UG305: Dynamic Multiprotocol User's Guide

This user guide provides details about implementing the Silicon Labs Dynamic Multiprotocol. Dynamic multiprotocol time-slices the radio and rapidly changes configurations to enable different wireless protocols to operate reliably at the same time. Dynamic Multiprotocol operation is described using Bluetooth with EmberZNet and with RAIL.

KEY POINTS

- Dynamic Multiprotocol architecture
- About the Radio Scheduler
- Radio Scheduler examples
- Working with the Dynamic Multiprotocol examples
- About the Zigbee/Bluetooth LE Dynamic Multiprotocol Demonstration.

1 Introduction

This document describes how Silicon Labs software is designed to be used by multiple protocols on a single wireless chip. Dynamic multiprotocol time-slices the radio and rapidly changes configurations to enable different wireless protocols to operate reliably at the same time.

Dynamic Multiprotocol is a different solution than Switched Multiprotocol, which allows for switching between radio configurations by bootloading separate firmware for each protocol. For information on Switched Multiprotocol, see *UG267: Switched Multiprotocol User's Guide*. For more information about multiprotocol implementations in general, see *UG103.16: Multiprotocol Fundamentals*.

1.1 Requirements

1.1.1 Zigbee/Bluetooth

- EmberZNet SDK version 6.0.0 or higher
- Bluetooth SDK version 2.6.0 or higher
- Micrium OS-5 kernel.
- An EFR32 chip with at least 512 kB of flash (required to run all the necessary software components)
- IAR Embedded Workbench for ARM (IAR-EWARM) 7.80

1.1.2 RAIL/Bluetooth

- Bluetooth SDK version 2.7.0 or higher
- RAIL 2.1 or later
- Micrium OS-5 kernel
- An EFR32 chip with at least 512 kB of flash (required to run all the necessary software components)
- IAR Embedded Workbench for ARM (IAR-EWARM) 7.80 (required for the RAIL Switch application)

1.2 Terminology

The following lists some of the terminology specific to the dynamic multiprotocol implementation and the radio scheduler.

Radio Operation: A specific action to be scheduled. A radio operation has both a radio configuration and a priority. Each stack can request that the radio scheduler perform three radio operations:

- Background receive: Continuous receive, intended to be interrupted by other operations
- Scheduled receive: Receive at a future time with a minimum duration
- Scheduled transmit: Transmit at a future time with a minimum duration

**Radio Config**: Determines the state of the hardware that must be used to perform a radio operation.

Priority: Each operation from each stack has a default priority. An application can change default priorities.

RAIL (Radio Abstraction Interface Layer) provides an intuitive, easily-customizable radio interface layer and API, which supports proprietary or standards-based wireless protocols. RAIL has a new interface and a new component, the radio scheduler, which enables dynamic multiprotocol operation.

RTOS (Real Time Operating System) Kernel: The part of the operating system that is responsible for task management, and inter-task communication and synchronization. This implementation uses the Micrium OS-5 kernel.

2   Architecture

Dynamic Multiprotocol makes use of the EFR32 hardware and the RAIL software as its building blocks. The following diagram illustrates the general structure of the software modules.
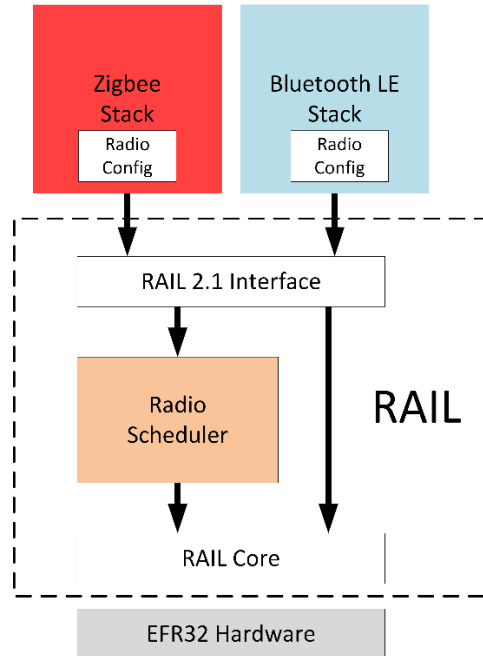
Beginning with version 2.0, RAIL supports passing a radio configuration to the RAIL API calls. This configuration describes various PHY parameters that are used by the stack.

The Radio scheduler is a software library that intelligently answers requests by the stacks to perform radio operations to maximize reliability and minimize latency.

The RAIL core configures the EFR32 hardware in response to instructions from the radio scheduler.

2.1   Wireless Protocol Support

Different wireless protocols have different characteristics that have been leveraged with the design of Dynamic Multiprotocol. For example, Bluetooth Low Energy is very strict and predictable in its schedule of radio tasks; advertisements and Connection intervals occur at set times. In contrast, a Zigbee network running on IEEE 802.15.4 is more flexible in the timing of many message events; CSMA (carrier sense multiple access) in IEEE 802.15.4 adds random backoffs so that delays for Zigbee events are on the order of milliseconds. This allows IEEE 802.15.4 messages to be sent around the Bluetooth Low Energy events and still be reliably received.

2.2 Micrium OS

Each stack runs a separate RTOS task utilizing the Micrium OS-5 kernel to provide the task switching. A task is equivalent to a thread in other operating systems. The tasks coordinate using various IPC (interprocess communication) mechanisms (message queues and semaphores) to pass information back and forth.
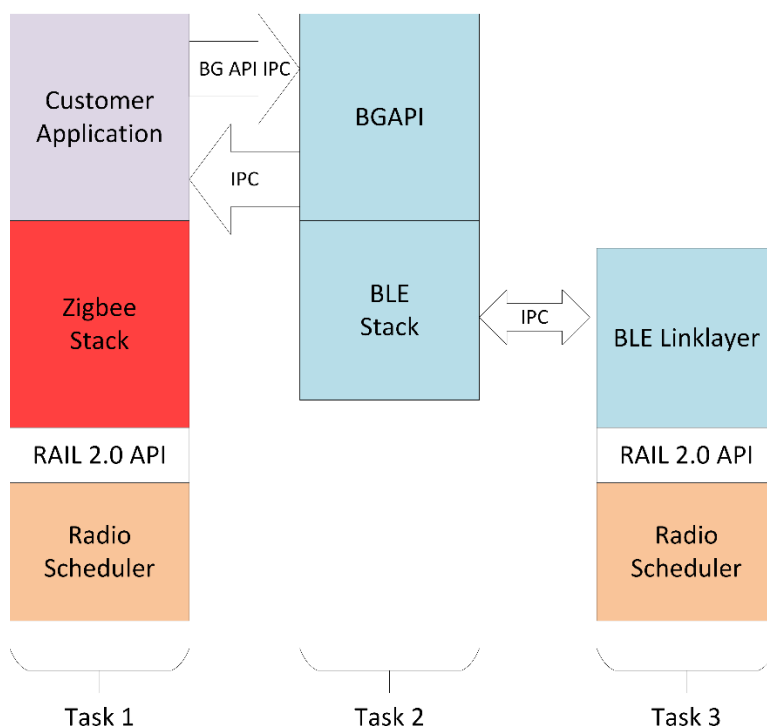
2.3 Single Firmware Image

Dynamic Multiprotocol allows a software developer to generate a single monolithic binary that is loaded onto an EFR32. Software up-dates are done by upgrading the entire binary. This is accomplished using the Gecko Bootloader, the details of which can be found in *UG266: Silicon Labs Gecko Bootloader User's Guide.*

2.4 Independent Stack Operation

The Silicon Labs stacks still operate independently of one another in a Dynamic Multiprotocol situation. Certain long-lived radio opera-tions will have an impact on another protocol's latency and compliant operation. It is up to the application to determine any special con-siderations for these events. See section <u>Radio Scheduler</u> for more information.

3   About the Radio Scheduler

The Radio Scheduler is designed to allow for radio operations that can be scheduled and prioritized. Different radio operations in each protocol may be more or less important, or more or less time sensitive, depending on the situation. The scheduler can take those into account when making decisions about conflicts and how to adjudicate them.

Most radio scheduler functions are handled automatically by underlying stack and RAIL code. The application developer only needs to use the stack through its normal API.

At a high level, the stack sends a radio operation (for example a scheduled receive or transmit). The radio operations are queued and then serviced at a future time based upon their parameters. When it is time to start the radio operation the scheduler examines whether or not a competing event exists and whether or not the operation can be delayed. If the scheduler cannot run the event it returns the result to the higher layer, which may retry with new parameters.

Once the radio operation has started, the corresponding stack can send the scheduler additional operations based on the results of the previous operation (for example waiting for an ACK). At the end of each operation or sequence of operations the stack must yield use of the radio.

3.1   Radio Operation

Each event in the scheduler is broken up into elements called Radio Operations, which are associated with a radio config and a priority.

Every operation has a priority and is interrupted if the scheduler receives a higher priority operation that overlaps in time. Lower priority radio operations that cannot be run based on their schedule parameters will fail, and it is up to the stack to retry them. Once the scheduler actively runs a radio operation from the stack, the stack can continue to send additional radio operations until it voluntarily yields, or until the scheduler receives a higher priority radio operation and preempts it.

