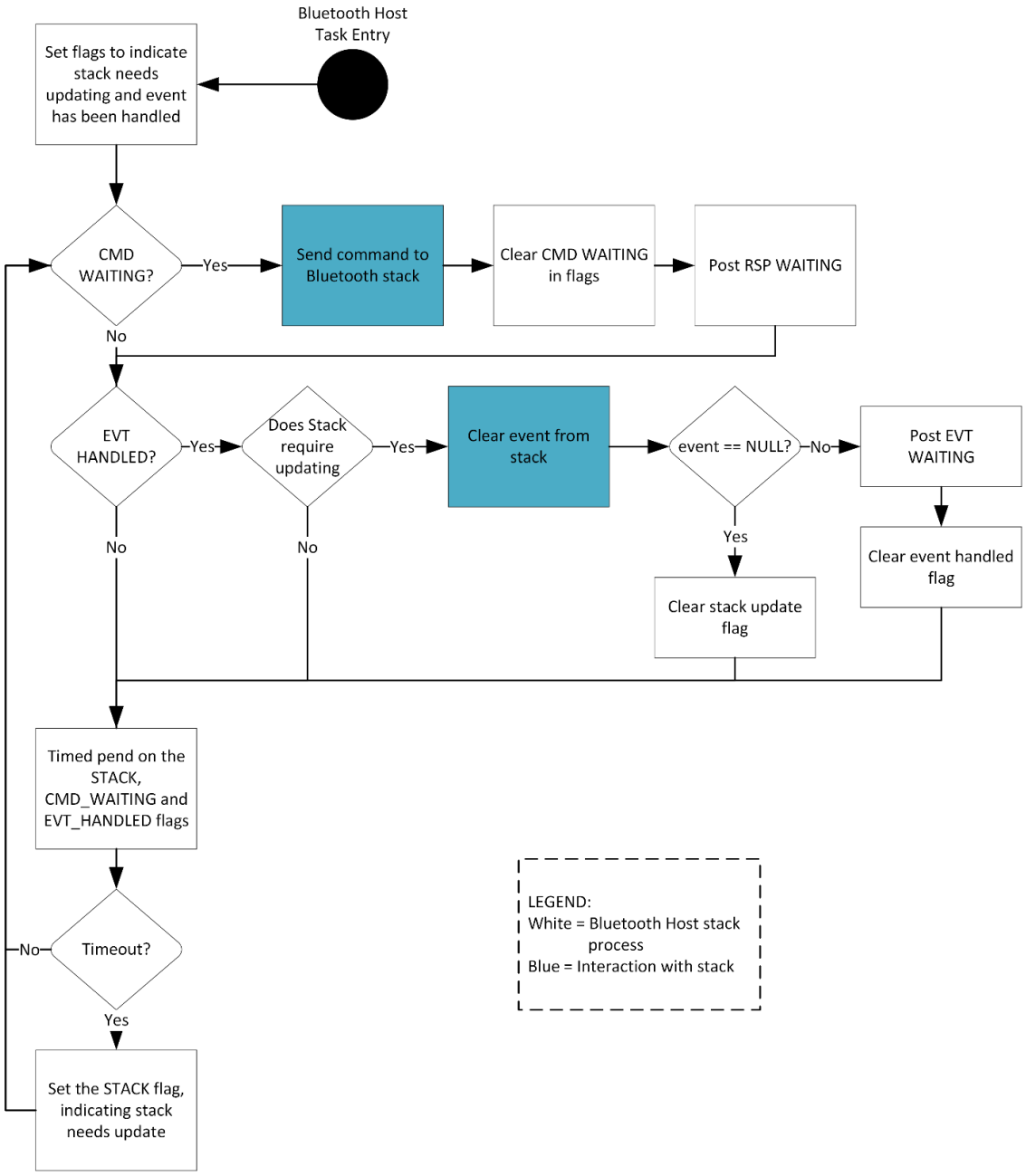
2.3.3 Command Handling

Commands can be sent to the stack from multiple tasks. Responses to these commands are forwarded to the calling task. Commands and responses are synchronized with the `SL_BT_RTOS_EVENT_FLAG_CMD_WAITING` and `SL_BT_RTOS_EVENT_FLAG_RSP_WAITING` flags and the BluetoothMutex mutex.

Commands are prepared and sent to the stack by a helper function called `sli_bt_cmd_handler_rtos_delegate()`. This function is called by any of the BGAPI functions and is made re-entrant through the use of a mutex. The function starts by pending on the mutex. When it gains control of the mutex the command is prepared and placed into shared memory, then the `SL_BT_RTOS_EVENT_FLAG_CMD_WAITING` flag is set to indicate to the stack that a command is waiting to be handled. This flag is cleared by the Bluetooth Host Task to indicate that the command has been sent to the stack and that it is now safe to send another command.

Then execution pends on the `SL_BT_RTOS_EVENT_FLAG_RSP_WAITING` flag, which is set by the Bluetooth Host Task when the command has been executed. This indicates that a response to the command is waiting. Finally, the mutex is released.

The following diagram illustrates how the Bluetooth Host Task operates.



1. On task startup, the `SL_BT_RTOS_EVENT_FLAG_STACK` is set to indicate that the stack needs updating and the `SL_BT_RTOS_EVENT_FLAG_EVT_HANDLED` flag is set to indicate that no event is currently being handled.

2. If the `SL_BT_RTOS_EVENT_FLAG_CMD_WAITING` flag is set, `sli_bt_cmd_handler_rtos_delegate()` is called to handle the command.
3. If the `SL_BT_RTOS_EVENT_FLAG_STACK` and the `SL_BT_RTOS_EVENT_FLAG_EVT_HANDLED` flags are set, `sl_bt_pop_event()` is called to get an event from the stack. If an event is found waiting, the `SL_BT_RTOS_EVENT_FLAG_EVT_WAITING` flag is set and the `SL_BT_RTOS_EVENT_FLAG_EVT_HANDLED` flag is cleared to indicate to the Event Handler Task that an event is ready to be handled and to the Bluetooth Host Task that an event is currently in the process of being handled. Otherwise, the `SL_BT_RTOS_EVENT_FLAG_STACK` flag is cleared to indicate that the stack does not require updating.
4. At this point, the task checks to see if the stack requires updating and whether any events are waiting to be handled. If no events are waiting to be handled and the stack does not need updating then it is safe to sleep and the Bluetooth Host Task does a pend on the `SL_BT_RTOS_EVENT_FLAG_STACK`, `SL_BT_RTOS_EVENT_FLAG_EVT_HANDLED` and `SL_BT_RTOS_EVENT_FLAG_CMD_WAITING` flags.
5. Steps 2 – 4 are repeated indefinitely.

