AN1322: Dynamic Multiprotocol Development with *Bluetooth*® and Zigbee EmberZNet SDK 7.0 and Higher

This application note provides details on developing Dynamic Multiprotocol applications using Bluetooth and Zigbee in SDK 4.0 and higher. It describes how to configure applications in Simplicity Studio using Zigbee EmberZNet SDK v. 7.0 and higher. It then provides a detailed walkthrough on how the underlying code functions. For details on Dynamic Multiprotocol Application development that apply to all protocol combinations see *UG305: Dynamic Multiprotocol User's Guide*.

Zigbee EmberZNet SDK v7.0 introduced a component-based project architecture that replaced AppBuilder. If you are working with Zigbee EmberZNet SDK v 6.10.x or lower, see *AN1133: Dynamic Multiprotocol Developer with Bluetooth and Zigbee EmberZNet SDK 6.x and Lower* for this information.

KEY POINTS

- Generating and loading dynamic multiprotocol example applications.
- Adding dynamic multiprotocol functionality to an existing Zigbee project.
- Details on the application User Interface.
- How the Zigbee example applications function.
- How the Bluetooth application functions.

1 Introduction

The example applications referenced here can be controlled either from a protocol-specific switch application or from a Bluetooth-enabled smartphone app. This application note provides details on how these examples are designed and implemented. It also describes how to generate, compile, and load example application code, and how to add dynamic multiprotocol functionality to an existing Zigbee project. The application note is intended to be used when developing your own Zigbee/Bluetooth dynamic multiprotocol implementations.

Note: The Zigbee dynamic multiprotocol solution is currently only supported for SoC architectures. Support for NCP architectures has been deprecated in favor of DMP RCP. Please contact Silicon Labs Sales for more information on our multiprotocol software roadmap.

1.1 Resources

- *UG305: Dynamic Multiprotocol User's Guide* provides details on:
  - Dynamic Multiprotocol Architecture
  - Radio Scheduler operation (with examples)
  - Task Priority management
- *AN1135: Using Third Generation Non-Volatile Memory (NVM3) Data Storage* explains how NVM3 can be used as non-volatile data storage in Dynamic Multiprotocol applications with Zigbee and Bluetooth.

1.2 Development Environment Requirements

- Simplicity Studio 5
- SDK 4.0 or higher, which includes Zigbee EmberZNet SDK version 7.0.0 or higher and Bluetooth SDK 3.3 or higher.
- An EFR32 chip with at least 512 kB of flash (required to run all the necessary software components)

To work with the demos, download the EFR Connect app from Google Play Store or App Store.

2 Working with the Zigbee/Bluetooth Examples

This section describes

- How to build and flash the dynamic multiprotocol applications supplied with the Zigbee EmberZNet SDK.
- How to add Bluetooth to a Zigbee project and turn it into a dynamic multiprotocol project.

2.1 Application Generation

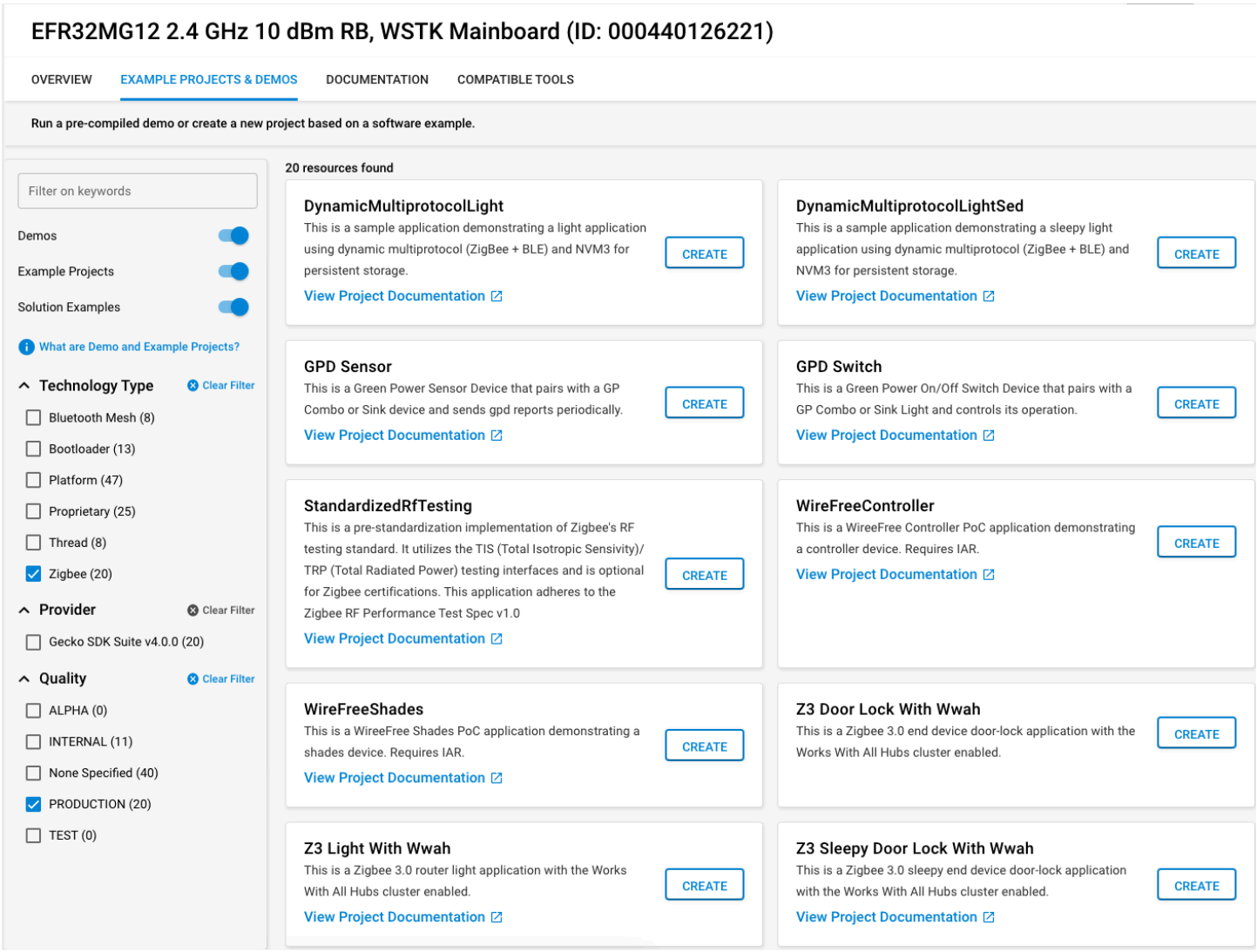
To work with Zigbee/Bluetooth dynamic multiprotocol applications as decribed in this application note, you must install SDK 4.0 or higher. The applications can be built with GCC (The GNU Compiler Collection) or IAR-EWARM. See *QSG180: Getting Started with EmberZNet PRO* for information on installing the SDKs and setting up compilers.

Dynamic multiprotocol applications are generated, built, and uploaded in the same way as other applications. If you are not familiar with these procedures, see *QSG180: Zigbee EmberZNet Quick-Start Guide for SDK 7.0 and Higher* for details. The dynamic multiprotocol applications included with the EmberZNet SDK are:

- **Zigbee BLE – DynamicMultiprotocolLight** is an application designed to demonstrate a DMP device with Zigbee 3.0 coordinator capabilities.
- **Zigbee BLE – DynamicMultiprotocolLightSed** is an application designed to demonstrate a DMP device with SED capabilities.

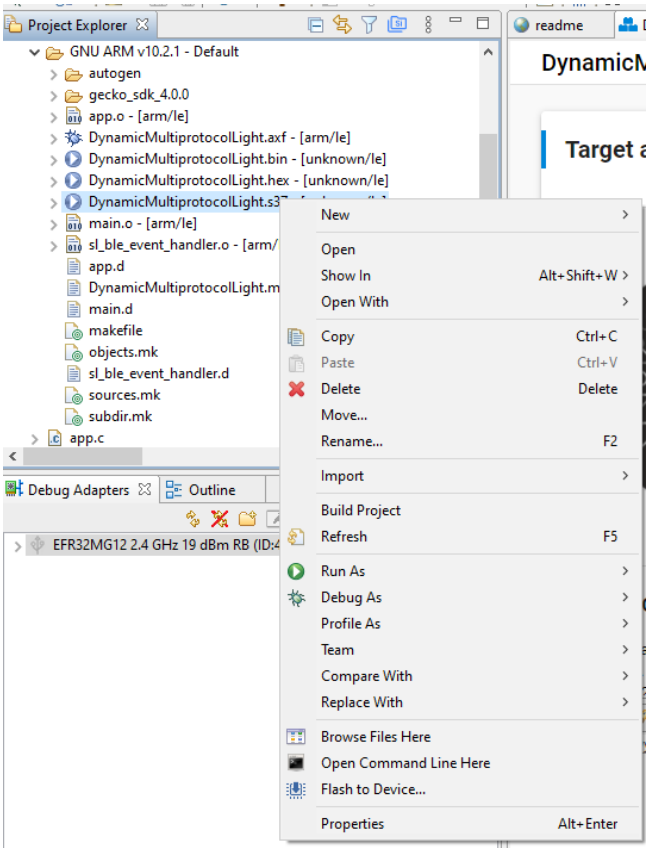
The following summary procedure uses the **DynamicMultiprotocolLight** example application.

1. In Simplicity Studio, start a new project based on the **DynamicMultiprotocolLight** example. It is easiest to select Zigbee as the Technology Type and filter on the word “dynamic.”