Each stack can ask the Radio Scheduler to perform one of three operations:

1. Background Receive
2. Scheduled Receive
3. Scheduled Transmit

Each operation has the following parameters:

Parameter	Description
Start Time	An indication at what point in the future this radio operation will run. This could be "run right now" or some value in microseconds in the future.
Priority	A number that indicates the relative priority of the operation. Bluetooth LE radio operations are almost always higher priority than Zigbee operations.
Slip Time	An amount of time that the event can be delayed beyond its start time and still be acceptable to the stack. This may be 0, in which case the event cannot be slipped.
Transaction Time	The approximate amount of time that it takes to complete the transaction. Transmit events usually have a much more well-defined transaction time, while receive events are often unknown. This is used to help the radio scheduler determine whether an event can be run.

The stack defines these various parameters appropriate to the operation being executed. For example, Bluetooth connection events are often scheduled in the future and have no allowed slip, whereas Zigbee transmit events can often be delayed a small amount and start later.

3.1.1   Background Receive

This is a continuous receive mode that is intended to be interrupted by other operations. If Background Receive is higher priority than other operations, those radio operations will not be scheduled and will not run. It is up to the stacks or application to change the priority or voluntarily yield.

3.1.2 Scheduled Receive

This is a receive at a future time with a minimum duration. The radio scheduler will take into consideration the radio switching time in deciding whether or not the operation will be scheduled. If it cannot be scheduled then the scheduler sends a fail event to the calling stack. The radio operation is automatically extended until the stack voluntarily yields, or the scheduler receives a higher priority operation and interrupts it. Extending the receive allows the stack to continue a radio operation based on the requirements of the higher level protocol, for example transmission of a response based on the received data.

Scheduled transmit and scheduled receive are very similar. The main difference is the initial operation that is performed when the scheduler starts the radio operation.

3.1.3 Scheduled Transmit

This is a transmit at a future time with a minimum duration. This minimum duration can include expected follow-on events, for example an ACK to an IEEE 802.15.4 transmit. However, the minimum time for this operation does not have to include unexpected events that may extend the time beyond the minimum duration, for example backoffs due to CCA failures in IEEE 802.15.4.

Scheduled transmit and scheduled receive are very similar. The main difference is the initial operation that is performed when the scheduler starts the radio operation.

3.2 Radio Config

Each radio operation is associated with a predefined radio config that determines the state of the hardware that must be used to perform the operation. The Radio Configs keep track of the stack's current state so that future radio tasks will use the same radio parameters. Radio Configs may be active or dormant. If the stack changes an active Radio Config then RAIL makes an immediate change to the hardware configuration as well, for example changing a channel. If the radio config is not currently active then the next scheduled radio operation will use the new radio config.

3.3 Yield

Once a sequence of radio operations is actively being run, the stack may continue to add operations extending the initial task until the stack has nothing more to do for the particular message exchange. A stack must voluntarily yield unless it is performing a background receive. If a stack does not yield then it will continue to extend its radio task, and lower priority radio tasks will then trigger a failure back to the corresponding stack that requested that radio task. A higher priority task cannot interrupt a lower priority radio operation currently running that has not yielded.

3.4 Default Priorities

The stacks each have a default set of priorities based on Silicon Labs' analysis of how best to cooperate to maximize the duty cycle and avoid dropped connections. The priorities are as follows, from highest to lowest:

1. Bluetooth LE Scheduled Transmit
2. Bluetooth LE Scheduled Receive
3. Other protocol Scheduled Transmit
4. Other protocol Background Receive

These priorities may be overridden or changed by the application. It is up to the application to decide under what circumstances to change them.

3.5 Slip Time

Every radio operation must have a "slip time", or maximum start time, meaning the maximum time in the future when the task can be started if it cannot begin at the requested start time. This allows for the scheduler to work around higher priority events that are occurring at the same time, or higher priority events that extend beyond their expected duration. The protocol generally dictates what the slip time can be, but the radio scheduler is capable of handling this on a per-task basis, allowing a stack to slip some events but not others.

In general, IEEE 802.15.4 has longer slip time and Bluetooth LE has a minimal slip time.

3.6   Interrupting a Radio Operation

A scheduled radio operation may be interrupted if a higher priority operation conflicts with it. This could occur in the following two circumstances:

1. A scheduled radio operation takes longer than expected and the corresponding stack does not yield before the higher priority radio operation must start.
2. A higher priority radio operation has just been scheduled to occur in the future and conflicts with a lower priority operation already scheduled.

3.7   Long Lived Radio Operations

Certain long lived Radio Operations can have an outsized impact on the correct operation of the product. The application may need to coordinate these operations between the protocols. If the application does not then the radio scheduler priorities will take precedence. For example, an IEEE 802.15.4 energy scan can require that the radio stay on to gather sufficient energy readings. If the application does not properly coordinate the operations, the scan could be interrupted prematurely due to a higher priority Bluetooth operation.

4   Radio Scheduler Examples

All examples use Bluetooth LE and Zigbee, but the principles apply to other combinations.

The scheduler starts out by having a low priority Zigbee background receive operation. This represents an always-on router that may need to receive IEEE 802.15.4 packets at unknown times. A Bluetooth LE connection is also active and requires the stack to be ready to receive every 30 ms. The Bluetooth LE stack may schedule this well in advance due to the connection's predictable nature.

4.1   Priority Scheduling

This provides a basic example of adjudicating priorities of the different radio tasks.
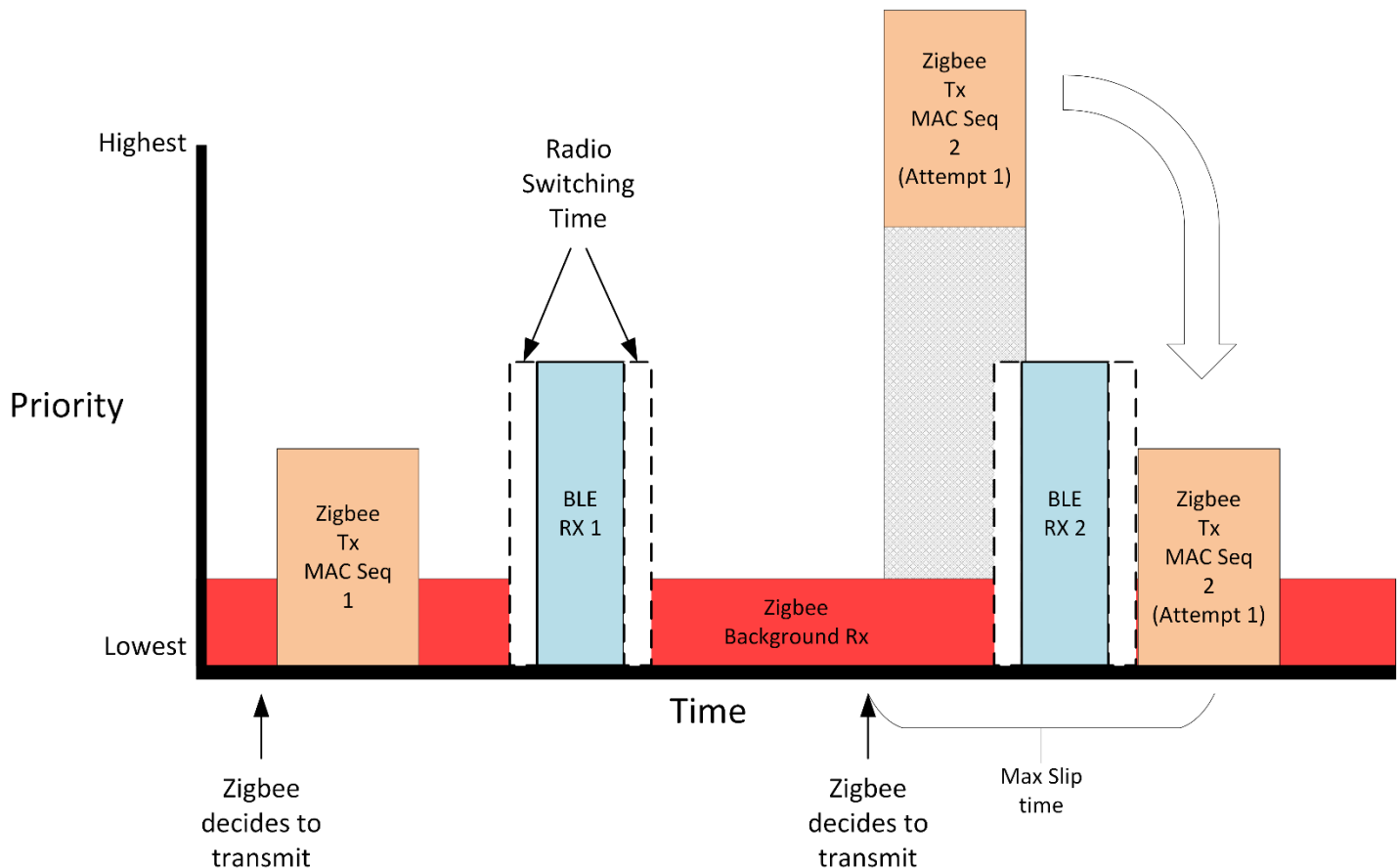
The Zigbee stack decides that it needs to send a packet. It may do this as an on-demand event, meaning the stack decides that it wants to send a packet *now* without informing the scheduler well in advance. This is in contrast to how Bluetooth LE operates, where the scheduled operations are known reasonably far in advance. The scheduler evaluates that it is possible to perform the Zigbee TX 1 radio operation and still service the higher priority Bluetooth LE reception event in the future. So the scheduler allows the transmit event to occur. The Zigbee stack performs all the pieces of this transmit operation (waiting for a MAC ack), and then voluntarily yields. The estimated transaction time of the Zigbee transmit radio operation does **NOT** include retries.