2.4 Event Handler Task

The purpose of this task is to handle events sent by the Bluetooth stack. This task waits for the `SL_BT_RTOS_EVENT_FLAG_EVT_WAITING` flag to be set. This flag is set by the Bluetooth Host Task to indicate that there is an event waiting to be handled. Once this flag has been set, sl_bt_process_event() is called to handle the event. Finally, the `SL_BT_RTOS_EVENT_FLAG_EVT_HANDLED` flag is set to indicate to the Bluetooth Host Task that the event has been handled and the Event Handler Task is ready to handle another event. This task has a lower priority than the Bluetooth Host Task and Link Layer Task.

This task handles the `gatt_server_user_write_request` event for the user-type OTA control characteristic and boot the device to OTA DFU mode. This task then dispatches events to `sl_bt_on_event()`, which needs to be integrated in the application.

2.5 Idle Task

When no tasks are ready to run, the OS calls the Idle Task. The Idle Task puts the MCU into lowest available sleep mode, EM2, by default.

3. Application Integration

This section describes the Bluetooth event handler and application tasks, and how they are used to implement a sample Bluetooth device.

3.1 Bluetooth Event Handler

The Bluetooth Event Handler implemented in `sl_bt_on_event()`—part of the event handler task—takes a pointer to an event and handles it accordingly. A full list of events can be found in the Silicon Labs Bluetooth API Documentation (see section 4. Additional Resources). Some events triggered by the stack are mainly informative and do not require the application to do anything. Because this is a simple application, it handles only a small set of events as follows:

- system_boot: This event indicates that the Bluetooth stack is initialized and ready to receive commands. This is where you set the discoverability and connectability modes.
- connection_closed: This event is triggered when a connection is closed. Advertising is restarted in this event to allow future connections.

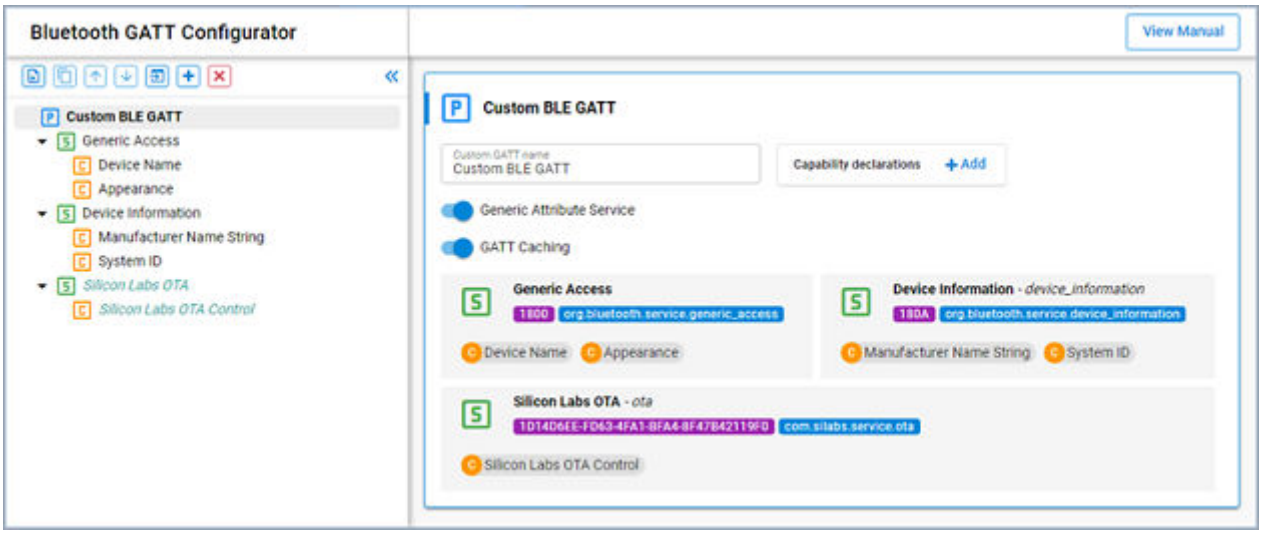
3.2 Customizing the Application

This section describes some common tasks such as customizing GATT attributes, adding event handlers, and adding support for other peripherals.

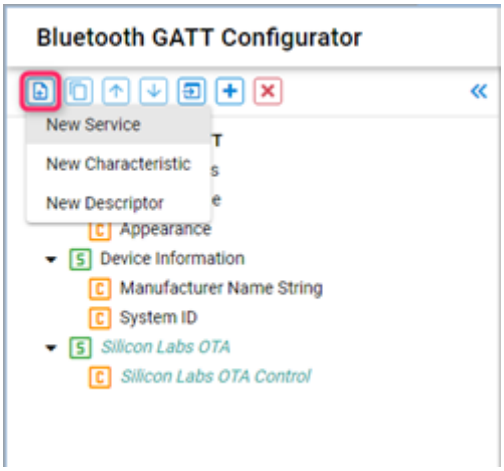
3.2.1 GATT Services and Characteristics

This section describes how to add a service and characteristic to control an LED on the Wireless Starter Kit. One of the tools provided with Simplicity Studio is the Bluetooth GATT Configurator. This tool provides a graphical interface for creating and editing the GATT database.

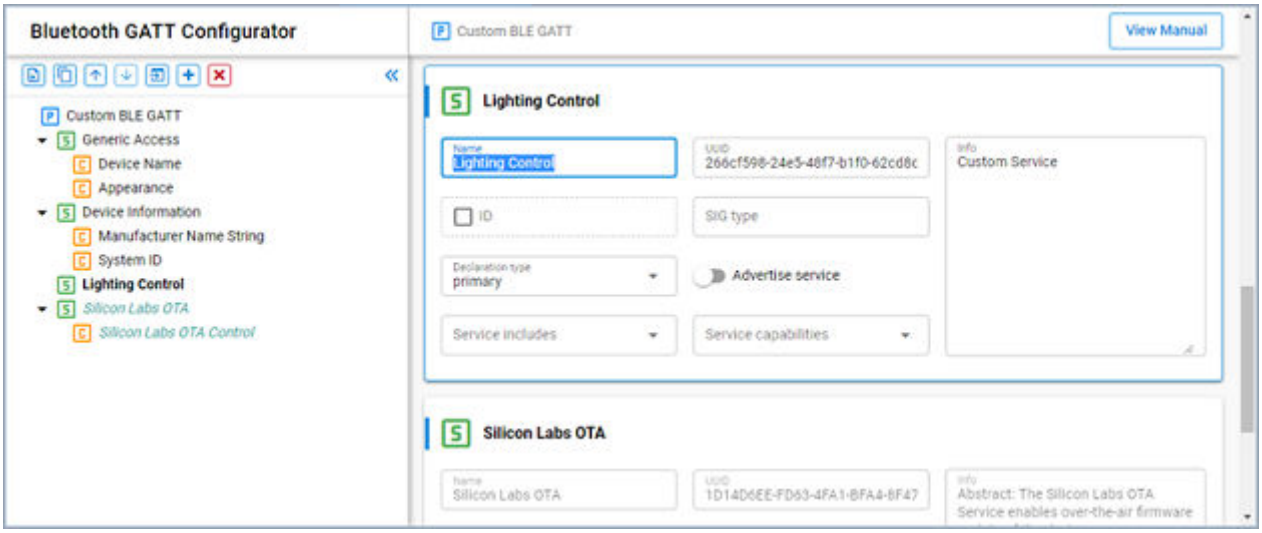
1. Open the Bluetooth GATT Configurator on the CONFIGURATION TOOLS tab in an open project or by double-clicking the file `config/btconf/gatt_configuration.btconf` in the Simplicity Studio Project Explorer.