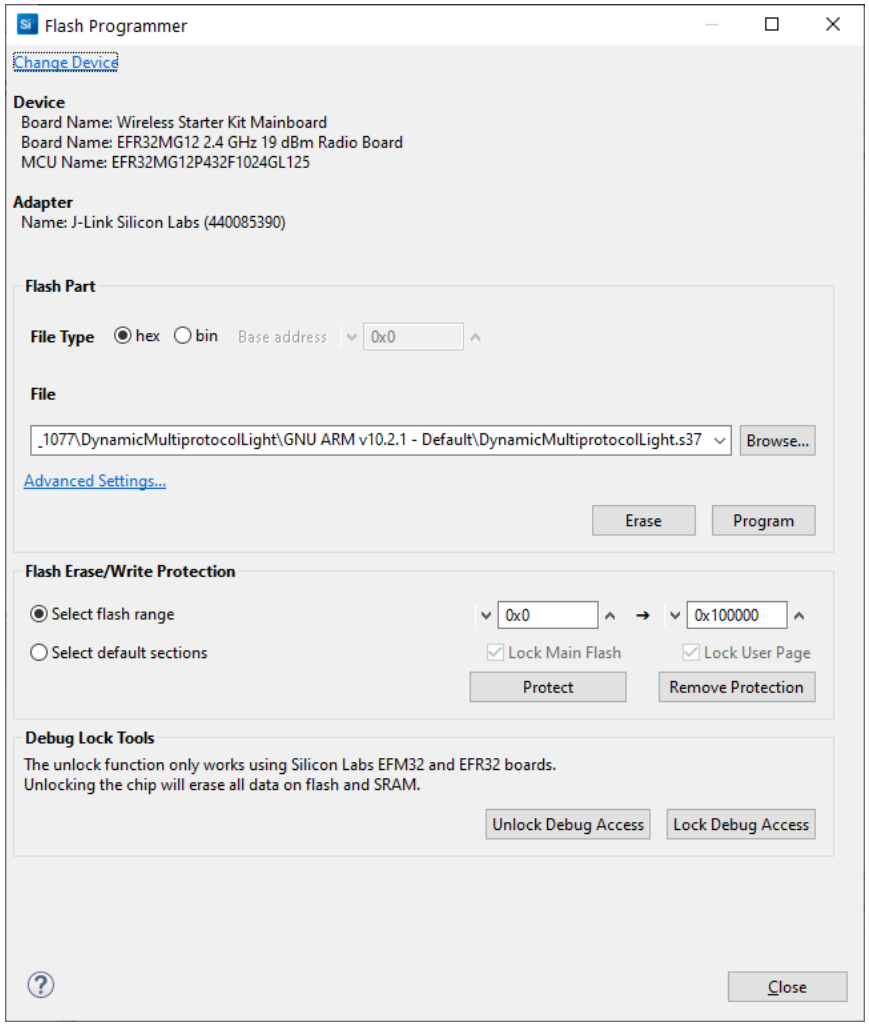
2. Once the project is created, files are generated automatically. Click **Build** (hammer icon) to build the application image.

3. To flash the application image, in Project Explorer view right-click the application .s37 file and select **Flash to Device**.

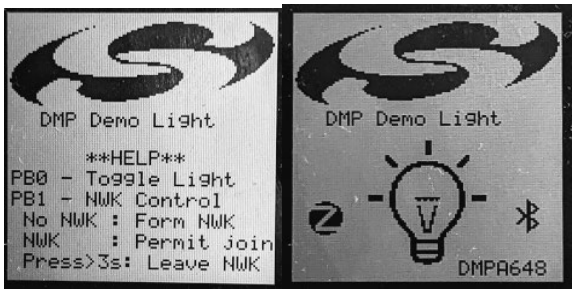


If you have more than one device connected, select the target. The Flash Programmer opens.

4. The path of the .s37 file should be auto populated. Click **Program** to flash the file to the target.



5. Application load success indicators are code-dependent. If the example projects are being used on a development board that supports LCD functionality, the LCD displays the following screen on power up. Press button PB0 to change to the light display. On other development boards that do not have additional peripherals to support a fully featured user interface, use the command line interface to run various commands.



Note: Silicon Labs examples require a bootloader. If the bootloader gets erased, an easy way to load a bootloader is to run the Dynamic Multiprotocol Light demo. This installs a combined bootloader/application image. Then you can flash your own application image to update only the application area. If you are using a board that is not compatible with the available demos, then you can load a bootloader by selecting an example, such as **SPI Flash Storage Bootloader (single image)**, and building it and flashing it as described above.

2.2 Converting a Zigbee Application to a Zigbee/Bluetooth LE Dynamic Multiprotocol Application

This section describes the configuration changes required to convert a working Zigbee application into a Zigbee/Bluetooth LE Dynamic Multiprotocol application. The instructions present the generic steps for the conversion, with specific examples based on turning the Z3Light example into the equivalent of **DynamicMultiprotocolLight.**

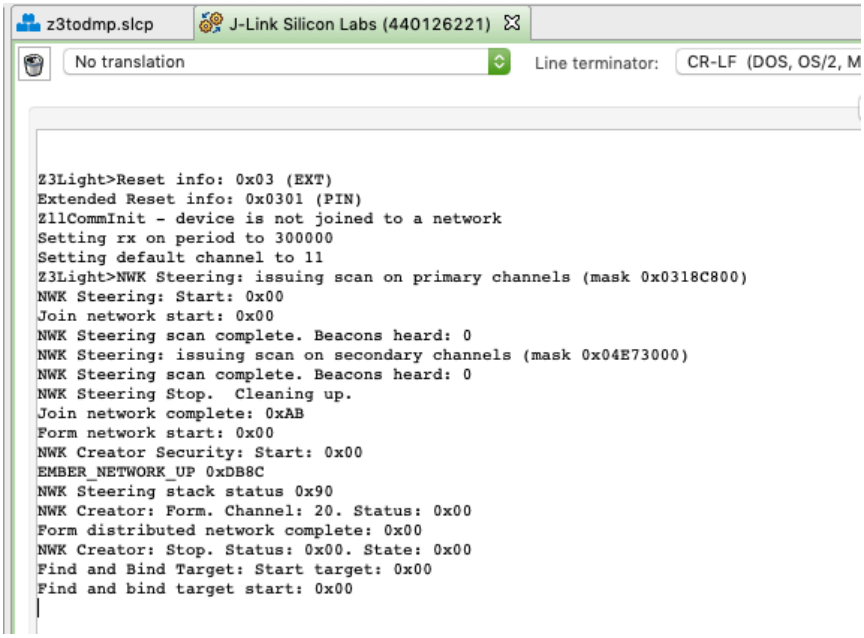
Requirements:

- Zigbee application set up to build with IAR ARM or GCC (these instructions use Z3 Light)
- Any EFR32 part with a minimum of 512 kB of flash and 64 kB of RAM

Note: The Dynamic Multiprotocol examples do not support OTA updates out of the box. To support OTA updates, uninstall the Zigbee LCD component. This frees up the port pins that are multiplexed with the external flash.

2.2.1 Generate and Build the Zigbee Application

The purpose of this step is to verify that the base Zigbee application had loaded and is working correctly, and that output is printing to the console. This example uses the Z3Light sample application. It begins with the default settings, so that the configuration changes are clear. Generate and build the project, load it to the board and check the Serial 1 output to make sure it is up and running.

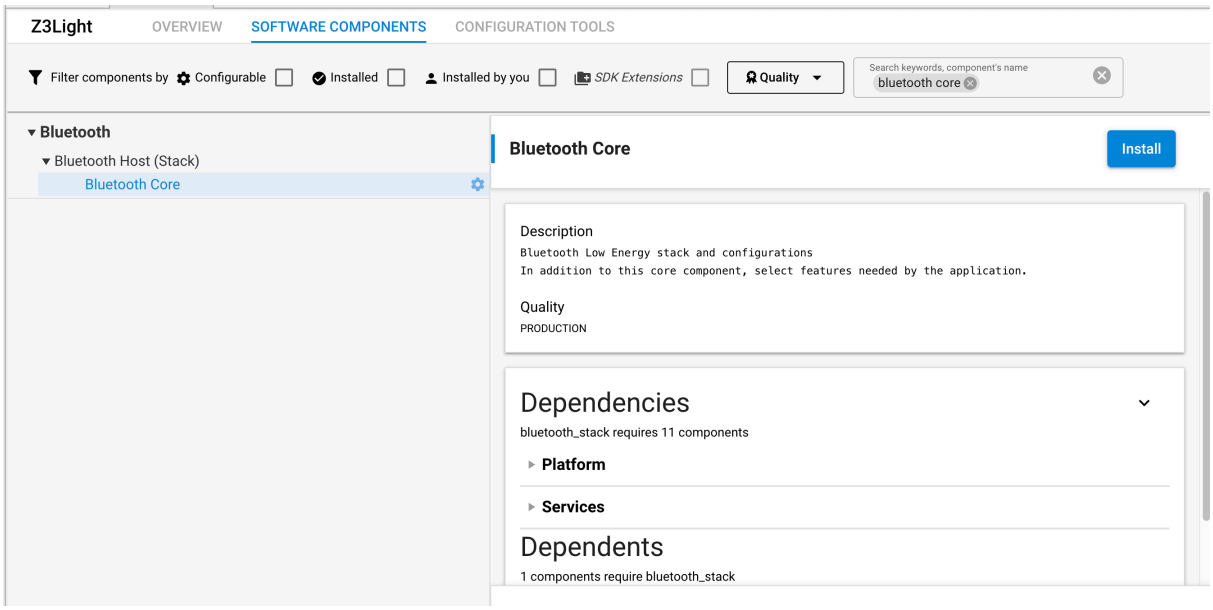


2.2.2 Configure the project

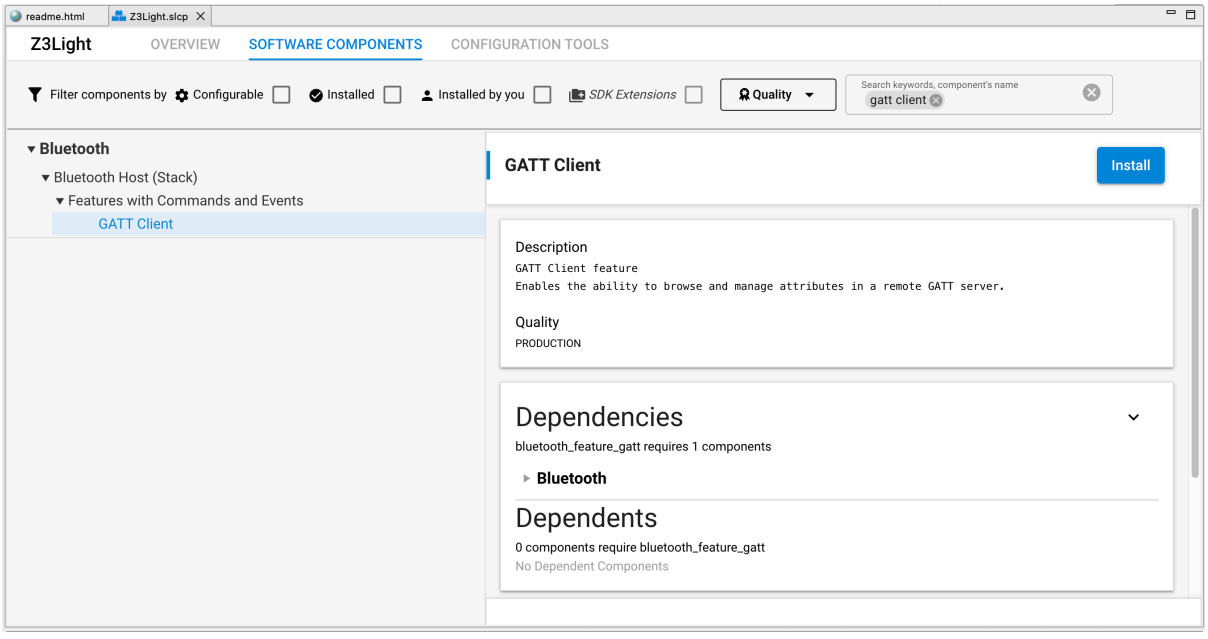
To convert the Z3Light application into a Zigbee-Bluetooth LE multiprotocol application similar to the DMP Light, follow the steps below:

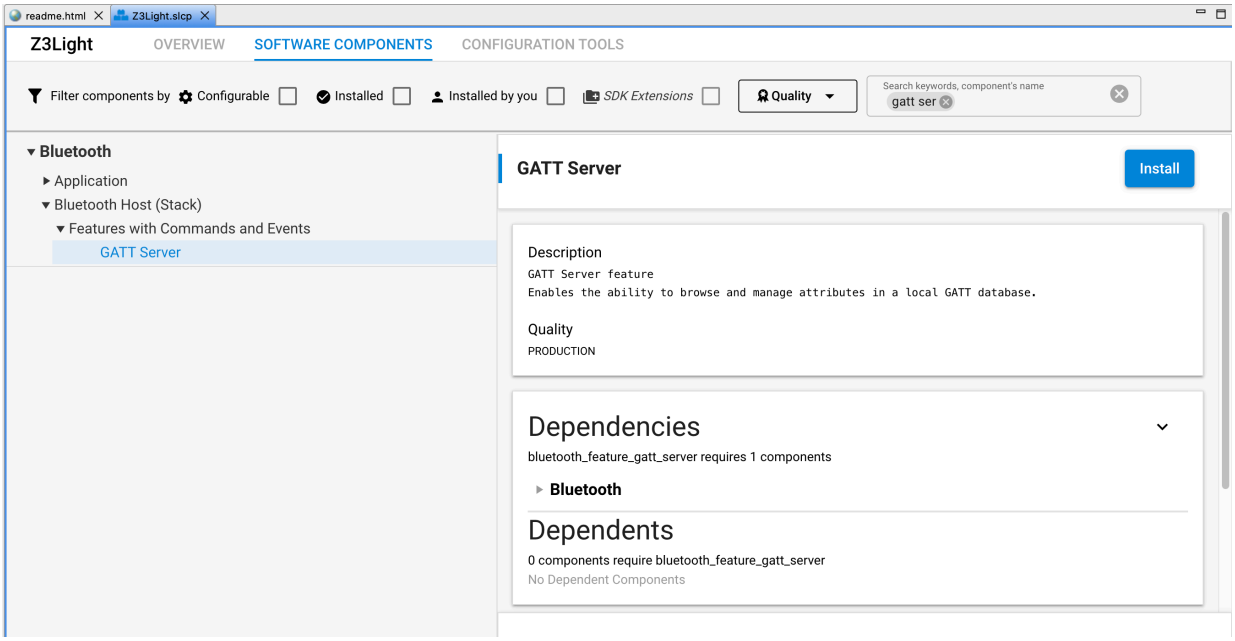
1. Navigate to the SOFTWARE COMPONENTS tab on the Z3Light project and search for and add the following components.
   - **Bluetooth Core** - Reason: This is the Bluetooth stack core component

     Note: Installing this enables multiple protocol stacks on the project and thereby also enables the CMSIS RTOS2 layer and FreeRTOS kernel, which is the default RTOS implementation. Micrium RTOS is also supported.



   - **GATT Client, GATT Server (Static GATT database), Security Manager, System** - Reason: Basic Bluetooth building blocks.

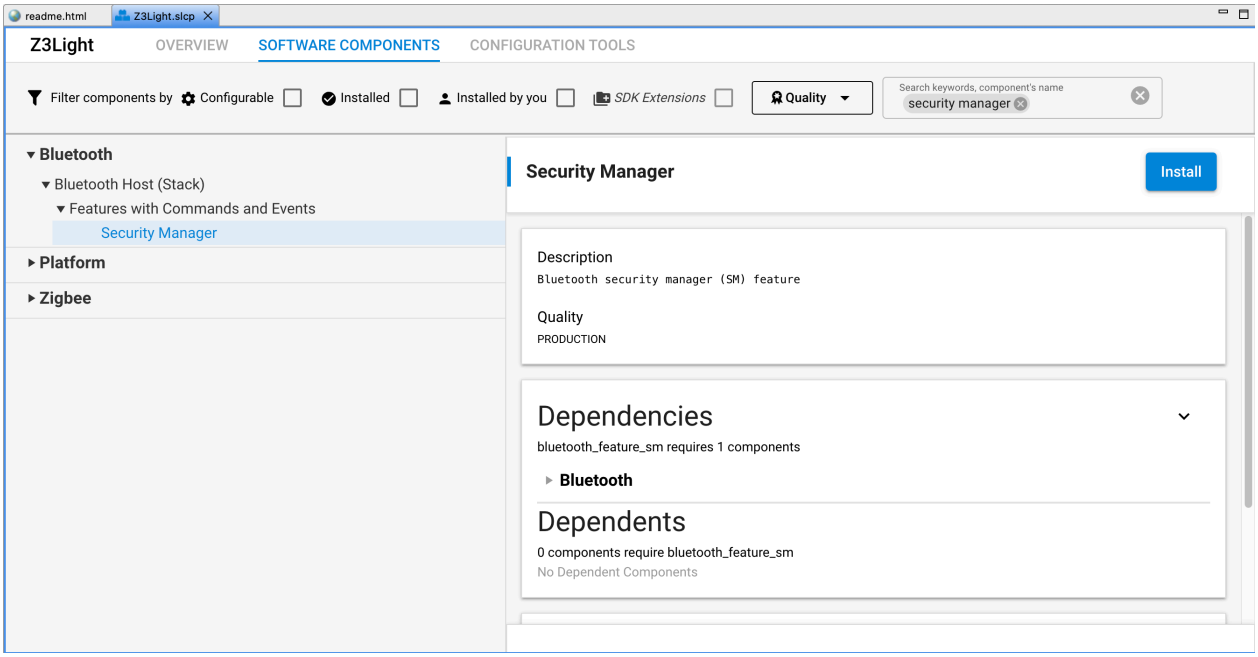


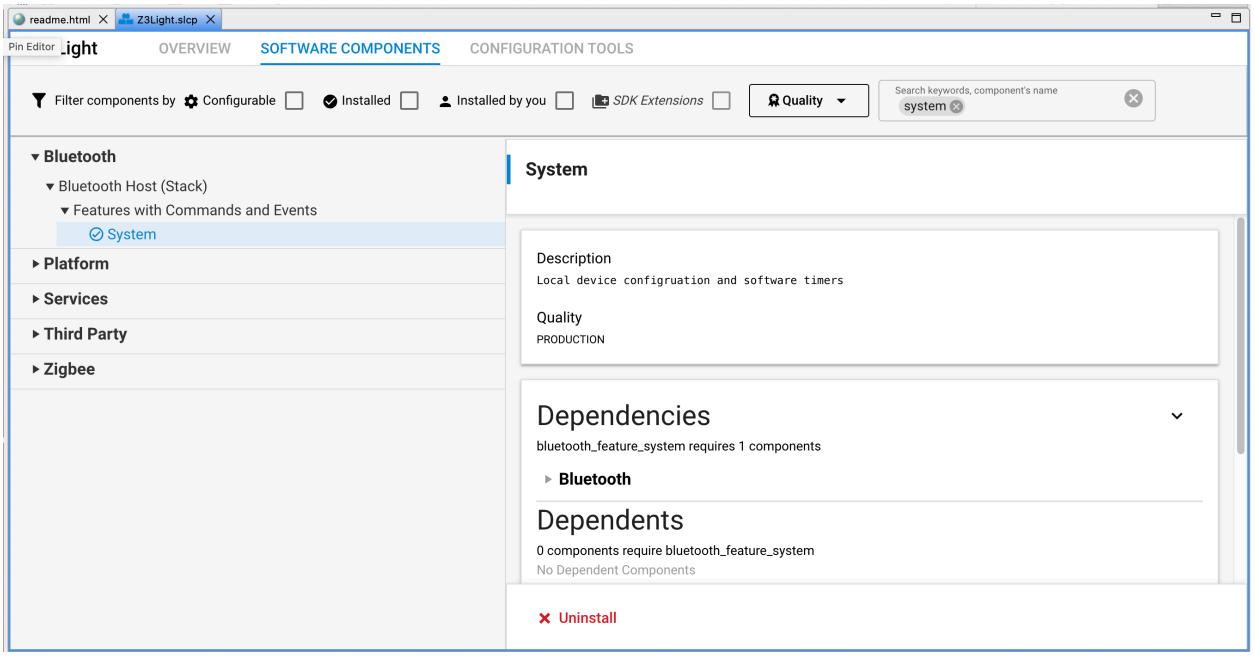


Static GATT Database and Configuration

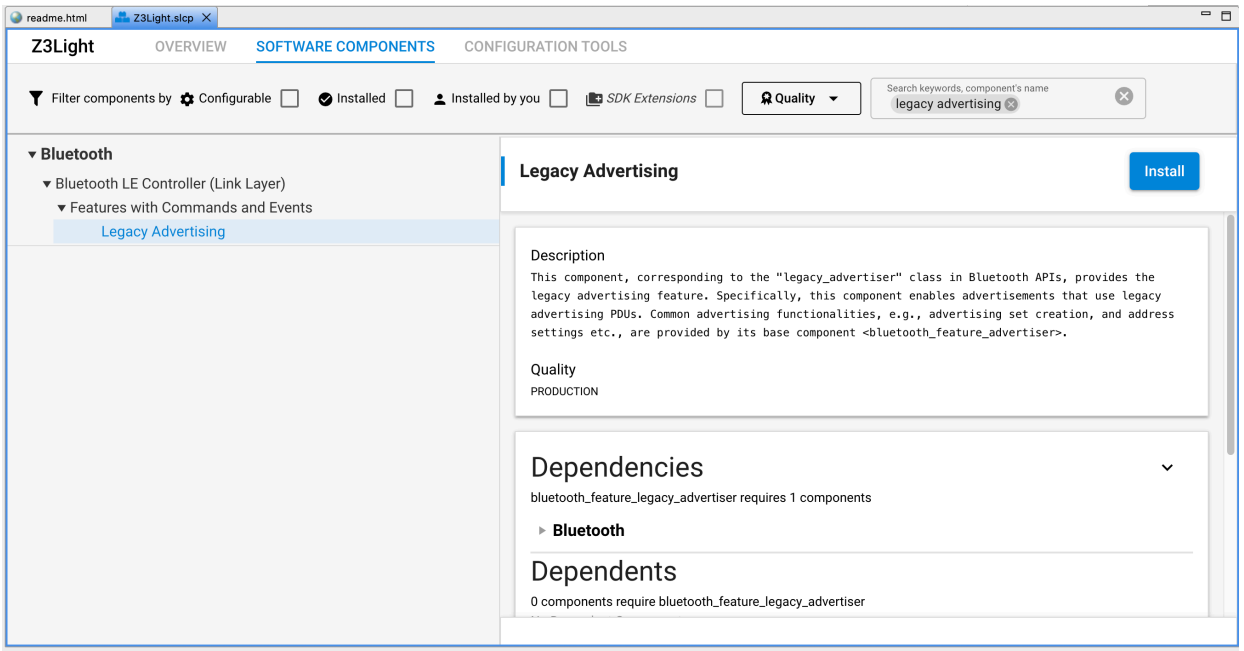
- Adds a basic, static GATT database. It can be customized by the GATT Configurator tool. It is compatible with the Dynamic GATT feature <bluetooth_feature_dynamic_gattdb> - static and dynamic GATT databases can co-exist. It will also provide the GATT Configurator tool.