In this example, Bluetooth LE is *already* scheduled to receive in the future and the Zigbee stack wants to transmit. For the first Zigbee TX 1 radio operation there is enough time before the Bluetooth LE RX 1 radio operation so the scheduler allows the stack to perform the operation. Later, when the Zigbee stack tries to schedule Zigbee TX 2 the scheduler determines there is not enough time before the high priority Bluetooth LE RX 2 event. However, the Zigbee stack has indicated that this action may slip its start time. The radio scheduler determines that given the expected duration of the Bluetooth LE radio operation the Zigbee operation can start after that event and still be within the slip time indicated by the Zigbee stack.

If all goes as expected, the Zigbee transmit operation will have its **first** attempt occur without any failures due to scheduling.

4.2   Priority Interruption Example

This example illustrates a higher priority operation interrupting a lower priority one.
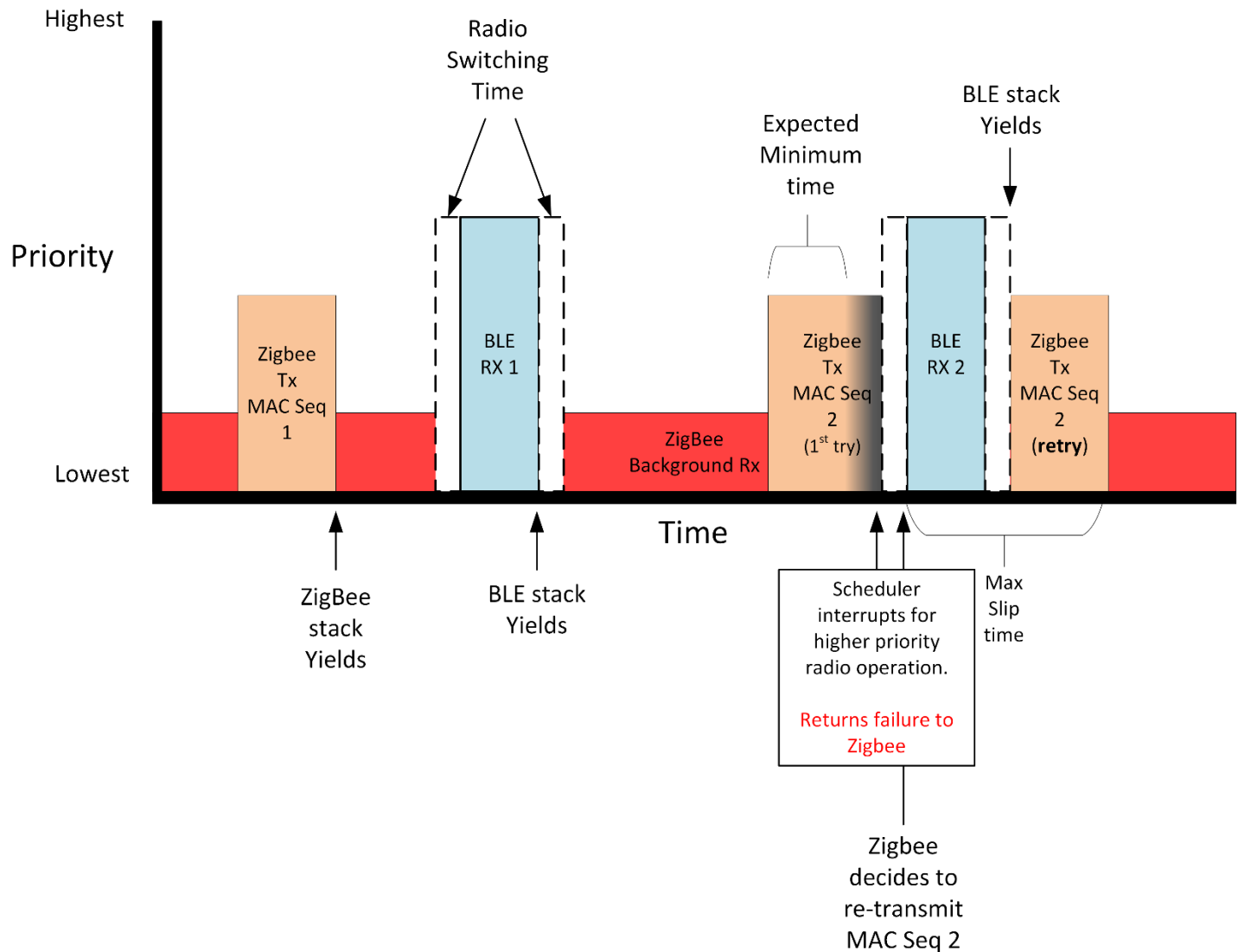
This example starts in the same way as the previous example. Zigbee and Bluetooth LE both have a radio operation that is scheduled without any collision.

Later, the Zigbee stack decides it wants to send another packet for the Zigbee TX 2 event. The scheduler determines that it *should* be possible to schedule this event and service the Bluetooth LE RX 2 event later, based on the minimum time that the Zigbee TX 2 event must take. However, the Zigbee TX 2 event takes longer than expected due to a long random backoff and does not yield in time. This causes the event to collide with a higher priority radio operation, and so the Radio Scheduler interrupts the Zigbee event and returns a failure to the higher level stack. The Bluetooth LE event occurs normally and when it is complete it voluntarily yields to any lower priority operations.

Upon receiving the failure from the radio scheduler the Zigbee stack immediately attempts to retry the MAC message. It schedules the operation and includes a slip time. At this point the Bluetooth LE stack has priority over the radio and thus the operation cannot be started yet, but the scheduler accepts the new radio operation. The Bluetooth LE stack completes its scheduled receive and yields the radio. The scheduler then triggers the Zigbee transmit operation to occur because it is still within the slip time of the initial start operation. After the transmit completes the scheduler returns to the background receive operation.

4.3   Higher Priority Operation that is Extended

This example shows what happens when a higher priority operation takes longer than originally anticipated and causes a lower priority operation to miss its opportunity.
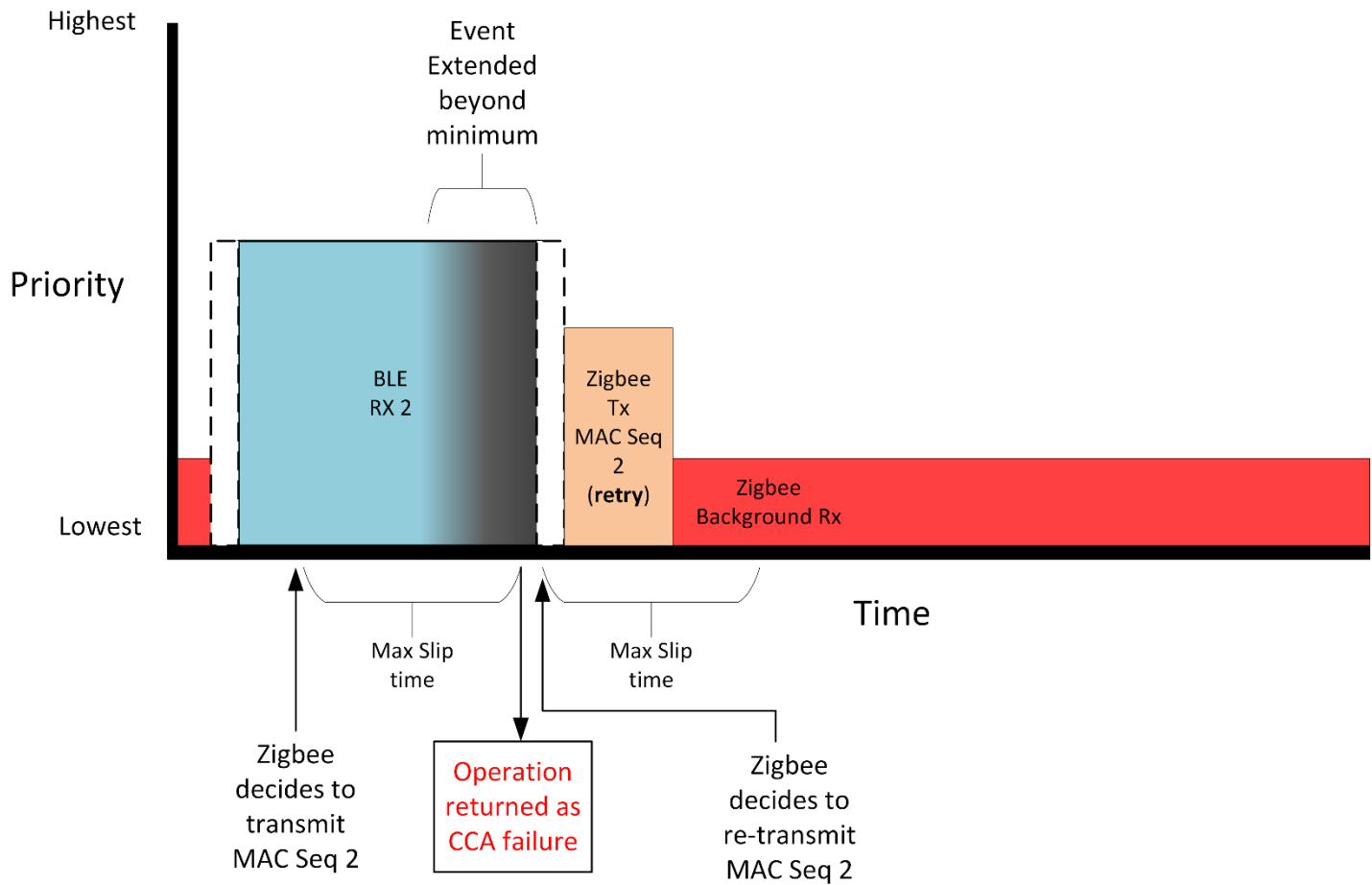
In this case, Bluetooth LE has a scheduled receive that is currently taking place. Zigbee decides to send a packet but it cannot be run right now. The scheduler accepts the operation under the assumption that the Bluetooth LE event will complete before the end of the slip time of the Zigbee event. However, the Bluetooth LE event extends longer due to the fact that additional packets are sent between the devices. The Bluetooth LE operation has priority so the Zigbee operation eventually runs out of slip. An error is returned to the stack. Zigbee decides to re-transmit the packet. Again, the Zigbee stack indicates the operation should start now but may slip into the future. The scheduler is in the middle of changing the radio config so it cannot begin the operation immediately. Instead, it slips the radio operation start time a small amount and then executes the operation.