2. Create a new service. Select **Custom BLE GATT** in the left pane. Click the Add New Item control in the top left corner and select **New Service**.

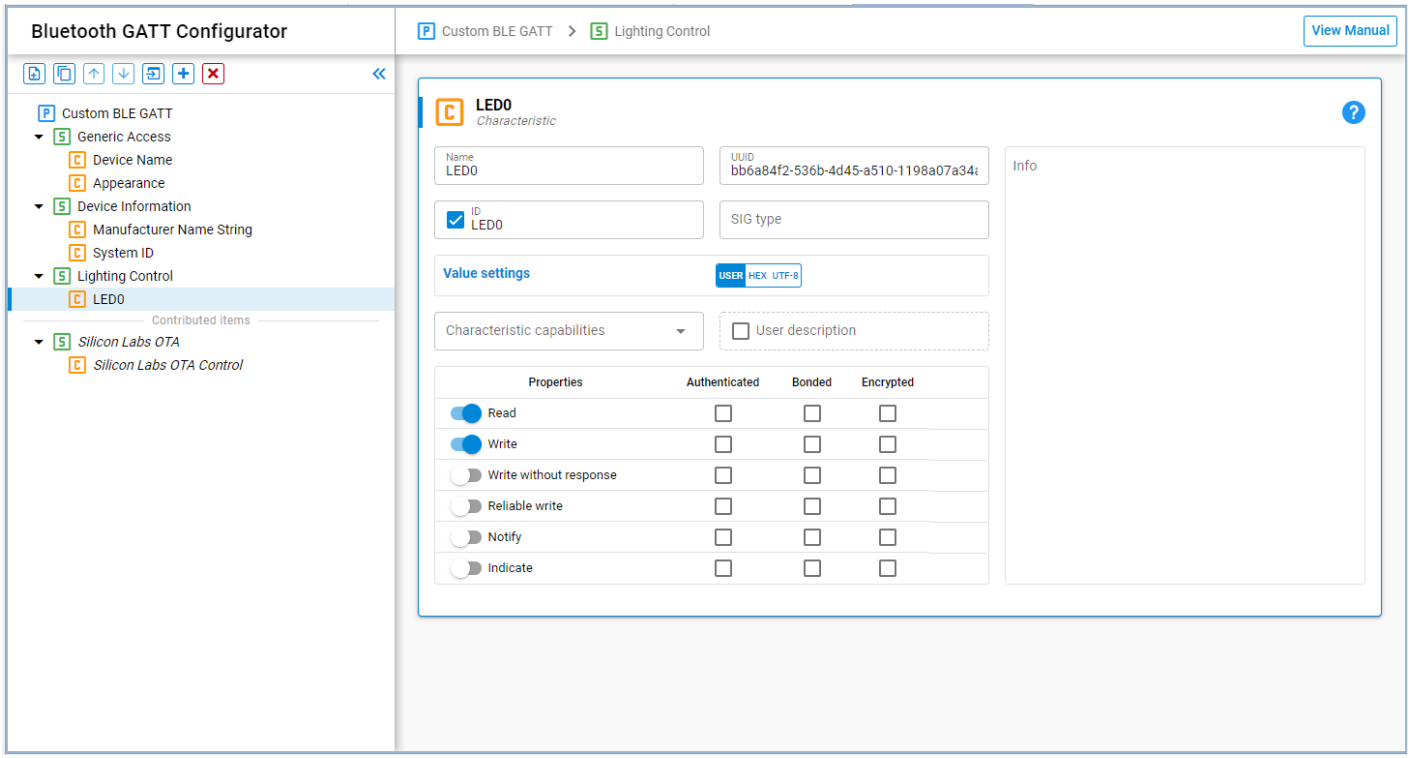


3. Select the created Custom Service and name the service **Lighting Control**, as shown in the following figure.



4. Add a characteristic.
   - Select the **Lighting Control** service. Click the Add New Item control in the top left corner and select **New Characteristic**.
   - Select the created Custom Characteristic and name the characteristic **LED0** at the right.
   - Check the **id** checkbox and enter **LED0** for the ID.
   - Set **Value Settings** to User.
   - Set the Properties **Read** and **Write** toggle switches.

   The characteristic should now look like the following figure.