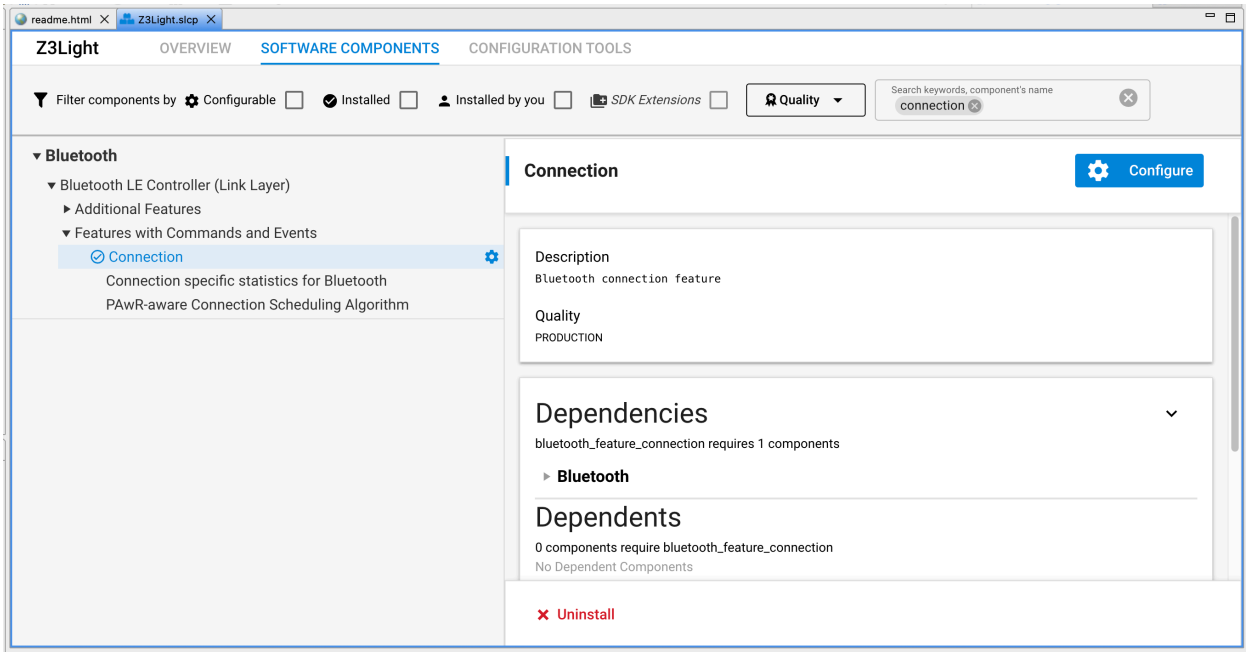
View Documentation

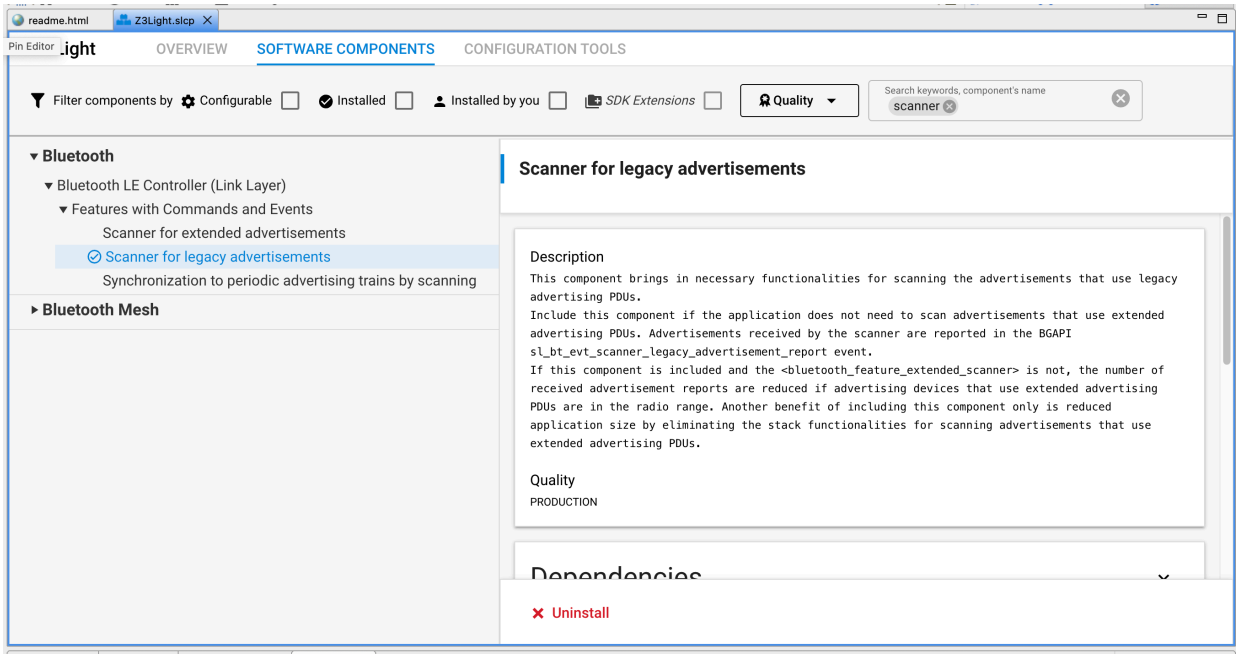




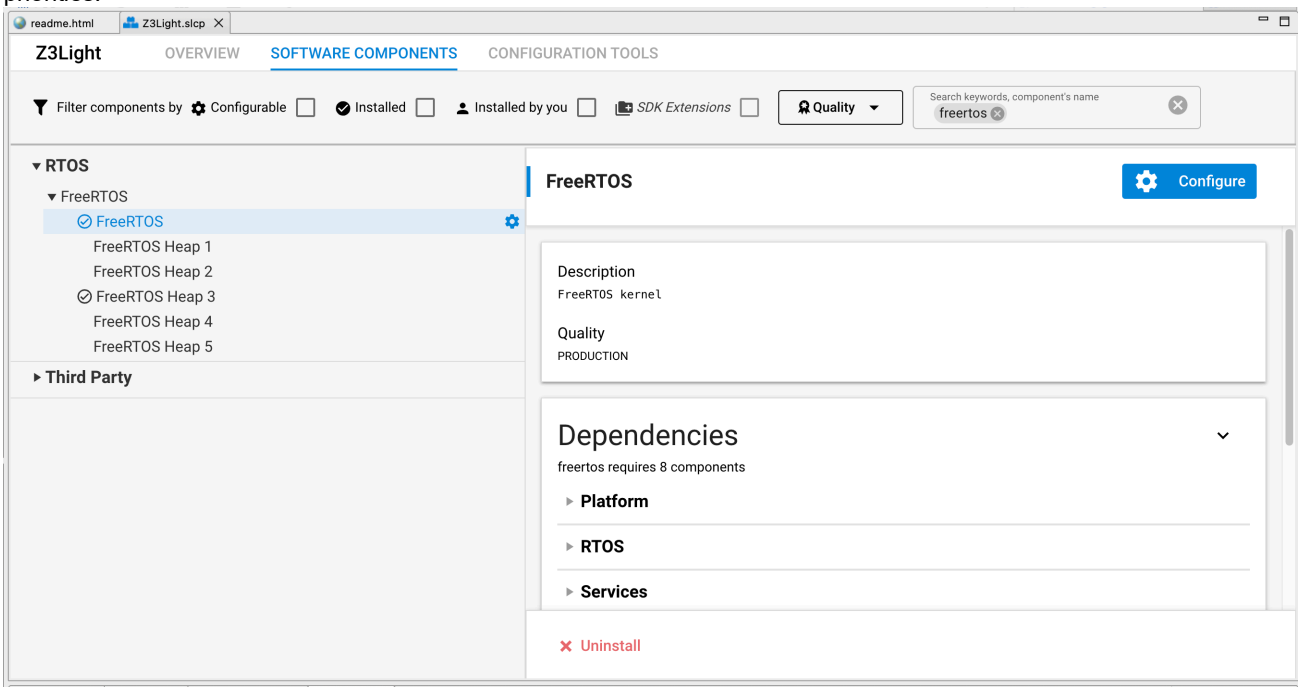
- **Legacy Advertising, Connection, Scanner**. Reason: Basic Bluetooth features.