4.4   Higher Priority operation without Interruption

In this example the radio scheduler is running on a node acting as a Bluetooth LE slave and that node has a number of connections to different masters. It also has a periodic advertising beacon that is transmitted. The following figure shows a case where these events are occurring virtually back-to-back and do not allow for enough time to switch back to the Zigbee radio config. Therefore it will create a period where the Zigbee stack is unable to transmit even with the slip time.
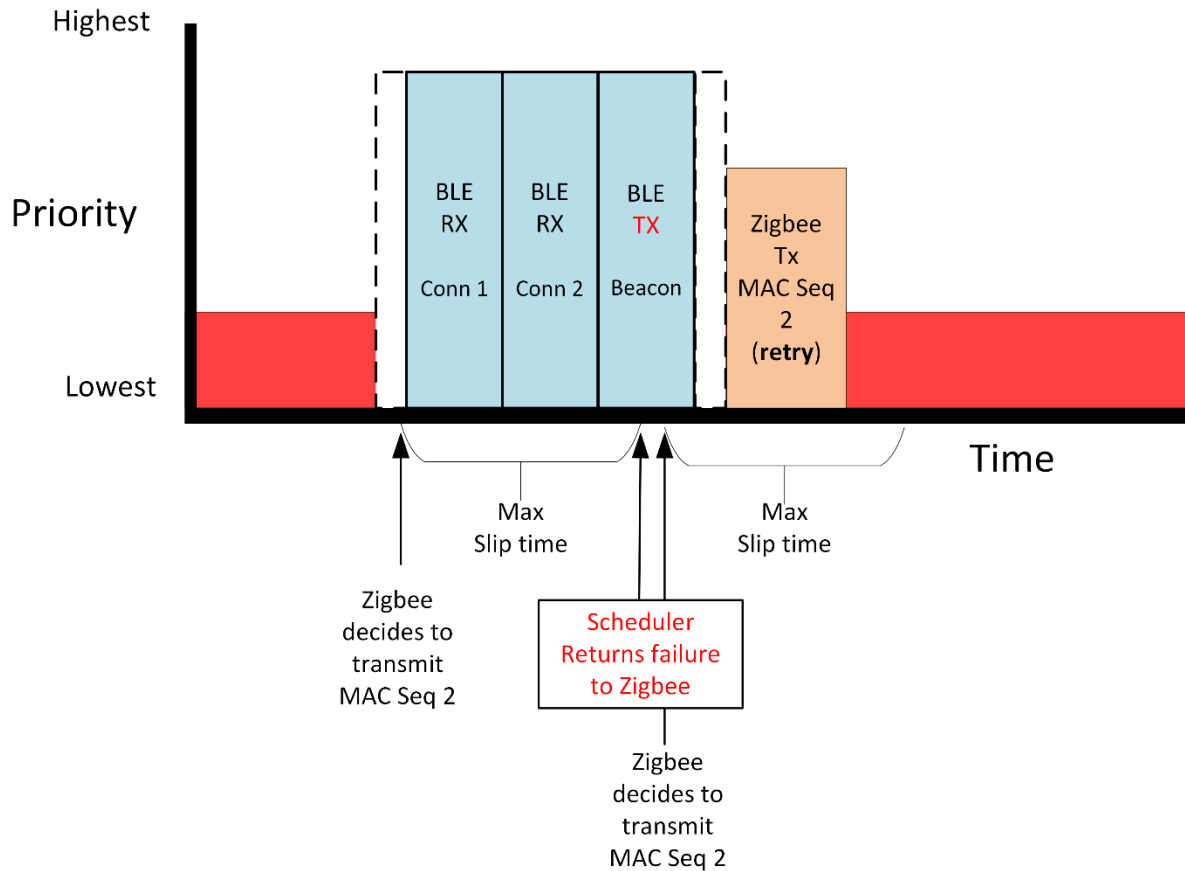
Zigbee asks the scheduler to schedule a transmit radio operation. Even though the scheduler knows that the event will fail due to scheduled higher priority operations, it still accepts the scheduled event. This is done for two reasons. First, circumstances may change and the event can be executed. Second, the stack sitting on top of the radio scheduler may try to retry the action. If the result of the failed scheduling was returned immediately then the stack's attempt to retry would be unlikely to succeed since no time has passed. Instead, by queuing the event and returning the failure *after* the slip time has expired, a retry (with its own slip time) has a better chance of success as the set of upcoming radio operations will be different.

4.5 Receive When a Higher Priority Operation is Running

This example illustrates what happens when Bluetooth LE is active and a lower priority operation will be receiving data.
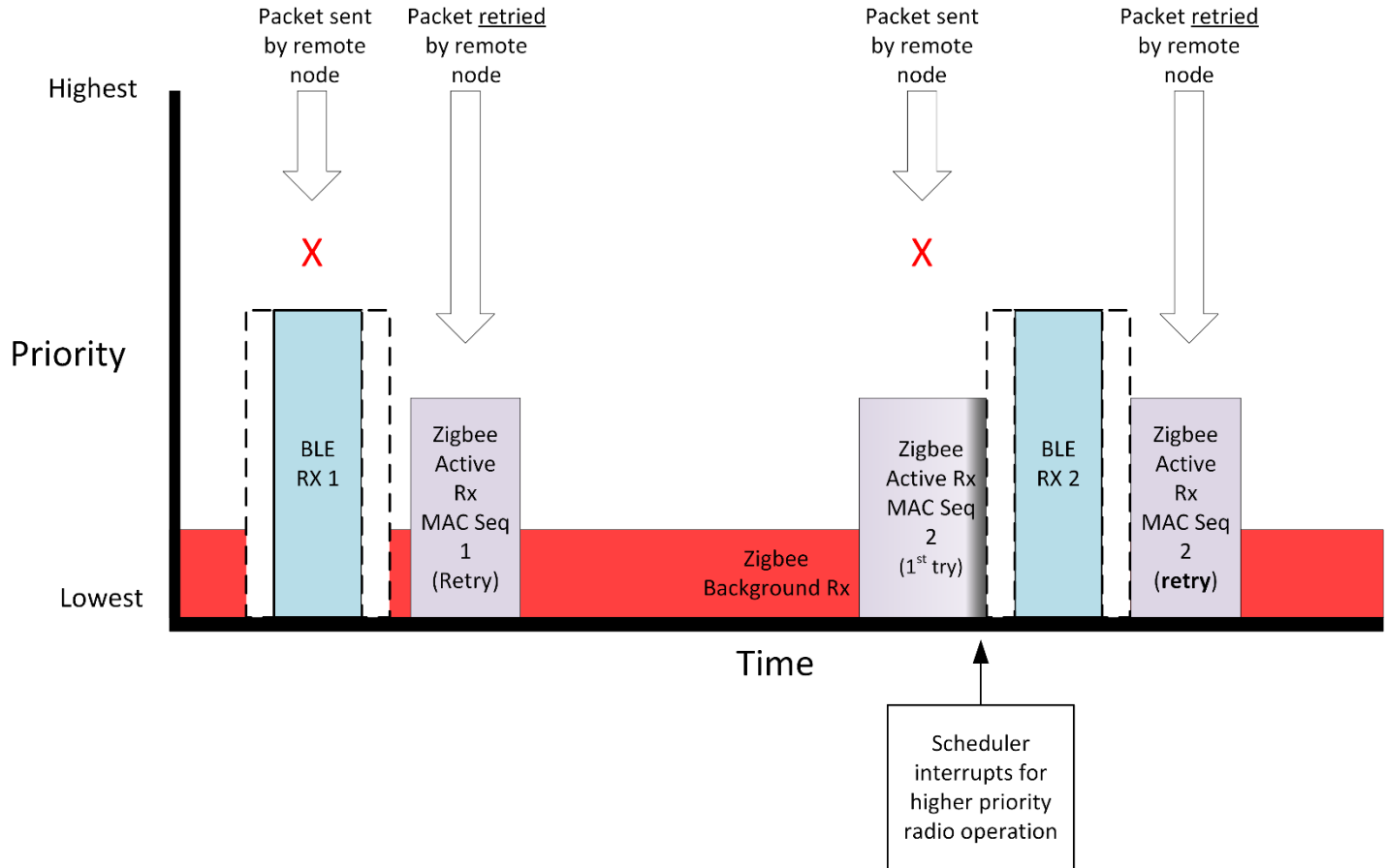
In the first case, when an IEEE 802.15.4 message is sent and the Bluetooth LE stack is utilizing the radio for an active receive the Zigbee stack will not be online to receive the message. However, the Zigbee sender of the message will retry in most cases and with backoffs and other timing alterations is not going to conflict with another higher priority scheduled Bluetooth receive events unlikely to collide. The Zigbee message is received successfully.