5. Click **Save** and Simplicity Studio will automatically update the generated source code.

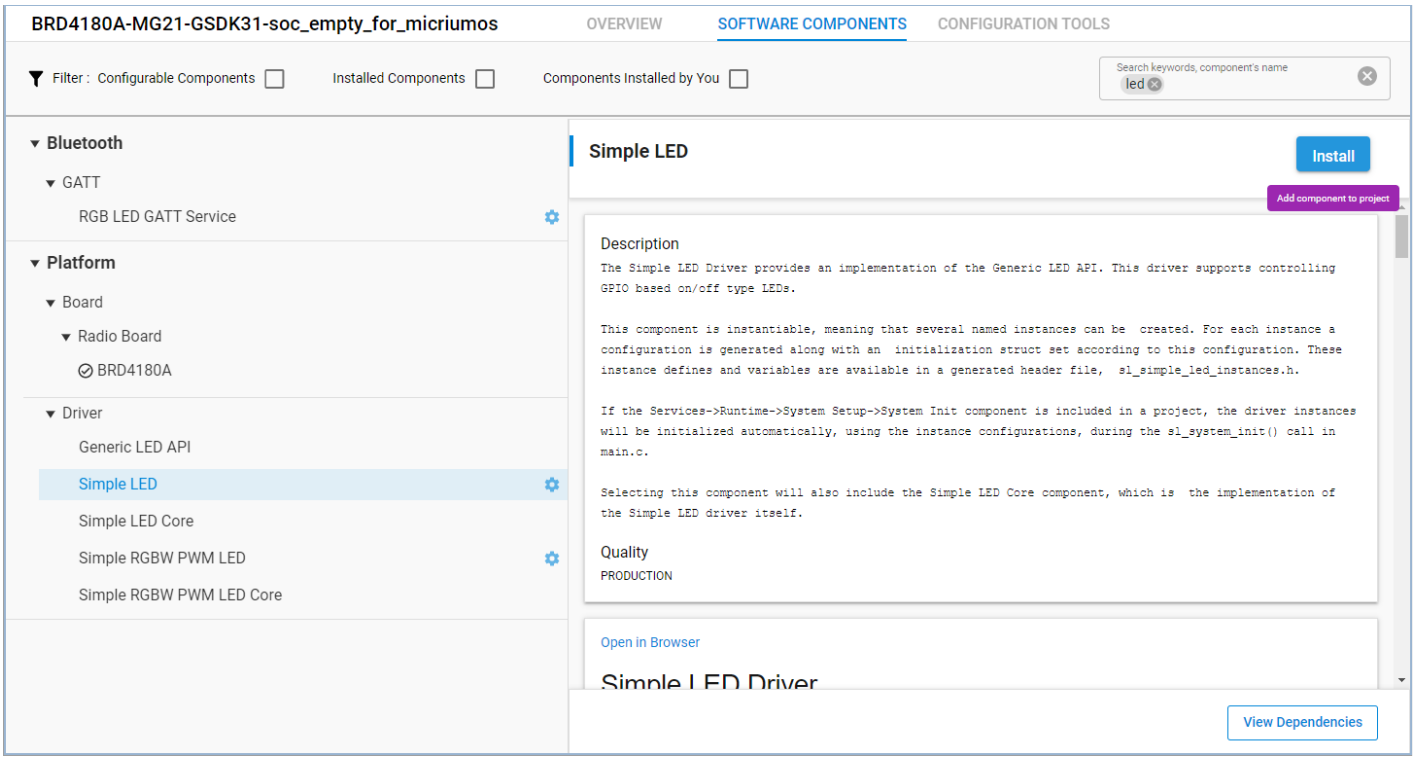
3.2.2 Event Handlers

This section discusses how to add event handlers for reading from and writing to the GATT characteristic added in section 3.2.1 GATT Services and Characteristics. The characteristic has write and read permissions. It is a user type so the application needs to handle the following events:

- gatt_server_user_write_request
- gatt_server_user_read_request

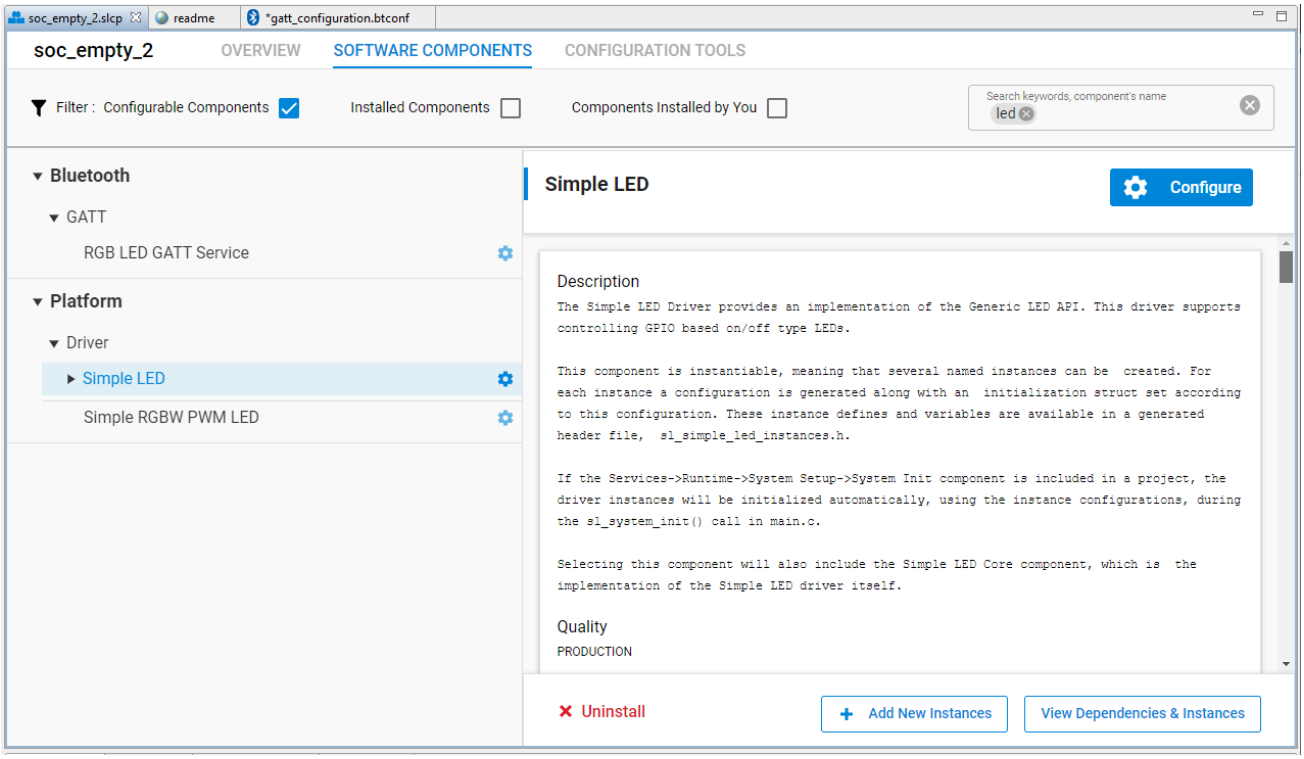
The Simple LED Driver component is required to access the LEDs on the Wireless Starter Kit. Simplicity Studio provides the Software Components Configuration tool to manage driver and API components.

1. Open the Project Configurator by double-clicking the `soc_empty.slcp` file in the Simplicity Studio Project Explorer. Click the **SOFTWARE COMPONENTS** tab on the top.
2. Enter **led** in the search field in the top right corner. Select **Simple LED** in the left pane and click **Install**, as shown in the following figure.



3. A pop-up Create A Component Instance window appears. Leave the default instance name **led0** as is and click **Done**.
4. Click **Add New Instances**. A pop-up Create A Component Instance window appears.

5. Leave the default instance name as **led1** and click **Done**. It should now look like the following figure.



After you have created the **led0** and **led1** instances of the Simple LED Driver component, the LED driver is automatically installed and turned on. The application can access directly the LED0 and LED1 by calling LED APIs.