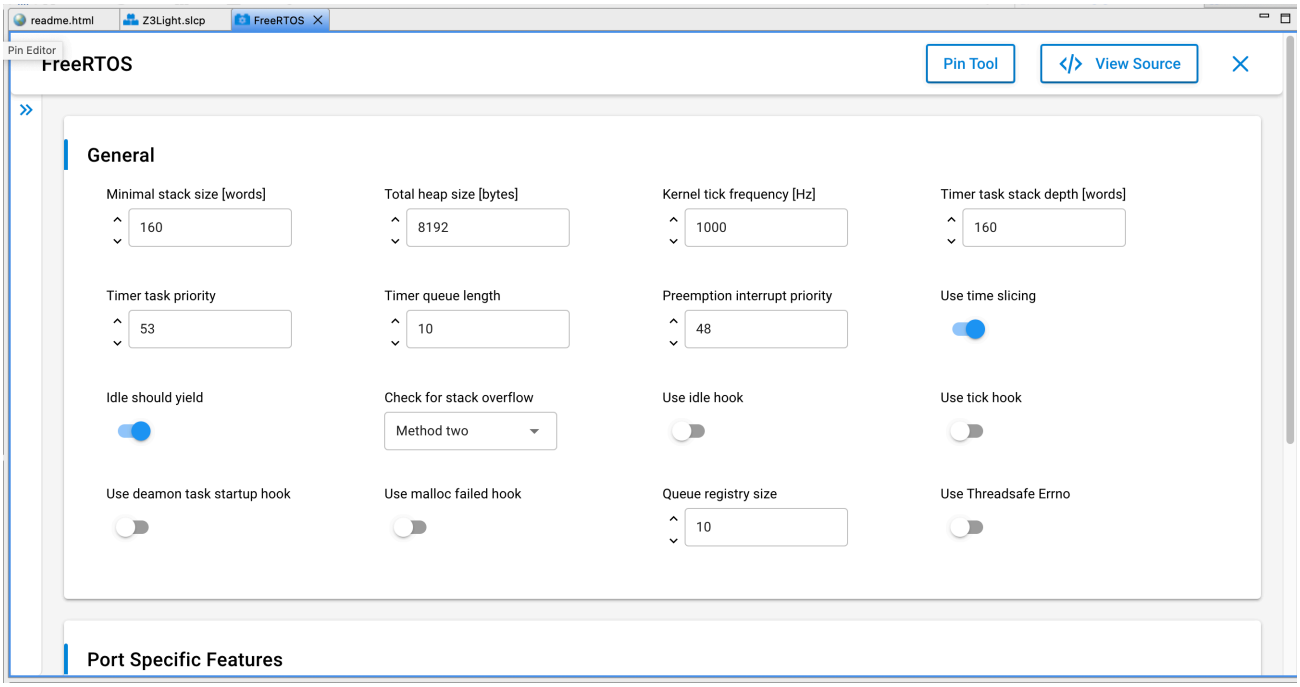



readme.html
Z3Light.slcp
Z3Light
OVERVIEW
SOFTWARE COMPONENTS
CONFIGURATION TOOLS
Filter components by
Configurable
Installed
Installed by you
SDK Extensions
Quality
Search keywords, component's name
connection
Bluetooth
Bluetooth LE Controller (Link Layer)
Additional Features
Features with Commands and Events
Connection
Connection specific statistics for Bluetooth
PAwR-aware Connection Scheduling Algorithm
Connection
Configure
Description
Bluetooth connection feature
Quality
PRODUCTION
Dependencies
bluetooth_feature_connection requires 1 components
Bluetooth
Dependents
0 components require bluetooth_feature_connection
No Dependent Components
Uninstall


readme.html
Z3Light.slcp
Pin Editor
OVERVIEW
SOFTWARE COMPONENTS
CONFIGURATION TOOLS
Filter components by
Configurable
Installed
Installed by you
SDK Extensions
Quality
Search keywords, component's name
scanner
Bluetooth
Bluetooth LE Controller (Link Layer)
Features with Commands and Events
Scanner for extended advertisements
Scanner for legacy advertisements
Synchronization to periodic advertising trains by scanning
Bluetooth Mesh
Scanner for legacy advertisements
Description
This component brings in necessary functionalities for scanning the advertisements that use legacy advertising PDUs.
Include this component if the application does not need to scan advertisements that use extended advertising PDUs. Advertisements received by the scanner are reported in the BGAPI sl_bt_evt_scanner_legacy_advertisement_report event.
If this component is included and the <bluetooth_feature_extended_scanner> is not, the number of received advertisement reports are reduced if advertising devices that use extended advertising PDUs are in the radio range. Another benefit of including this component only is reduced application size by eliminating the stack functionalities for scanning advertisements that use extended advertising PDUs.
Quality
PRODUCTION
Uninstall

- **If your application uses Free RTOS, configure FreeRTOS component and increase Timer task priority to 53.** Reason: Due to the usage of RTOS event flags in the Bluetooth stack, the timer task priority must be higher than all of the Bluetooth RTOS task priorities.





2. Add an implementation of `sl_bt_on_event(sl_bt_msg_t* evt)` in your app.c file. The following is an example implementation of the Bluetooth LE event handler that starts advertisements on boot and prints out information as some of the most common events occur:

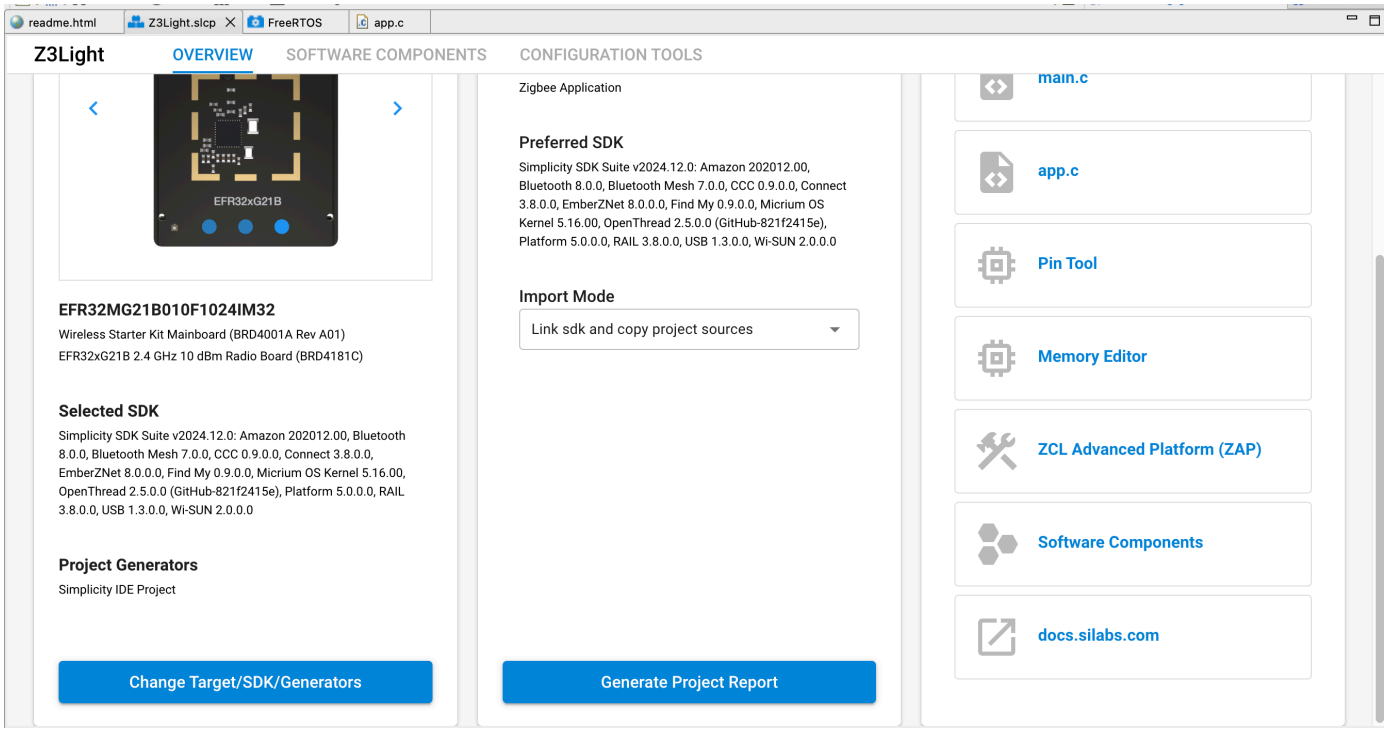
```
#include "sl_bluetooth.h"
#include "sl_bluetooth_advertiser_config.h"
#include "sl_bluetooth_connection_config.h"

#include "gatt_db.h"
uint8_t adv_handle;
```

```
#define DEVNAME_LEN 8
#define UUID_LEN 16 // 128-bit UUID

// to convert hex number to its ascii character
uint8_t ascii_lut[] = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E',
'F' };

void zb_ble_dmp_print_ble_address(uint8_t *address)
{
  sl_zigbee_app_debug_print("\nBLE address: [%02X %02X %02X %02X %02X %02X]\n",
                            address[5], address[4], address[3],
                            address[2], address[1], address[0]);
}

void enableBleAdvertisements(void)
{
  sl_status_t status;

  /* Create the device Id and name based on the 16-bit truncated bluetooth address
     Copy to the local GATT database - this will be used by the BLE stack
     to put the local device name into the advertisements, but only if we are
     using default advertisements */
  uint8_t type;
  bd_addr ble_address;
  static char devName[DEVNAME_LEN];

  status = sl_bt_system_get_identity_address(&ble_address, &type);
  if ( status != SL_STATUS_OK ) {
    sl_zigbee_app_debug_println("Unable to get BLE address. Errorcode: 0x%x", status);
    return;
  }
  uint16_t devId = ((uint16_t)ble_address.addr[1] << 8) + (uint16_t)ble_address.addr[0];

  devName[0] = 'D';
  devName[1] = 'M';
  devName[2] = 'P';
  devName[3] = ascii_lut[( (ble_address.addr[1] & 0xF0) >> 4)];
  devName[4] = ascii_lut[(ble_address.addr[1] & 0x0F)];
  devName[5] = ascii_lut[( (ble_address.addr[0] & 0xF0) >> 4)];
  devName[6] = ascii_lut[(ble_address.addr[0] & 0x0F)];
  devName[7] = '\0';

  sl_zigbee_app_debug_println("devName = %s", devName);
  status = sl_bt_gatt_server_write_attribute_value(gattdb_device_name,
                                                   0,
                                                   strlen(devName),
                                                   (uint8_t *)devName);

  if ( status != SL_STATUS_OK ) {
    sl_zigbee_app_debug_println("Unable to sl_bt_gatt_server_write_attribute_value device name. Er-
rorcode: 0x%x", status);
    return;
  }


  status = sl_bt_advertiser_set_timing(adv_handle,
                                       (100 / 0.625), //100ms min adv interval in terms of 0.625ms
                                       (100 / 0.625), //100ms max adv interval in terms of 0.625ms
                                       0,   // duration : continue advertisement until stopped
                                       0);   // max_events :continue advertisement until stopped
  if (status != SL_STATUS_OK) {
    return;
  }
  status = sl_bt_advertiser_set_report_scan_request(adv_handle, 1);     //scan request reported as
events
  if (status != SL_STATUS_OK) {
    return;
```