The second case shows that, in the case of an active receive, the Zigbee stack may still be interrupted and not receive (or ACK) the message. Successful communication relies on retries at the MAC or higher layer to send this message again and verify the Dynamic Multiprotocol device receives the message.

While there may be considerations for whether or not active receive should be interrupted, it is difficult for the scheduler to make that determination. In general the robustness of the protocols should allow for messages to be successfully received even with interruptions.

5   Working with the Dynamic Multiprotocol Examples

5.1   Zigbee/Bluetooth Example

5.1.1   Application Generation

To work with Zigbee/Bluetooth dynamic multiprotocol applications you must install both the EmberZNet SDK version 6.0.0.0 or higher, and the Bluetooth SDK version 2.6.0 or higher. The Micrium kernel is installed along with the EmberZNet SDK. IAR Embedded Work-bench for ARM (IAR-EWARM) 7.80 must be installed and used as your compiler. See *QSG106: Getting Started with EmberZNet PRO* for information on installing the SDKs and IAR-EWARM.

Dynamic multiprotocol applications are generated, built, and uploaded in the same way as other applications. If you are not familiar with these procedures, see *QSG106: Getting Started with EmberZNet PRO* for details. The following summary procedure uses the **DynamicMultiprotocolDemoLightSoC** example application.

1. In Simplicity Studio, start a new project.
2. In the new project dialogs, select ZCL Application Framework V2, the EmberZNet SoC stack, then either check Start with a blank application or, as in this procedure, select the DynamicMultiprotocolDemoLightSoC example.
3. Name the project. In project setup, if you have both compilers installed, uncheck GNU ARM, then click **Finish**.
4. If you forget to uncheck GNU ARM and your project General tab still shows it as a compiler, change to IAR EWARM.
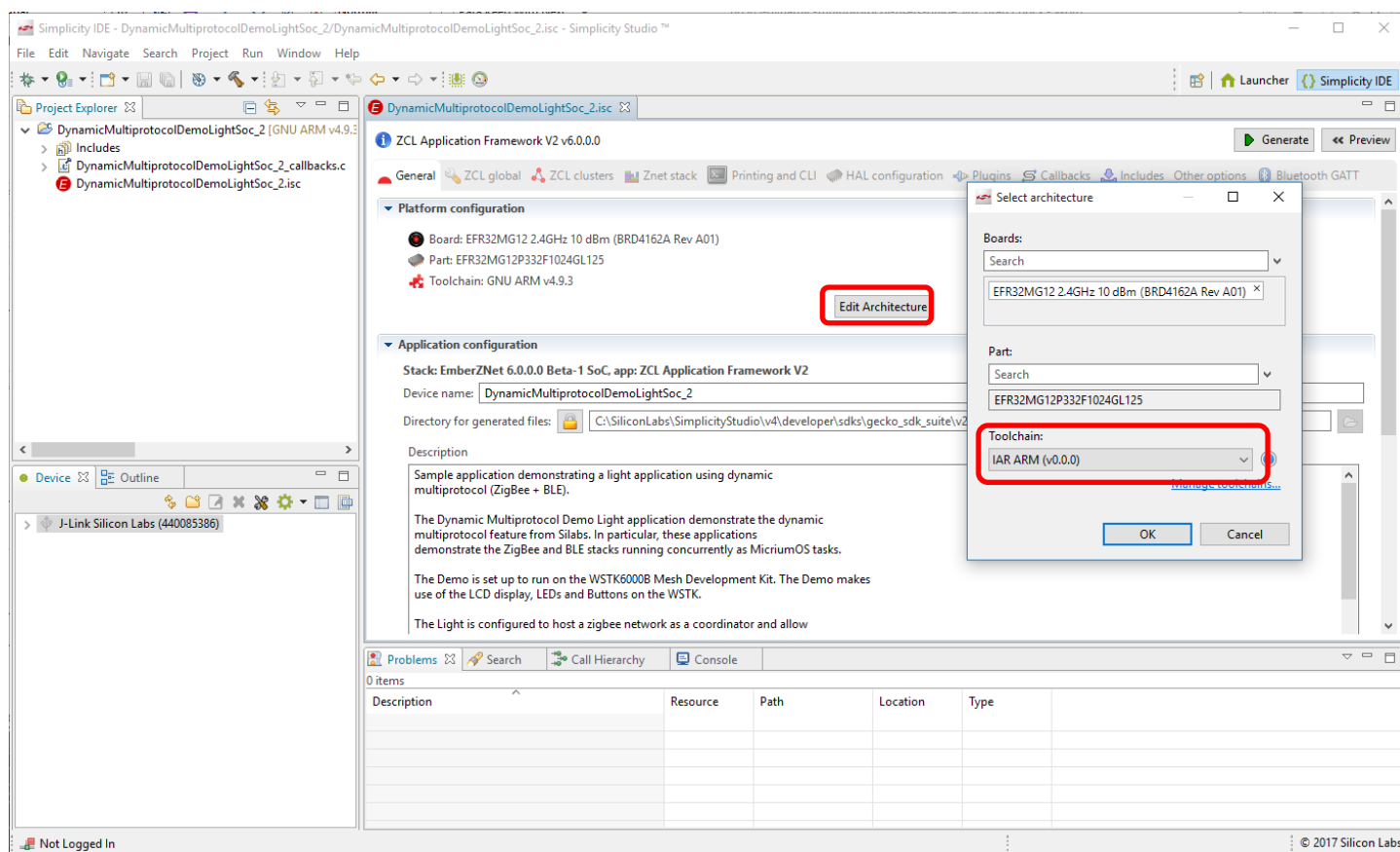
5. Click **Generate** to generate project files.
6. Click to build the application image.

7. Note the part number for your device and the directory for generated files.
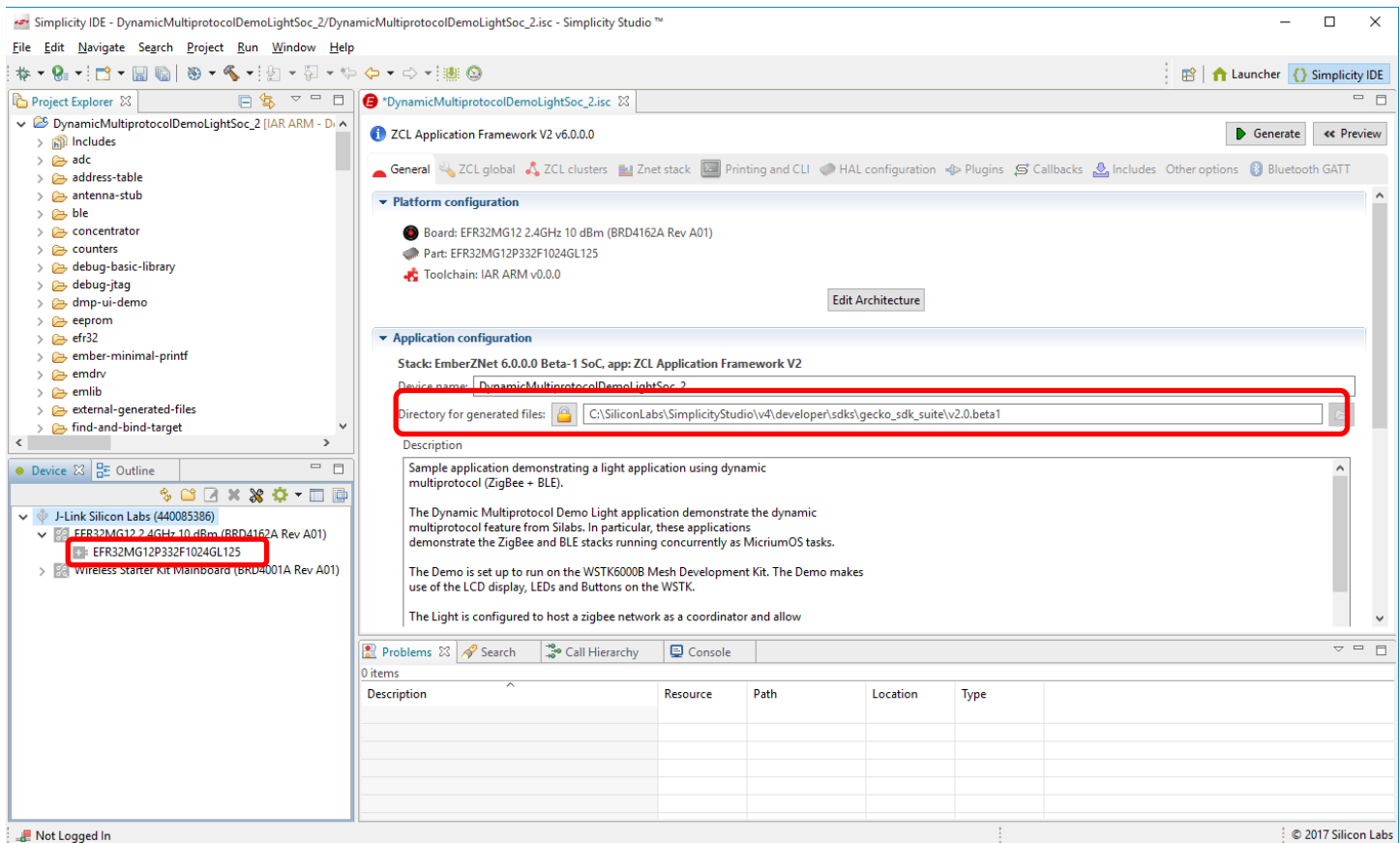
8. Right-click the target J-Link under Devices, and select Upload Application.