Add the following code in `app.c` to include the header file that has declarations of functions to set and clear the LEDs.

```
#include "sl_simple_led_instances.h"
```

As described in section 3.1 Bluetooth Event Handler, Bluetooth events for the application should be handled in `sl_bt_on_event()`. Add the following code to implement the user write request handler.

```
case sl_bt_evt_gatt_server_user_write_request_id:
  if (evt->data.evt_gatt_server_user_write_request.characteristic == gattdb_LED0) {
    if (evt->data.evt_gatt_server_user_write_request.value.data[0]) {
      sl_led_turn_on(&sl_led_led0);
    }
    else {
      sl_led_turn_off(&sl_led_led0);
    }
    sl_bt_gatt_server_send_user_write_response(
      evt->data.evt_gatt_server_user_write_request.connection,
      evt->data.evt_gatt_server_user_write_request.characteristic,
      0);
  }
break;
```

This event handler verifies that the characteristic to be written is the LED0 characteristic, and then turns the LED0 either on or off depending on the data written. Finally, it sends a response to the remote GATT client to indicate that the write has been performed.

Add the following code to implement the user read request handler.

```
case sl_bt_evt_gatt_server_user_read_request_id:
  if(evt->data.evt_gatt_server_user_read_request.characteristic == gattdb_LED0) {
    led0_state = sl_led_get_state(&sl_led_led0);
    sl_bt_gatt_server_send_user_read_response(
      evt->data.evt_gatt_server_user_read_request.connection,
      evt->data.evt_gatt_server_user_read_request.characteristic,
      0,
      1, &led0_state, &sent_len);
  }
break;
```

This event handler sends the state of the LED0 to the client. The event handling requires the two variables `led0_state` and `sent_len` declared in `sl_bt_on_event()` as in the following code.

```
uint8_t led0_state = 0;
uint16_t sent_len = 0;
```

As your application requires, you can add similar handlers for other events in this way, as your application requires them. The event handlers implemented in `sl_bt_on_event()` should complete their work quickly. If your application needs to perform heavy or asynchronous work, you may need to creating create additional application tasks may be needed. This is discussed in section 3.3 Adding Application Tasks.

3.2.3 Adding Support for Other Peripherals

The easiest way to add support for other peripherals is through the use of Silicon Labs' emlib/emdrv peripheral libraries. These libraries contain APIs for initializing and controlling the EFR32 family's peripherals. A link to the documentation for these libraries is found in section 4. Additional Resources.

These libraries are packaged as software components and are included in the Silicon Labs Gecko Platform SDK. By using the Software Components Configuration tool, you can easily add the peripheral drivers and libraries to your projects. The tool also resolves dependencies and automatically installs the components that the peripheral component requires.

3.3 Adding Application Tasks

You may need to add application tasks in your Bluetooth application to simplify the application implementation. It is especially useful when your application needs to execute a procedure that requires significant computation, is not promptly responsive, or is independent of Bluetooth events. This section demonstrates LED blinking on the Wireless Starter Kit where you will learn how to create an Application Task in your application.

To create application tasks, Micrium RTOS or FreeRTOS support needs to be added to the **SOC-Empty** example project. Follow the steps in section 1.2 Micrium OS Configuration to install the **Micrium OS Kernel** component or in section 1.3 FreeRTOS Configuration to install the **FreeRTOS Heap 3** component.

Note: Even though the Bluetooth RTOS application example uses the CMSIS-RTOS2 API functions for the Bluetooth features, it is highly recommended that the user application code should not use CMSIS-RTOS2 API but instead use the native Micrium OS RTOS or FreeRTOS API to get the most out of the system. The CMSIS-RTOS2 API implementation does not support all the RTOS features, and is more suited to support easier Bluetooth stack porting to different RTOSes.

3.3.1 Adding an Application Task with the Micrium OS RTOS

Include the header file in `app.c`.

```
#include "os.h"
```

To create a Micrium RTOS task, you need to declare a Task Control Block (TCB), allocate a memory space as the stack of the task, write the task's code, and set the task name, priority, and other parameters required by the `OSTaskCreate()` API. The task's priority must be unique in the application and needs to be lower than the priorities of the Link Layer, Bluetooth Host Task, and Event Handler Task. The lower the number, the higher the priority. Therefore, Silicon Labs recommends that you specify priority numbers greater than or equal to **10** to your application tasks. Bluetooth stack task priorities can be found in the “sl_bt_rtos_config.h” and can be configured in the **Bluetooth Core**-component under Bluetooth – RTOS.

Add the following code at the global level in `app.c`.

```
//Application task
#define SL_BT_RTOS_APPLICATION_PRIORITY         10u

#ifndef APPLICATION_STACK_SIZE
#define APPLICATION_STACK_SIZE (1000 / sizeof(CPU_STK))
#endif
static  void  ApplicationTask (void  *p_arg);
static  OS_TCB   ApplicationTaskTCB;
static  CPU_STK  ApplicationTaskStk[APPLICATION_STACK_SIZE];
```

Add the following code in `app_init()` to create the task.

```
RTOS_ERR os_err;

//Application task
OSTaskCreate(&ApplicationTaskTCB,
             "Application Task",
             ApplicationTask,
             0u,
             SL_BT_RTOS_APPLICATION_PRIORITY,
             &ApplicationTaskStk[0u],
             APPLICATION_STACK_SIZE / 10u,
             APPLICATION_STACK_SIZE,
             0u,
             0u,
             0u,
             (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
             &os_err);
```

Then, write the task's code in `ApplicationTask()`. A task usually runs infinitely and needs to yield execution to other tasks when it has completed. An execution yield means freeing up CPU time for other tasks. This is usually achieved by calling a time or event API. The following code is the task implementation that turns the LED1 on or off every 1 second.

```
//Application task
static  void  ApplicationTask (void  *p_arg)
{
  RTOS_ERR  os_err;
  (void)p_arg;

  while (DEF_TRUE) {
    // Put your application code here!
    OSTimeDlyHMSM(0, 0, 1, 0,
                  OS_OPT_TIME_DLY | OS_OPT_TIME_HMSM_NON_STRICT,
                  &os_err);
    sl_led_toggle(&sl_led_led1);
  }
}
```