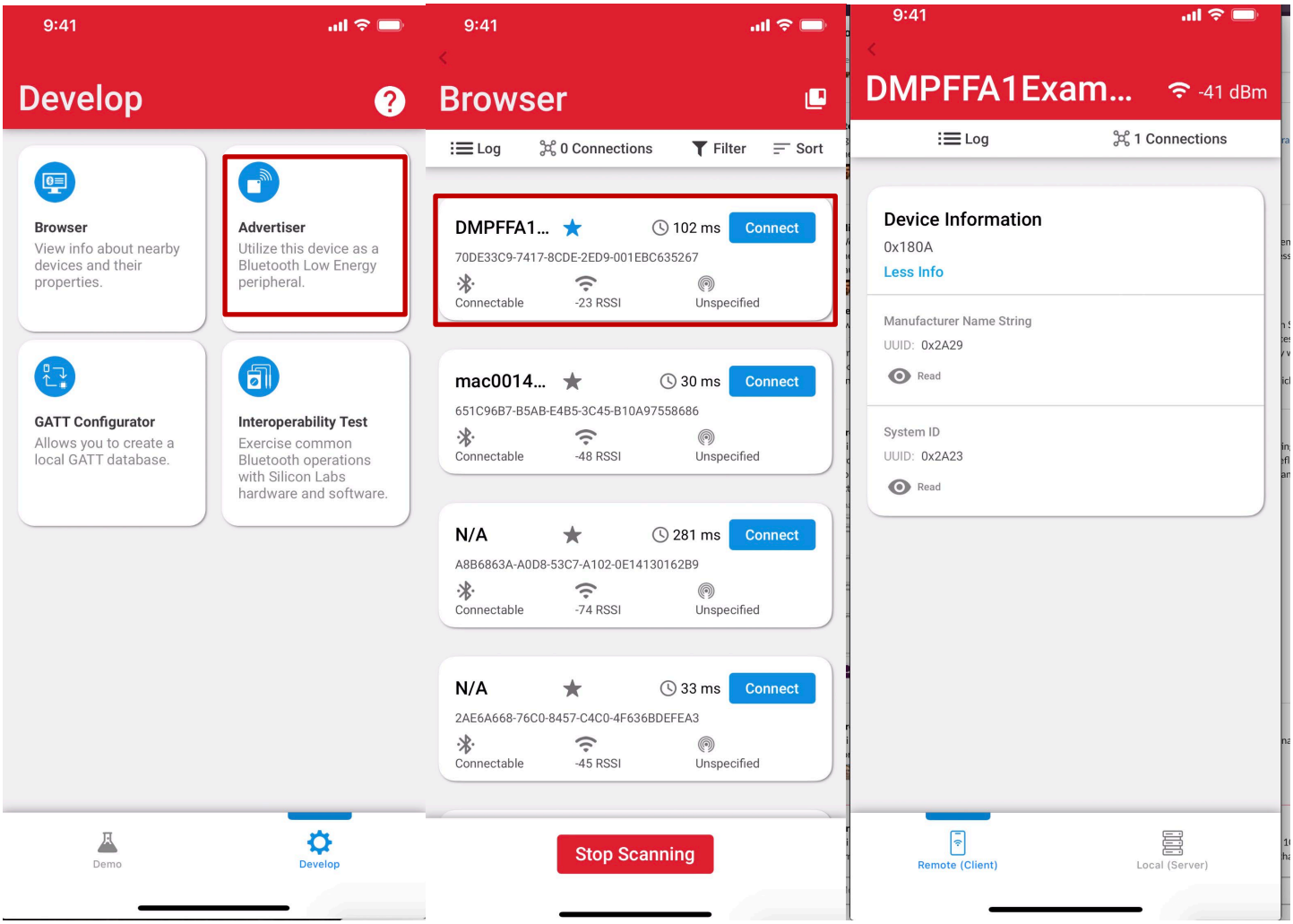
```
  }
  /* Start advertising in user mode and enable connections*/
  status = sl_bt_legacy_advertiser_start(adv_handle,
                                         sl_bt_advertiser_connectable_scannable);
  if ( status ) {
    sl_zigbee_app_debug_println("sl_bt_legacy_advertiser_start ERROR : status = 0x%0X", status);
  } else {
    sl_zigbee_app_debug_println("BLE custom advertisements enabled");
  }
}


/** @brief
 *
 * This function is called from the BLE stack to notify the application of a
 * stack event.
 */
void sl_bt_on_event(sl_bt_msg_t* evt)
{
  switch (SL_BT_MSG_ID(evt->header)) {

    case sl_bt_evt_system_boot_id: {
      bd_addr ble_address;
      uint8_t type;
      sl_status_t status = sl_bt_system_hello();
      sl_zigbee_app_debug_println("BLE hello: %s",
                                  (status == SL_STATUS_OK) ? "success" : "error");

      status = sl_bt_system_get_identity_address(&ble_address, &type);
      zb_ble_dmp_print_ble_address(ble_address.addr);

      #define SCAN_WINDOW 5
      #define SCAN_INTERVAL 10

      status = sl_bt_scanner_set_parameters(sl_bt_scanner_scan_mode_active,
                                            (uint16_t)SCAN_INTERVAL,
                                            (uint16_t)SCAN_WINDOW);

      status = sl_bt_advertiser_create_set(&adv_handle);
      if (status) {
        sl_zigbee_app_debug_println("sl_bt_advertiser_create_set status 0x%02x", status);
      }

      // start advertising
      enableBleAdvertisements();
    }
    break;

    case sl_bt_evt_connection_opened_id: {
      sl_zigbee_app_debug_println("sl_bt_evt_connection_opened_id \n");
      sl_bt_evt_connection_opened_t *conn_evt =
        (sl_bt_evt_connection_opened_t*) &(evt->data);

      //preferred phy 1: 1M phy, 2: 2M phy, 4: 125k coded phy, 8: 500k coded phy
      //accepted phy 1: 1M phy, 2: 2M phy, 4: coded phy, ff: any
      sl_bt_connection_set_preferred_phy(conn_evt->connection, sl_bt_gap_phy_1m, 0xff);

      enableBleAdvertisements();
      sl_zigbee_app_debug_println("BLE connection opened");
    }
    break;

    case sl_bt_evt_connection_phy_status_id: {
      sl_bt_evt_connection_phy_status_t *conn_evt =
        (sl_bt_evt_connection_phy_status_t *)&(evt->data);
      // indicate the PHY that has been selected
      sl_zigbee_app_debug_println("now using the %dMPHY\r\n",
```

```
                                    conn_evt->phy);
    }
    break;

    case sl_bt_evt_connection_closed_id: {
      sl_bt_evt_connection_closed_t *conn_evt =
        (sl_bt_evt_connection_closed_t*) &(evt->data);

      // restart advertising, set connectable
      enableBleAdvertisements();
      sl_zigbee_app_debug_println(
        "BLE connection closed, handle=0x%02x, reason=0x%02x",
        conn_evt->connection, conn_evt->reason);
    }
    break;

    default:
      break;
  }
}
```

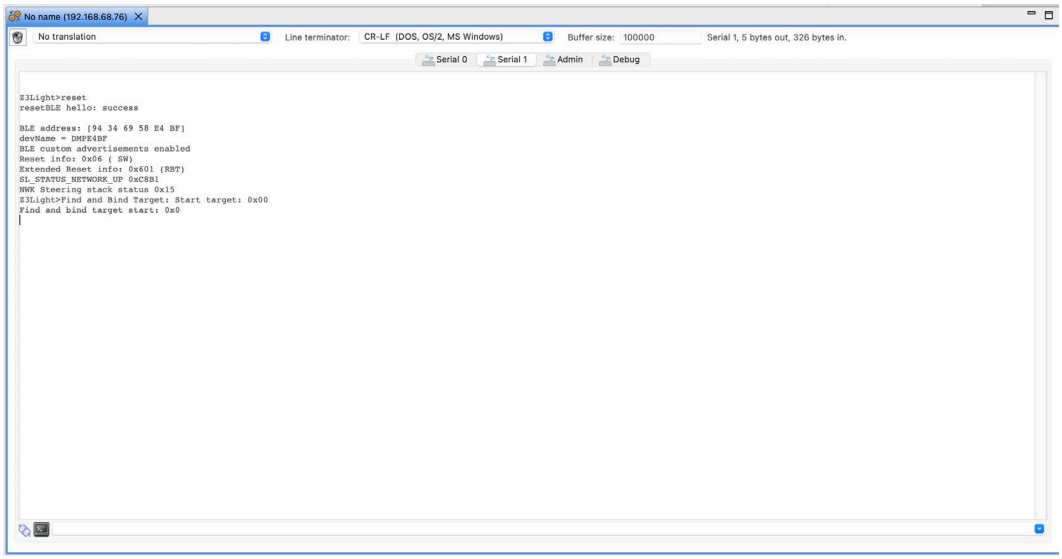
3. Save your new Z3Light project and click **Generate** in the project overview pane.



4. Build and flash the project and look for the device in the “Connected Lighting demo” screen of the EFR Connect smartphone app.



You can also see Bluetooth LE activity related printing in the Serial 1 tab of the console.



This is very basic Bluetooth functionality. To learn more about programming Bluetooth LE functionality, see Getting Started with Silicon Labs Bluetooth LE Development.

3 About the Zigbee/Bluetooth LE Examples

The Zigbee/Bluetooth LE Dynamic Multiprotocol examples demonstrate a light that can be controlled via Bluetooth LE and Zigbee. Software examples may be compiled using the sample SoC appliations in the EmberZNet SDK. The purpose of the examples is to show how to implement a dynamic multiprotocol application using the Silicon Labs EmberZNet stack.

The Dynamic Multiprotocol Demo application has these main components.

1. Mainboard User Interface (LCD, Buttons, LEDs (optional for parts with these peripherals)
2. Zigbee application (Coordinator or Sleepy End Device)
3. Bluetooth application
4. CLI interface

3.1 Mainboard User Interface

The mainboard-interface application code has three main components. These help to enhance the user experience, but are not essential to the core DMP functionality. If your demo radio board does not support LCD, a minimal version of these applications is automatically chosen and the buttons, LED and LCD components are automatically removed from the project.

3.1.1 Buttons

The DynamicMultiprotocol sample applications use two buttons on the mainboard. This functionality is provided using two instances of the **Simple Button** component and can be easily uninstalled if the mainboard does not have buttons. Button PB0 toggles the local state of the light. Button PB1 controls network operations such as form, join, and leave.

3.1.2 LED

The sample app displays the current state of the On/Off light using the two LEDs on the mainboard. This application code is provided using two instances of the **Simple LED** component.