9. Browse to `<folder on General tab>\app\builder\<projectName>\IAR ARM - <qualifier>\<project name>` and select the .gbl file.

10. Silicon Labs strongly recommends that, if you have not already loaded a bootloader onto your device, you do so now. Check **Bootloader image**, then browse to the following folder and select the .s37 file corresponding to the radio board part number, for example 'bootloader-storage-spiflash-efr32mg12p432f1024gl125.s37'.:

```
C:\SiliconLabs\SimplicityStudio\v4\developer\sdks\gecko_sdk_suite\<version>\platform\bootloader\s
ample-apps\bootloader-storage-spiflash\
```

11. When both images are selected, the dialog should resemble the following figure. Click **OK**.
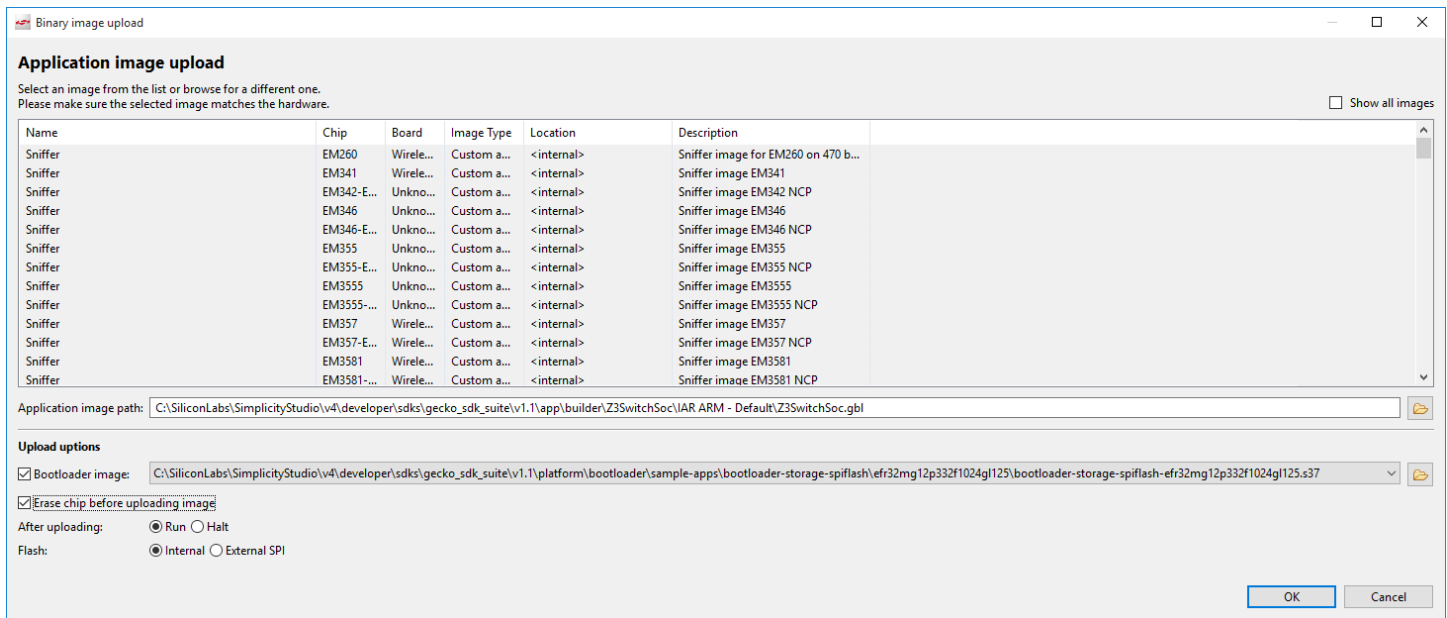
12. Application load success indicators are code-dependent. With the **DynamicMultiprotocolDemoLightSoC** example, the LCD should display the following before changing over to the light bulb display:

5.1.2 EmberZNet Dynamic Multiprotocol Plugin Configurations

This section describes the configurations in a Zigbee application to implement Zigbee/Bluetooth LE Dynamic Multiprotocol functionality. The instructions assume you have started with a blank application.

In the Stack Libraries group:

- Enable **Bluetooth stack**.
  - Max number of Bluetooth connections: The maximum simultaneous connections in Bluetooth. The higher the number the more RAM is required for the application.
- Disable **Zigbee PRO Stack Library**
- Enable **Zigbee PRO BC table MbedTls Stack Library**. MbedTLS is required to arbitrate use of the encryption block, which both Zigbee and Bluetooth access for security.

In the Utility group:

- Enable **Micrium RTOS**. The stack then runs as a Micrium OS task.
  - CPU Usage Tracking: Checked by default. Uncheck to save RAM. .
  - Zigbee stack size: Defaults to 1000
  - Up to three **Application Tasks**. These are custom Micrium tasks other than the stack. Priority is the priority that Micrium will use to execute the tasks. The Zigbee stack and Bluetooth stack have priorities higher than 7 but they will yield to the application task. This is automatically managed by the Micrium kernel.
- Disable **Simple Main**.
- Enable **mbed TLS.**

- Enable **Idle/Sleep**. Dynamic Multiprotocol implementations use that logic to manage when the stack yields the radio.

In the RAIL group:

- Enable **RAIL Library Multiprotocol**, which provides RAIL library functionality with multiprotocol support.
- Disable **RAIL Library**

Note: The Multiprotocol group contains plugins used for a Switched Multiprotocol application. They are not enabled for a dynamic multiprotocol implementation.

5.2 RAIL/Bluetooth Example

To work with RAIL/Bluetooth dynamic multiprotocol applications you must install both the Flex SDK version 2.1.0 or higher, and the Bluetooth SDK version 2.7.0 or higher. The Micrium kernel is installed along with the Bluetooth SDK. You can use IAR Embedded Workbench for ARM (IAR-EWARM) 7.80 (or higher). The RAIL Switch and Bluetooth/RAIL Light Dynamic multiprotocol applications are generated, built, and uploaded in the same way as other applications in their SDKs.

- To see details about installing Simplicity Studio and the Flex SDK and building an example application, see *QSG138: Getting Started with the Silicon Labs Flex SDK for the Wireless Gecko (EFR32™) Portfolio*.
- To see details about installing Simplicity Studio and the Bluetooth SDK and building an example application, see *QSG139: Bluetooth Development with Simplicity Studio*.

Note: In a demonstration configuration with multiple RAIL/Bluetooth dynamical protocol light devices and a single switch device, unpredictable behavior may occur. We recommend testing with a single light device and a single switch device.

The following summary procedures are provided for your convenience.

5.2.1 Building the RAIL:Switch Application

1. In Simplicity Studio, start a new project.
2. In the Applications dialog, select Silicon Labs Flex SDK, and click **Next**.
3. In the Select Applications dialog, select **RAIL: Switch** and click **Next.**
4. In the Project Configuration dialog, name the project and click **Next**.
5. In project setup, make sure that the correct board and part are shown. Click **Finish**.
6. If you get an auto-upgrade notice. click **OK**.
7. Click **Generate** to generate project files.
8. Either automatically compile and flash using the debug button, or manually compile and then load.

Application load success indicators are code-dependent. With the RAIL: Switch example, the LCD displays a short menu before changing over to the light bulb display.


5.2.2 Building the Bluetooth Light Application

The Bluetooth Light application requires the Gecko Bootloader be loaded on the device. The Gecko Bootloader is loaded when you load the precompiled **SOC - Light - RAIL - DMP** demonstration as described in *QSG155: Using the Silicon Labs Dynamic Multiprotocol Demonstrations*. Alternatively you can build and load your own Gecko Bootloader combined image (called **<projectname>-combined.s37)**, as described in chapter 4 of *UG266: Silicon Labs Gecko Bootloader User's Guide*.

1. In Simplicity Studio, start a new project.

2. In the Applications dialog, select Bluetooth SDK, and click **Next**.

3. In the Select Applications dialog, select **SOC - Light - RAIL - DMP** and click **Next.**

4. In the Project Configuration dialog, name the project and click **Next.**

5. In project setup, make sure that the correct board and part are shown and, if you have both IAR and GCC configured, select one. The active toolchain will be used in the project. Click **Finish**.

6. You do not have to generate the files unless you have modified the GATT database.

7. Automatically compile and flash using the debug button.

Application load success indicators are code-dependent. With the **SOC - Light - RAIL - DMP** example, the LCD displays a light bulb.
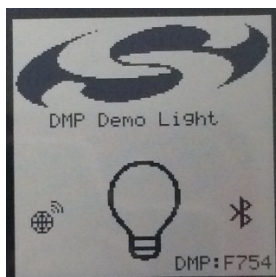
5.2.3 Changing the PHY Configuration

The default PHY configuration for the RAIL/Bluetooth example is a sub Gigahertz configuration. You may want to modify this PHY configuration as you begin to develop applications for your own hardware.