3.3.2 Implementing a Time- and Event-Driven Task with the Micrium OS RTOS

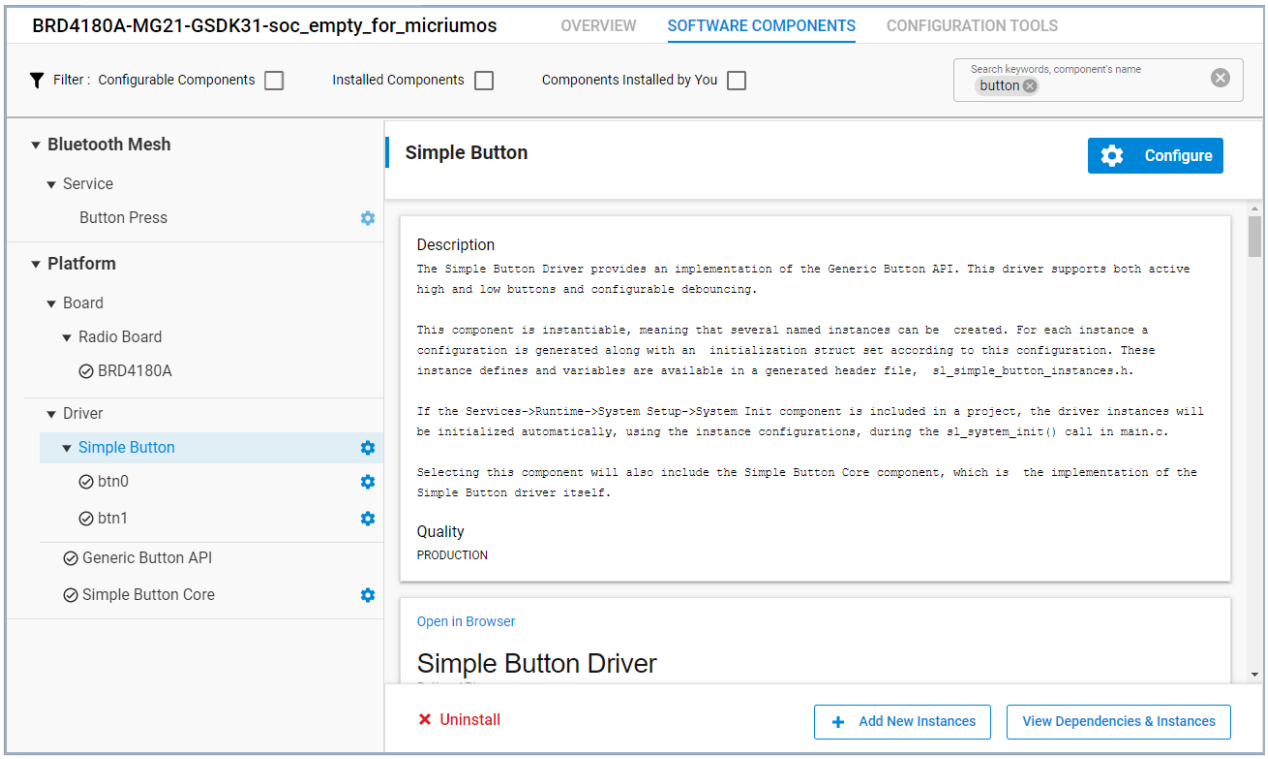
A task can be time-driven, event-driven, or both time- and event-driven. The model determines when the procedure is executed:

1. A timer expires.
2. An event occurs.
3. Either 1 or 2.

The Link Layer Task, Bluetooth Host Task, and Event Handler Task are event-driven and the LED toggling task demonstrated in section 3.3 Adding Application Tasks is time-driven.

The LED toggling task demonstrated in section 3.3 Adding Application Tasks is a good example to be changed to a time- and event-driven task. This task calls a time API to suspend itself for a period of specific time. A time- and event- driven task requires an event API that supports timeout so the task suspends its execution until either an event occurs or a timeout expires. This section demonstrates how an LED blinks at different frequencies based on whether or not a button on the Wireless Starter Kit is pushed.

The Simple Button component is required to access the buttons on the Wireless Starter Kit. Follow the steps in section 3.2.2 Event Handlers to install the component and create the **btn0** and **btn1** instances in the Software Components Configuration tool as shown in the following figure.



Add the following code in `app.c` to include the header file that has the declaration of the button state changed callback function.

```
#include "sl_simple_button_instances.h"
```

The demonstration uses Micrium Event Flag Management APIs for inter-task communication. The principle of the Event Flag Management is a task waits for a flag or some flags in an event flag group to be set and another task sets a flag in the same event flag group when a corresponding event occurs. A flag is a bit of a 32-bit variable. Therefore, flags are a bit pattern, which is a combination of bits of a 32-bit variable.

The easiest way for tasks using an event flag group is to declare it as a global variable. Add the following code at the global level in `app.c`.

```
OS_FLAG_GRP application_event_flags;

#define APPLICATION_EVENT_FLAG_BTN_ON    ((OS_FLAGS)1)
#define APPLICATION_EVENT_FLAG_BTN_OFF   ((OS_FLAGS)2)
```

Add the following code in `app_init()` to create the event flag group before creating the Application Task.

```
OSFlagCreate(&application_event_flags,
             "Application Flags",
```

```
            (OS_FLAGS)0,
            &os_err);
```

Next, modify the task's code in `ApplicationTask()` as follows. The task calls `OSFlagPend()` to suspend its execution and waits for any of the `APPLICATION_EVENT_FLAG_BTN_ON` and `APPLICATION_EVENT_FLAG_BTN_OFF` flags to be set. The task resumes to execute if any of the flags is set or when the timeout interval_ms expires. Therefore, the task toggles the LED1 at periodic intervals when no flag is set and changes the interval when a flag is set.

If you want to implement an event-driven task in this example, call `OSFlagPend()` with 0 as the third parameter and the task will wait forever until any of the flags is set.

```
//Application task
static  void  ApplicationTask (void  *p_arg)
{
  RTOS_ERR  os_err;
  (void)p_arg;
  uint32_t  interval_ms;
  OS_RATE_HZ  tick_rate;
  OS_FLAGS  flags;

  tick_rate = OSTimeTickRateHzGet(&os_err);
  interval_ms = 1 * tick_rate;

  while (DEF_TRUE) {
    // Put your application code here!
    flags = OSFlagPend(&application_event_flags,
                       (OS_FLAGS)APPLICATION_EVENT_FLAG_BTN_ON + APPLICATION_EVENT_FLAG_BTN_OFF,
                       interval_ms,
                       OS_OPT_PEND_BLOCKING + OS_OPT_PEND_FLAG_SET_ANY + OS_OPT_PEND_FLAG_CONSUME,
                       NULL,
                       &os_err);
    if (flags & APPLICATION_EVENT_FLAG_BTN_ON) {
      interval_ms = tick_rate / 5;  //200ms
      flags &= ~APPLICATION_EVENT_FLAG_BTN_ON;
    }
    if (flags & APPLICATION_EVENT_FLAG_BTN_OFF) {
      interval_ms = 1 * tick_rate;  //1s
      flags &= ~APPLICATION_EVENT_FLAG_BTN_OFF;
    }
    sl_led_toggle(&sl_led_led1);
  }
}
```

Finally, write the code in the callback function `sl_button_on_change()` to set the flags. The button driver calls this function when the state of the Button0 or Button1 has changed. The function sets the `APPLICATION_EVENT_FLAG_BTN_ON` flag when a button is pushed and sets the `APPLICATION_EVENT_FLAG_BTN_OFF` flag when a button is released. The code should like the following.

```
void sl_button_on_change(const sl_button_t *handle)
{
  RTOS_ERR os_err;

  if (sl_button_get_state(handle) == SL_SIMPLE_BUTTON_PRESSED) {
    OSFlagPost(&application_event_flags,(OS_FLAGS)APPLICATION_EVENT_FLAG_BTN_ON,OS_OPT_POST_FLAG_SET,&os_err);
  }
  else if (sl_button_get_state(handle) == SL_SIMPLE_BUTTON_RELEASED) {
    OSFlagPost(&application_event_flags,(OS_FLAGS)APPLICATION_EVENT_FLAG_BTN_OFF,OS_OPT_POST_FLAG_SET,&os_err);
  }
}
```

The LED toggling task now is event-driven when a button is pushed or released. It is also time-driven when no button is pushed or released. This will demonstrate:

1. The LED1 blinks every 1 second.
2. When a button is pushed and remains pushed, the LED1 blinks every 0.2 seconds.
3. When the button is released, the LED1 resumes blinking every 1 second.