3.1.3 LCD

The LCD enhances the overall user experience by providing helpful instructions and displaying the state of the node. This functionality is provided using the **Zigbee LCD Display** component. This component provides APIs to update the text and graphic on the LCD. These APIs are invoked from the application's Zigbee callbacks and Bluetooth event handlers.

3.2 Command Line Interface Task (CLI)

The CLI task runs as a relatively low priority task and processes commands and displays output. CLI commands post the semaphore and allow Zigbee RTOS tasks to run by invoking the function `sl_zigbee_wakeup_common_task ()` in the post command hook.

3.3 Zigbee Application

The **DynamicMultiprotocolLight** sample application is a Zigbee coordinator and **DynamicMultiprotocolLightSed** is a Zigbee sleepy end device. Both sample applications demonstrate a wireless light that can be controlled locally using a button or wirelessly using a Zigbee switch or a Bluetooth LE mobile application.

The following cluster set is supported by both the **DynamicMultiprotocolLight** and **DynamicMultiprotocolLightSed** applications:

- Basic
- Identify
- Scenes
- Groups
- On/Off
- ZLL Commissioning

The **DynamicMultiprotocolLight** example also supports Green Power Proxy Basic endpoint. Note that the examples were developed with a focus on demonstrating dynamic multiprotocol features and may not be Zigbee-certifiable.

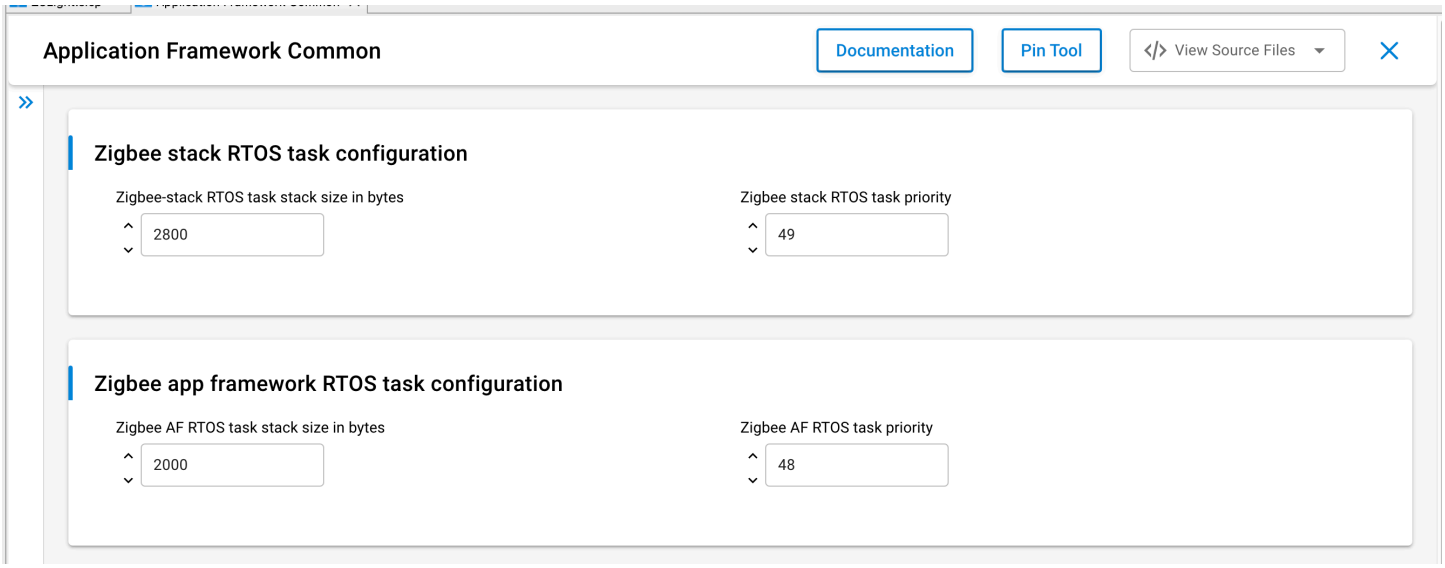
The On/Off cluster controls the LEDs and the bulb icon on the mainboard LCD to represent the state of the light.

3.3.1 Zigbee RTOS Task

The DMP sample applications utilize CMSIS-RTOS2 constructs and therefore are structured to support either Micrium OS or FreeRTOS. Micrium OS is set up as default RTOS. Free RTOS is also supported. The RTOS tasks are:

- Bluetooth link layer task (priority: 52)
- Bluetooth host stack task (priority: 51)
- Bluetooth event handler task (priority: 50)
- Zigbee stack task (priority: 49)
- Zigbee app framework task (priority: 48)
- Command Line Interface task (priority: 16)

Zigbee RTOS task-related configuration is in the Zigbee **Application Framework Common** component. Note that the Zigbee and Bluetooth task priorities must not be changed from their defaults in order to ensure that the application works as intended. Any application RTOS tasks must be lower priority than the Zigbee stack and Zigbee app framework tasks.



Custom RTOS tasks

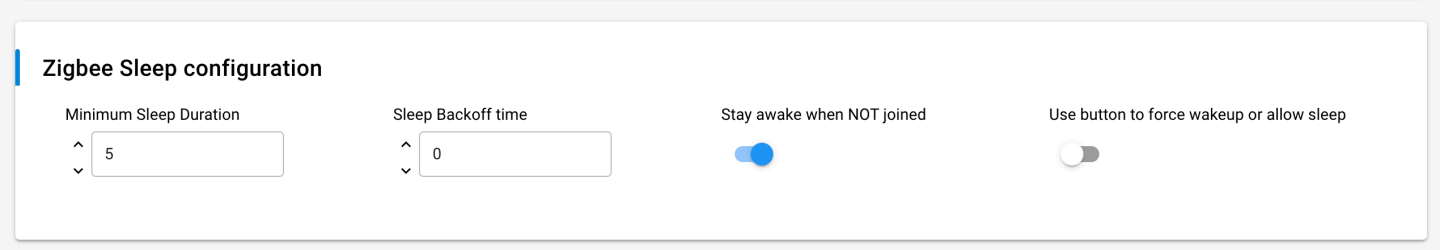
Custom tasks may be created following the example example code below. As of SiSDK 2024.6, all stack and application APIs are thread-safe.

```
static osThreadId_t task_id;
__ALIGNED(8) static uint8_t task_stack[TASK_STACK_SIZE];
__ALIGNED(4) static uint8_t task_cb[osThreadCbSize];
static osThreadAttr_t task_attr;

void create_new_task(void)
{
  task_attr.name = "Custom task";
  task_attr.stack_mem = &task_stack[0];
  task_attr.stack_size = sizeof(task_stack);
  task_attr.cb_mem = task_cb;
  task_attr.cb_size = osThreadCbSize;
  task_attr.priority = (osPriority_t)TASK_PRIORITY;
  task_attr.attr_bits = 0;
  task_attr.tz_module = 0;
```

```
  task_id = osThreadNew(task,
                        NULL,
                        &task_attr);
  assert(task_id != NULL);
}
```

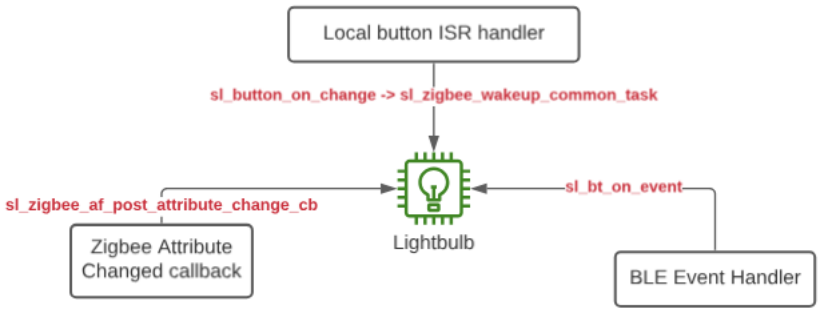
Power modes and sleep:

The Zigbee codebase utilizes power manager to provide input that enables entry and exit into Stack and application framework Several application override mechanisms control whether the microcontroller is allowed to enter sleep (EM2) or idle (EM1) modes. These flags, in combination with the time to the closest application or stack event, control how long the Zigbee RTOS tasks yield for. Zigbee app framework specific options are also configured in the **Application Framework Common** component.



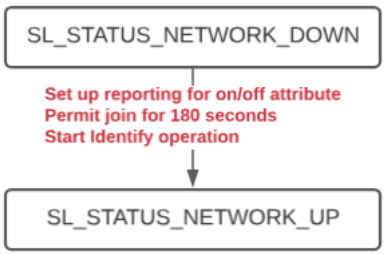
3.3.2 Application Code

On either DMP light application, once the Zigbee stack is set up to run, subsequent interactions with the stack occur via event handlers. The following figure shows the 3 ways to change the state of the light (local button press, Zigbee message from a switch, BLE message from the EFR connect app) and the handlers that fire in each of these cases.



Each enabled cluster must have a corresponding component that handles the callbacks for the cluster. Alternatively, this can be provided by a custom implementation in the project callbacks file. In addition, the Zigbee callbacks file subscribes to optional stack callbacks, such as stack status callbacks, to show the network state and perform other operations based on change of state.

Whenever the coordinator sample application starts pjoin, it starts identifying and also puts all the connected lights in identify mode. This helps the joining switch to identify all the lights present in the network. The sleepy sample application does the same on the steering status callback.



The On/Off attribute can be changed locally using the button PB0. The button_on_change ISR routine fires on change of the button state. Note that, since this routine is executed from an interrupt context, printing messages in this routine is not recommended. For this reason, once time stamps are recorded and states are set, a separate event handler is set to active to further process the button press. Since the event handler runs from the Zigbee task context, the semaphore must be posted by invoking the function `sl_zigbee_ wakeup_common_task()`.

The On/Off attribute may also be changed by receiving a Zigbee on-off toggle command from a remote device like the Z3Switch. This path follows the sl_zigbee_af_post_attribute_change_cb. Any change to the attribute will also trigger a notification over a Bluetooth LE connection, if one is open. In addition to the state, the trigger source and the EUI of the trigger source are recorded for tracking.

The On/Off attribute may also be modified using the EFR Connect mobile application. The light displays on the app as “DMPxxxx” where xxxx are the last four digits of the Bluetooth LE MAC address. The characteristic can be read and written using the mobile application. This triggers a change to the Zigbee attribute.

CAUTION for SDK 4,4 and older: The Zigbee stack APIs are not thread-safe. As such, all calls to EmberZNet functions should be made from the Zigbee task to avoid the risk of concurrency issues. To avoid the risk of shared resources, if you want to send Zigbee messages or use EmberZNet functions from a task other than the Zigbee Stack Task, you must schedule a custom event from within the non-Zigbee Stack task. In the corresponding event handler function for the custom event the Zigbee stack APIs can be used, as the event handler will be called from the Zigbee Stack Task context.