To change the PHY configuration, in the interface for a project based on the **RAIL:Switch** application, select your desired PHY.
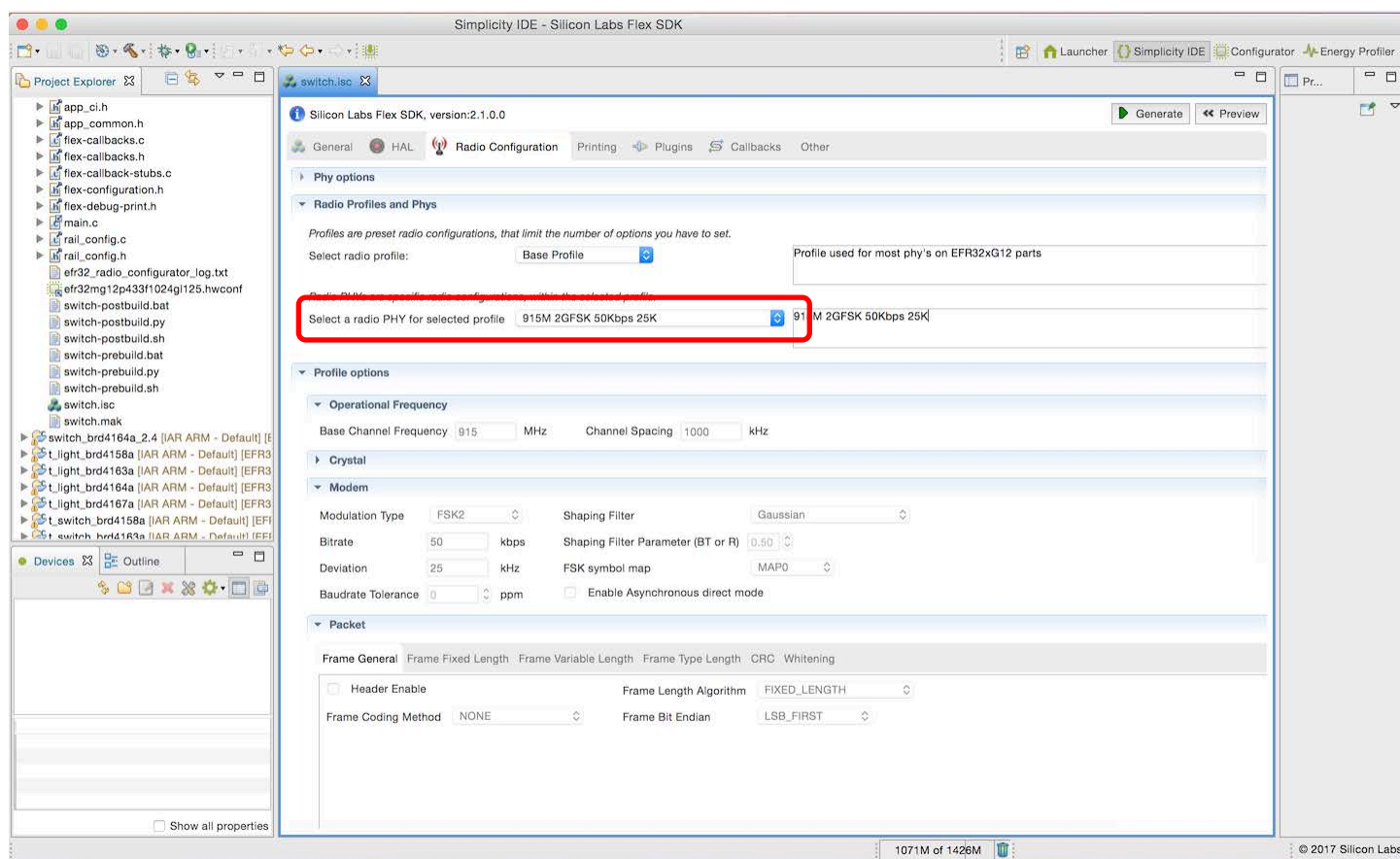
After you have selected the new PHY, click **Generate**. Updated rail_config.c and rail_config.h files are generated. If you want to run the switch application with the new PHY, build it as you normally would.
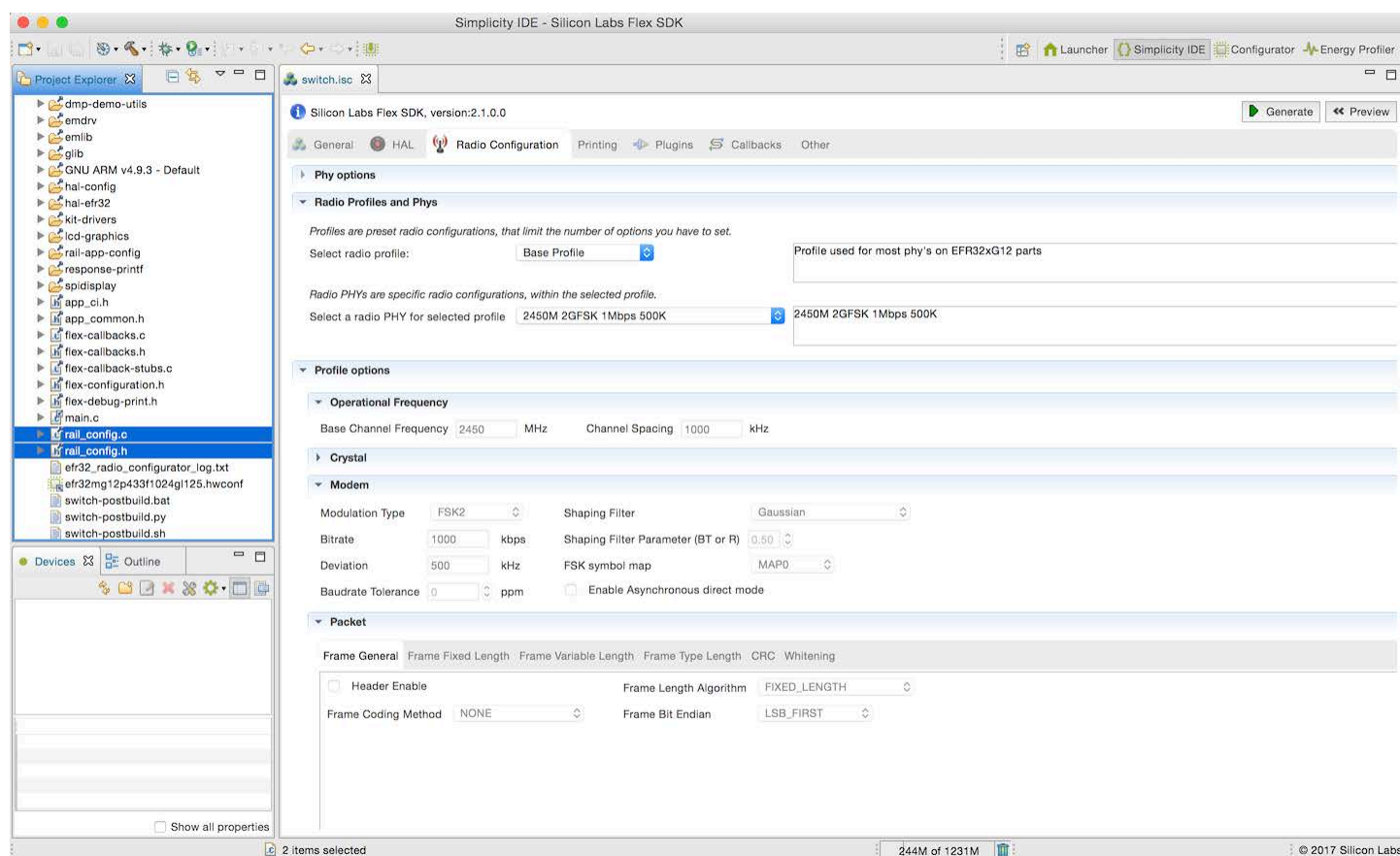
If you have not already started a project in the Bluetooth SDK based on **SOC - Light - RAIL - DMP**, do so now. In your browser of choice, navigate to the Switch folder of your workspace, copy rail_config.c and rail_config.h and paste those files into the SOC - Light - RAIL - DMP folder, being sure to overwrite the files in that folder. Then build and flash the application as you normally would.

6 About the Zigbee/Bluetooth LE Example

The Zigbee/Bluetooth LE Dynamic Multiprotocol example demonstrates a light that can be controlled from both Bluetooth and a Zigbee network. Software.is included both as compiled demonstrations and as example code in the EmberZNet SDK version 6.0.0. Demonstration functionality is illustrated in *QSG155: Using the Silicon Labs Dynamic Multiprotocol Demonstration*. The purpose of the example is show the way of implementing a dynamic multiprotocol application using the Silicon Labs EmberZNet stack.

The example is generated in the Simplicity IDE in the same way as any other Zigbee application is generated, but including some specific configuration options, as described in section **Configuring a Dynamic Multiprotocol Application**

The Dynamic Multiprotocol Demo application has three main components.

1. UI (LCD and Buttons)
2. Zigbee application
3. Bluetooth application

6.1 User Interface

The user interface is developed specifically for the dynamic multiprotocol demonstration, and APIs to update the text and graphic on the LCD are called directly from Zigbee and Bluetooth event handlers. The implementation to manipulate the LCD is contained in the following files,

```
bitmaps.h //Contains the arrays containing the bitmap of the graphics drawn on the LCD
```

```
dmp_ui.c //Contains the functions to change the state of the display based on the state of the application
```

```
dmp_ui.h //Header file exporting functions implemented in the dmp_ui.c
```

The above uses the display driver library supplied by Silicon Labs to update the content on the LCD display mounted on the WSTK.