3.3.3 Adding an Application Task with the FreeRTOS

Include the header file in app.c.

```
#include"FreeRTOS.h"
#include "task.h"
```

To create a FreeRTOS task, you must declare a task data structure (TCB), allocate a memory space as the stack of the task, write the task's code, and set the task name, priority, and other parameters required by the xTaskCreateStatic() API. The task's priority must be unique in the application and must be lower than the priorities of the Link Layer, Bluetooth Host Task, and Event Handler Task. The lower the number, the higher the priority. Therefore, Silicon Labs recommends that you specify priority numbers greater than or equal to **10** for your application tasks. Bluetooth stack task priorities can be found in the “sl_bt_rtos_config.h” and can be configured in the **Bluetooth Core**-component under Bluetooth – RTOS.

Add the following code at the global level in `app.c`.

```
//Application task
#define SL_BT_RTOS_APPLICATION_PRIORITY         10u

static  void  vTaskApplication(void * pvParameters);
_ALIGNED(8) static StackType_t thread_application_stk[250 & 0xFFFFFFF8u];
_ALIGNED(4) static StaticTask_t thread_application_cb;
```

Add the following code in `app_init()` to create the task.

```
//Application task
  xTaskCreateStatic((TaskFunction_t)vTaskApplication,
                    "Application Task",
                    sizeof(thread_application_stk) / sizeof(StackType_t),
                    NULL,
                    (UBaseType_t)SL_BT_RTOS_APPLICATION_PRIORITY,
                    thread_application_stk,
                    &thread_application_cb);
```

Then, write the task's code in `vTaskApplication()`. A task usually runs infinitely and needs to yield execution to other tasks when it has completed. An execution yield means freeing up CPU time for other tasks. This is usually achieved by calling a time or event API. The following code is the task implementation that turns the LED1 on or off every 1 second.

```
//Application task
static void vTaskApplication(void * pvParameters)
{
  (void)pvParameters;

  while (1) {
    // Put your application code here!
    vTaskDelay(1000);
    sl_led_toggle(&sl_led_led1);
  }
}
```

3.3.4 Adding Initialization with the FreeRTOS

Initializing peripherals may require enabling interrupts. FreeRTOS disables interrupts until the scheduler is started, therefore these operations should be initiated in a separate task. The following code is the task implementation that initializes the relative humidity and temperature (RHT) sensor and logs the current temperature every 1 second.

```
// Sensor task
static void vTaskSensor(void * pvParameters)
{
    (void)pvParameters;

    sl_status_t sc;
    int32_t temperature = 0;
    uint32_t humidity = 0;

    // Put your initialization code here!
    sl_sensor_rht_init();

    while (1) {
        // Put your application code here!
        vTaskDelay(1000);
        sc = sl_sensor_rht_get(&humidity,
                               &temperature);
        sl_app_assert(sc == SL_STATUS_OK,
                      "Error reading temperature");
        sl_app_log("Temperature: %5.2f C\n",
                   (float)temperature / 1000);
    }
}
```

Beginning with SDK v3.1.2, logging and assertion are enhanced. Therefore, the code above can be rewritten as:

```
// Sensor task
static void vTaskSensor(void * pvParameters)
{
  (void)pvParameters;

  sl_status_t sc;
  int32_t temperature = 0;
  uint32_t humidity = 0;

  // Put your initialization code here!
  sl_sensor_rht_init();

  while (1) {
    // Put your application code here!
    vTaskDelay(1000);
    sc = sl_sensor_rht_get(&humidity,
                           &temperature);
    app_assert_status(sc);
    app_log("Temperature: %5.2f C\n",
            (float)temperature / 1000);
  }
}
```

3.3.5 Implementing a Time- and Event-Driven Task with the FreeRTOS

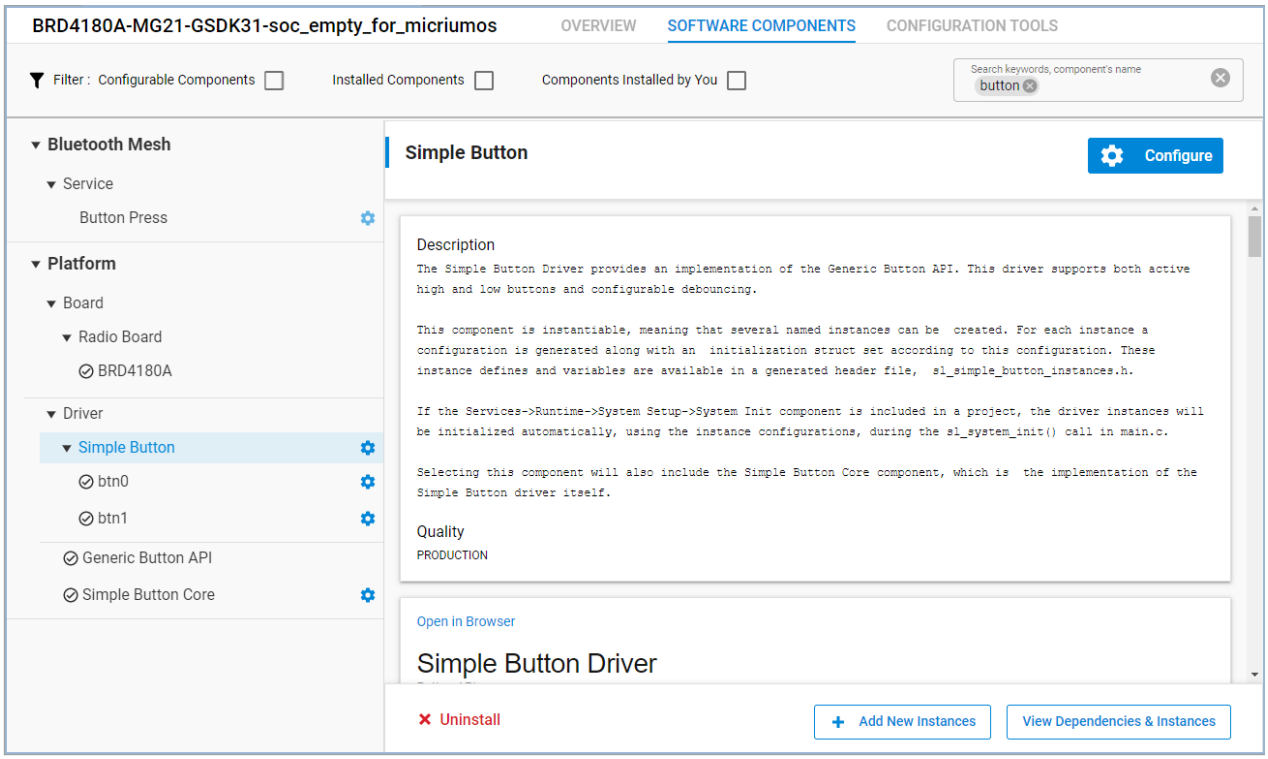
A task can be time-driven, event-driven, or both time- and event-driven. The model determines when the procedure is executed:

1. A timer expires.
2. An event occurs.
3. Either 1 or 2.

The Link Layer Task, Bluetooth Host Task, and Event Handler Task are event-driven and the LED toggling task demonstrated in section 3.3 Adding Application Tasks is time-driven.

The LED toggling task demonstrated in section 3.3 Adding Application Tasks is a good example to be changed to a time- and event-driven task. This task calls a time API to suspend itself for a period of specific time. A time- and event- driven task requires an event API that supports timeout so the task suspends its execution until either an event occurs or a timeout expires. This section demonstrates how an LED blinks at different frequencies based on whether or not a button on the Wireless Starter Kit is pushed.

The Simple Button component is required to access the buttons on the Wireless Starter Kit. Follow the steps in section 3.2.2 Event Handlers to install the component and create the **btn0** and **btn1** instances in the Software Components Configuration tool as shown in the following figure.



Add the following code in `app.c` to include the header file that has the declaration of the button state changed callback function.

```
#include "sl_simple_button_instances.h"
#include “event_groups.h“
```

The demonstration uses FreeRTOS Event Groups (or “flags”) APIs for inter-task communication. The principle of the Event Group is a task waits for a bit (or a flag) or some bits (flags) in an Event Group to be set and another task sets a bit in the same Event Group when a corresponding event occurs. An Event Group has 24-bits implemented in the default configuration. Therefore, it is a bit pattern which is a combination of bits of a 24-bit variable.

The easiest way for tasks using an event flag group is to declare it as a global variable. Add the following code at the global level in `app.c`.

```
// Declare a variable to hold the handle of the created Event Group.
EventGroupHandle_t xEventGroupApplicationHandle;

// Declare a variable to hold the data associated with the created Event Group.
StaticEventGroup_t xEventGroupApplicationData;

#define APPL_EVENT_GROUP_BIT_BTN_ON   ((EventBits_t)1)
#define APPL_EVENT_GROUP_BIT_BTN_OFF  ((EventBits_t)2)
```

Add the following code in `app_init()` to create the Event Group before creating the Application Task.

```
xEventGroupApplicationHandle = xEventGroupCreateStatic( &xEventGroupApplicationData );
```

Next, modify the task's code in `vTaskApplication()` as follows. The task calls `xEventGroupWaitBits()` to suspend its execution and waits for any of the APPL_EVENT_GROUP_BIT_BTN_ON and APPL_EVENT_GROUP_BIT_BTN_OFF bits to be set. The task resumes to execute if any of the bits is set or when the timeout xTicksToWait expires. Therefore, the task toggles the LED1 at periodic intervals when no bit is set and changes the interval when a bit isset.

If you want to implement an event-driven operation in this example, comment out the `sl_led_toggle()` from the TimeOut part. The task then toggles LED1 only after any of the bits are set. It will naturally still time out, but in practice it is event-driven as it immediately starts waiting for events again. Also the timeout could be set as long as possible to get fewer timeouts.

```
//Application task
static void vTaskApplication(void * pvParameters)
{
  (void)pvParameters;
  EventBits_t uxBits;
  TickType_t xTicksToWait = 1000 / portTICK_PERIOD_MS; // Wait maximum of 1000 ms

  while (1) {
    // Put your application code here!

    uxBits = xEventGroupWaitBits(
               xEventGroupApplicationHandle,   // The Event Group being tested.
               APPL_EVENT_GROUP_BIT_BTN_ON | APPL_EVENT_GROUP_BIT_BTN_OFF, // The bits to wait for
               pdTRUE,        // bits should be cleared before returning.
               pdFALSE,       // Don't wait for both bits, either bit will do.
               xTicksToWait );// Wait a maximum of 1000 ms (or 200 ms) for either bit to be set.

    if (uxBits & APPL_EVENT_GROUP_BIT_BTN_ON) {
      xTicksToWait = 200 / portTICK_PERIOD_MS; // Set timeout into 200 ms
      sl_led_toggle(&sl_led_led1);
    }
    else if (uxBits & APPL_EVENT_GROUP_BIT_BTN_OFF) {
      xTicksToWait = 1000 / portTICK_PERIOD_MS; // Set timeout into 1000 ms
      sl_led_toggle(&sl_led_led1);
    }
    else { // TimeOut
      sl_led_toggle(&sl_led_led1); // Comment out for an event-driven operation
    }
  }
}
```

Finally, write the code in the callback function `sl_button_on_change()` to set the bits. The button driver calls this function when the state of the Button0 or Button1 has changed. The function sets the APPL_EVENT_GROUP_BIT_BTN_ON bit when a button is pushed and sets the APPL_EVENT_GROUP_BIT_BTN_OFF bit when a button is released. The code should like the following.

```
void sl_button_on_change(const sl_button_t *handle)
{
  if (sl_button_get_state(handle) == SL_SIMPLE_BUTTON_PRESSED) {
    xEventGroupSetBits(xEventGroupApplicationHandle, APPL_EVENT_GROUP_BIT_BTN_ON);
  }
  else if (sl_button_get_state(handle) == SL_SIMPLE_BUTTON_RELEASED) {
    xEventGroupSetBits(xEventGroupApplicationHandle, APPL_EVENT_GROUP_BIT_BTN_OFF);
  }
}
```

The LED toggling task now is event-driven when a button is pushed or released. It is also time-driven when no button is pushed or released. This will demonstrate:

1. The LED1 blinks every 1 second.
2. When a button is pushed and remains pushed, the LED1 blinks every 0.2 seconds.
3. When the button is released, the LED1 resumes blinking every 1 second.

4. Additional Resources

Consult the following resources for additional information.

- Silicon Labs Bluetooth API Documentation
- Silicon Labs Bluetooth C Developer's Guide
- Silicon Labs Platform Documentation
- Micriμm OS Documentation
- The FreeRTOS Kernel

Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support & Community
www.silabs.com/community

SILICON LABS

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

www.silabs.com