As of SiSDK 2024.6, Zigbee stack APIs are thread-safe. Application framework APIs are protected by a mutex in the CLI task context. If they are to be called from a custom RTOS task, you are expected to surround the calls with sl_zigbee_af_acquire_lock and sl_zigbee_af_release_lock. It is also important to note that this version also introduced the standardization of all function names. Please refer to Zigbee API Reference Manual for more details.

3.4 Bluetooth Application

The Bluetooth application supports following services and characteristics. These are pre-selected in the GATT editor during project generation.

Service	Characteristic
Device Information	Manufacturer Name String Model Number String Serial Number String Firmware Revision String
Generic Access	Device Name Appearance
Silabs DMP Light	Light Trigger Source

3.4.1 Silabs DMP Light Service

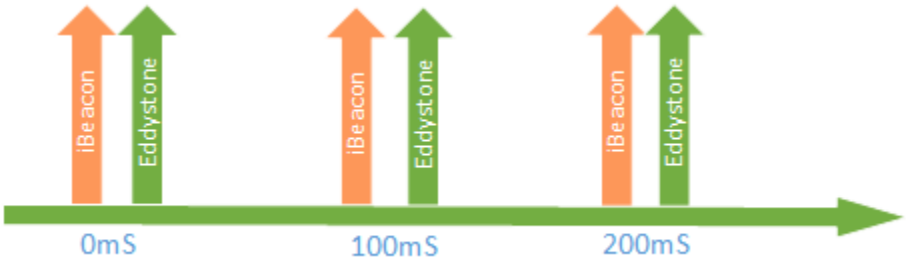
In the above table the ‘Silabs DMP Light’ is a custom service with a UUID of `bae55b96-7d19-458d-970c-50613d801bc9`. This custom UUID is used to uniquely identify the Light by the EFR Connect application.

The Service has two characteristics,

Characteristic	Data Type	Description
Light	8bit Boolean	Used to get and set the light state 1 = Light On 0 = Light Off
Trigger Source	8bit enum	Indicates the source of the Light state change command. 0 = Bluetooth 1 = Zigbee 2 = Button Press

3.4.2 Beacons

The application implements both an iBeacon as well as an Eddystone beacon. The default behavior is to transmit each beacon at 100 mS intervals.



3.4.3 Bluetooth Event Handling

The Bluetooth stack is initialized as part of the Bluetooth task. The Bluetooth task handles the Bluetooth LE link layer messaging and management. A number of events that are called in the context of the Zigbee task allow the user application to interact with the Bluetooth stack. The following diagram describes the Bluetooth-related events..

Note: Bluetooth event handling is same for both DMP demos.

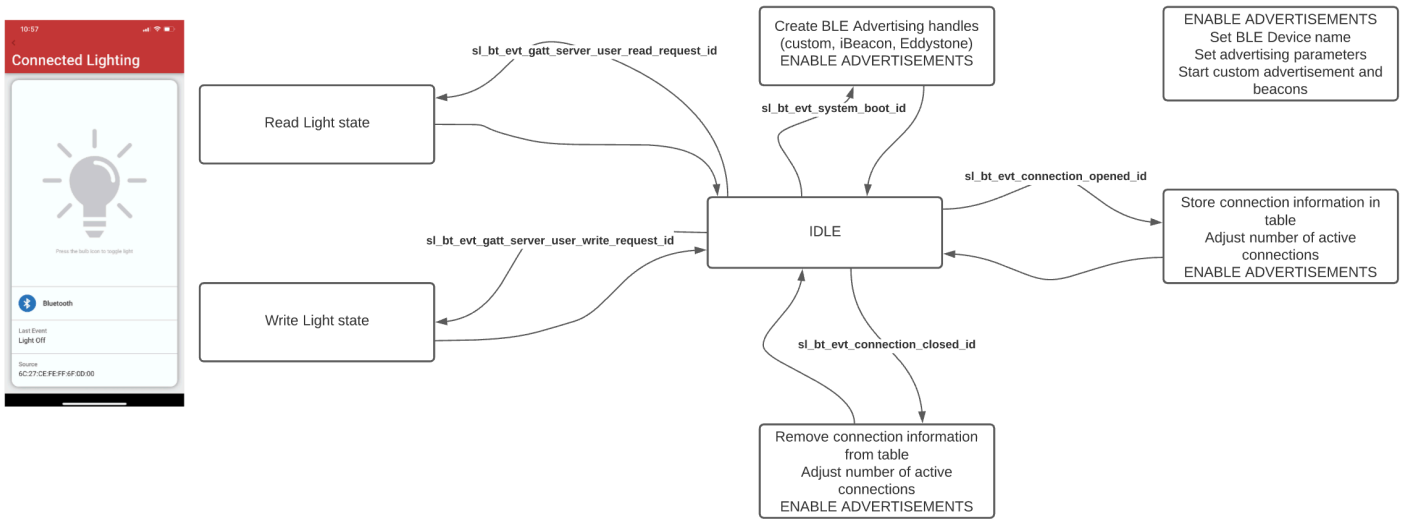
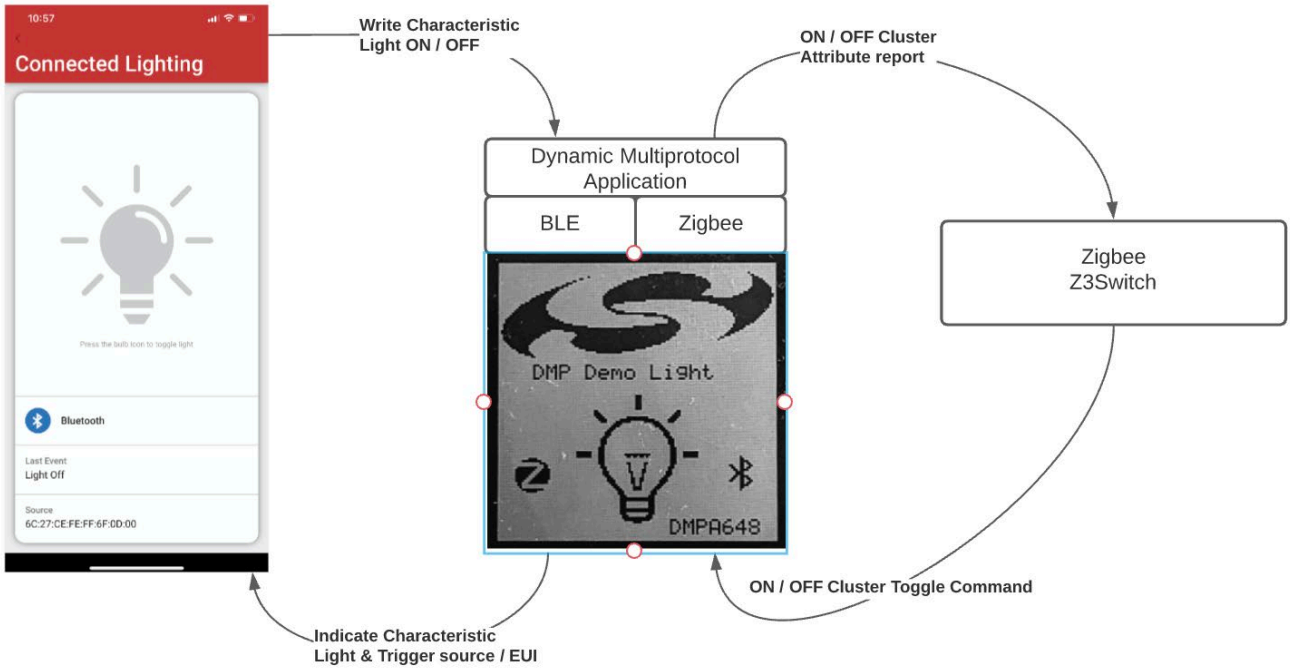


Figure 3-1. DMP Bluetooth Event Handler Definition

3.4.4 Bluetooth and Zigbee Interaction

The primary purpose of the example applications is to show Zigbee and Bluetooth working together on a device. For this purpose, when the Light receives a command to change its state through one protocol, it executes the command and sends out a notification to the other devices using the other protocol to keep everything in sync. Their interaction is the same in both examples.

Two basic operations are described below, first a write to Light characteristics from a Bluetooth connected device (shown in the following figure) and then a change in the Light state from a Zigbee device.



Write from the Bluetooth Connected Device

The application's services and characteristics are pre-selected in the GATT configurator in Simplicity Studio. On generation the characteristics are #define in the gatt_db.h. Using the #define reference, the characteristics can then be coupled to read and write Bluetooth requests. For example, the Light characteristic is reference from GATT as gatt_light_state which is then tied to an application-specific write API of writeLightState in the AppCfgGattServerUserWriteRequest in sl_bt_event_handler.c.

The application implements the Zigbee attribute write and a Bluetooth write response in the writeLightState function. Since ember functions are not thread-safe, the application posts a Zigbee event and a semaphore to wake the Zigbee task and invoke the `sl_zigbee_af_write_attribute` function.

The `sl_zigbee_af_write_attribute` function is used to write the attribute table of the Zigbee application with the value supplied by the Bluetooth connected device above. Since the on-off attribute of the on-off server cluster is a reportable attribute, it is reported to all devices setup in the binding table of the Light.

The sl_zigbee_af_post_attribute_change_cb function is then used to change the state of the LEDs and the LCD to indicate the state of the light on the WSTK main board.

Write from the Zigbee Connected Device

Any on-off client on the same network as the Light can send an on-off cluster's On, Off or Toggle command to the Light to change its state. Once such a command is received over the Zigbee interface, the Silicon Labs Zigbee framework interprets it and calls an appropriate handler to change the value of the on-off attribute of the on-off server cluster. In the example **Z3Switch** application, the on-off client sends a Toggle command to the Light, which toggles the value of the on-off attribute and triggers the sl_zigbee_af_post_attribute_change_cb. The callback is then used to change the state of the light as well as send notifications for both Trigger Source and Light characteristics to the connected Bluetooth devices and to update the LEDs and the LCD to indicate the change in the Light state. Example code for the callback can be found in the project callbacks file.

4 Document Revision History

Revision 0.4

December, 2024

- Revised to account for SiSDK 2024.12 release and associated changes

Revision 0.3

August, 2024

- Revised to account for SiSDK 2024.6 release and associated changes

Revision 0.2

March, 2023

- Added a caution on thread safety to section 3.2.2

Revision 0.1

December, 2021

- Initial release