6.2 Zigbee Application

The example is setup to be light and a coordinator on the Zigbee network. The following cluster set is supported by the application.

Supported Clusters
Basic
Identify
Scenes
Groups
On/Off
ZLL Commissioning

The example also supports Green Power Proxy Basic. Please note that the example was developed with a focus on demonstrating dynamic multiprotocol feature and may not be Zigbee-certifiable.

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

The dynamic multiprotocol applications make use of Micrium OS and the Zigbee application is run as a task of Micrium OS.

The hardware and peripherals of the chip are initialized before any tasks are created. A system task is created after initialization, which then creates the application tasks including the Zigbee and Bluetooth task.

From: micrium-rtos-main.c
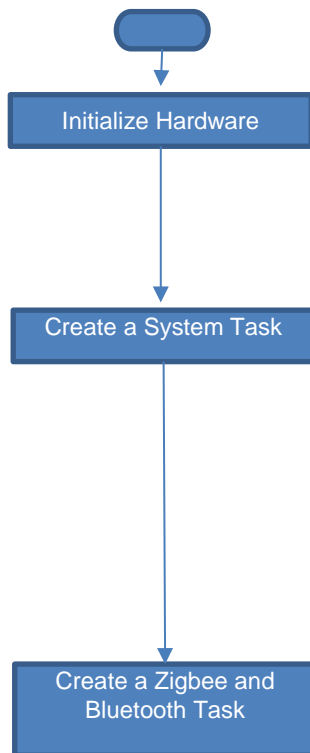
```
Micrium-rtos.main.c
```

```
halInit();
initMicriumCpu();
emberAfMainInit();
```

Initialize Hardware

```
OSTaskCreate(&systemStartTaskControlBlock,
             "System Start",
             systemStartTask,
             NULL,
             SYSTEM_START_TASK_PRIORITY,
             &systemStartTaskStack[0],
             SYSTEM_START_TASK_STACK_SIZE / 10,
             SYSTEM_START_TASK_STACK_SIZE,
             0, // Not receiving messages
             0, // Default time quanta
             NULL, // No TCB extensions
             OS_OPT_TASK_STK_CLR | OS_OPT_TASK_STK_CHK,
             &err);
```

Create a System Task

```
OSTaskCreate(&zigbeeTaskControlBlock,
             "Zigbee Stack",
             zigbeeTask,
             NULL,
             ZIGBEE_STACK_TASK_PRIORITY,
             &zigbeeTaskStack[0],

EMBER_AF_PLUGIN_MICRIUM_RTOS_ZIGBEE_STACK_SIZE / 10,

EMBER_AF_PLUGIN_MICRIUM_RTOS_ZIGBEE_STACK_SIZE,
             0, // Not receiving messages
             0, // Default time quanta
             NULL, // No TCB extensions
             OS_OPT_TASK_STK_CLR | OS_OPT_TASK_STK_CHK,
             &err);

bluetooth_start_task(BLE_LINK_LAYER_TASK_PRIORITY,
                     BLE_STACK_TASK_PRIORITY,
                     emberAfPluginBleGetConfig());
```

Create a Zigbee and Bluetooth Task
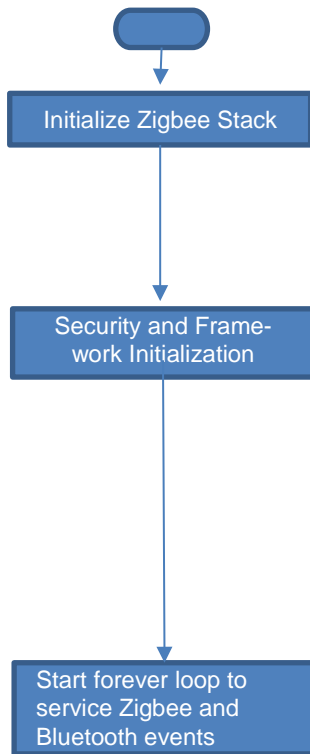
From: af-main-soc.c

Af-main-soc.c

```
status=emberInit();
```

```
emAfInitializeNetworkIndexStack();
// Initialize messageSentCallbacks table
emAfInitializeMessageSentCallbackArray();
emberAfEndpointConfigure();
emAfInit();

// The address cache needs to be initialized and used with the
source routing
// code for the trust center to operate properly.
securityAddressCacheInit(EMBER_AF_PLUGIN_ADDRESS_TABLE_SIZE,
// offset
        EMBER_AF_PLUGIN_ADDRESS_TABLE_TRUST_CENTER_CACHE_SIZE);
// size

EM_AF_NETWORK_INIT();
```

```
while (true) {
    halResetWatchdog();    // Periodically reset the watch-
dog.
    emberTick();            // Allow the stack to run.
    // Allow the ZCL clusters and plugin ticks to run. This
should go
    // immediately after emberTick
    // Skip these ticks if a crypto operation is ongoing
    if (0 == emAfIsCryptoOperationInProgress()) {
      emAfTick();
    }

    emberSerialBufferTick();
    emberAfRunEvents();
}
```

Initialize Zigbee Stack

Security and Frame-work Initialization

Start forever loop to service Zigbee and Bluetooth events

Once the Zigbee stack is set up to run, subsequent interactions with the stack occurs via event handlers, as shown in the following figure.
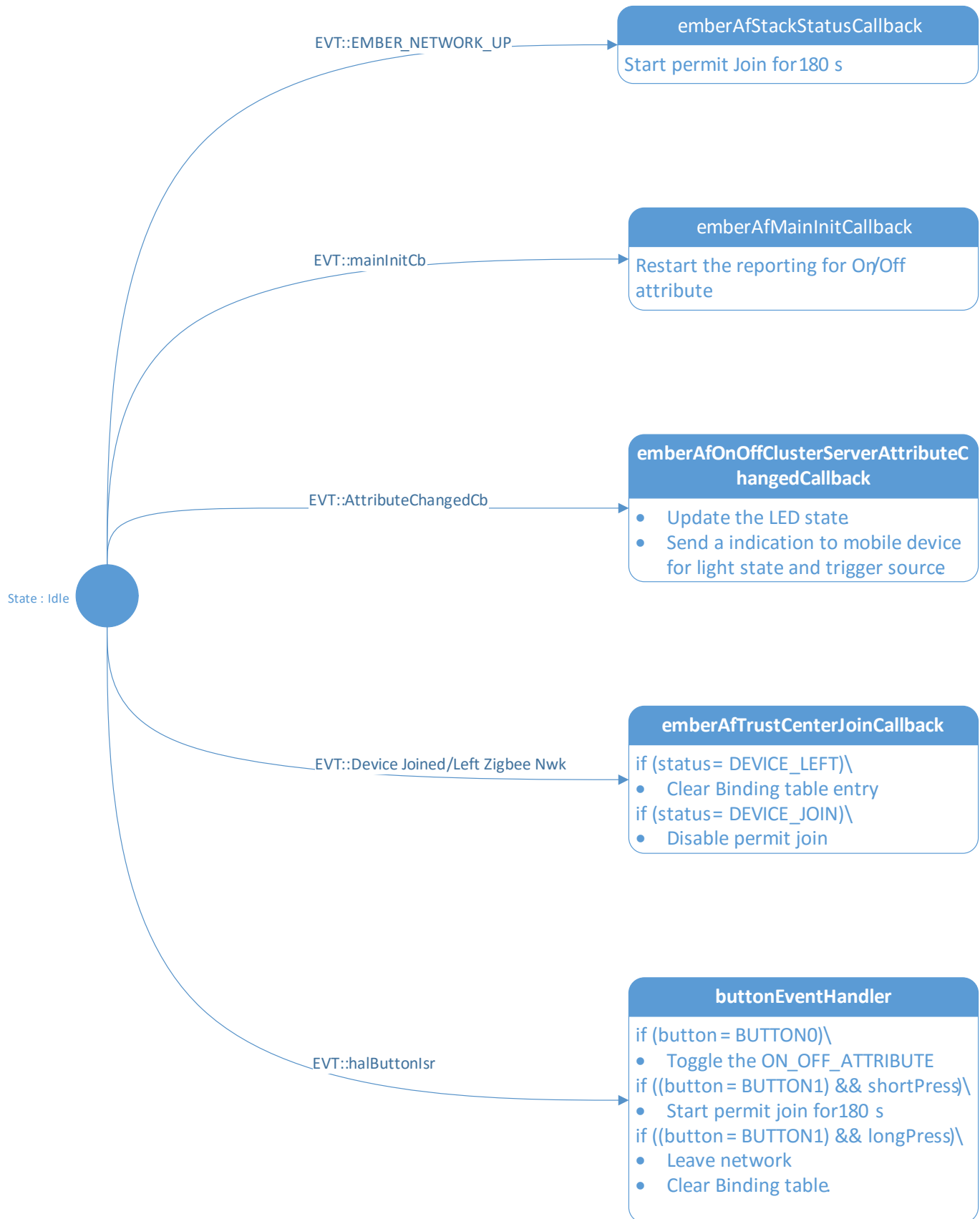
6.3   Bluetooth Application

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

6.3.1   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 Wireless Gecko 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

6.3.2   Beacons

The application implements both iBeacons as well as Eddystone beacons. The default behavior is to transmit each beacon at alternating 500 mS interval as shown below
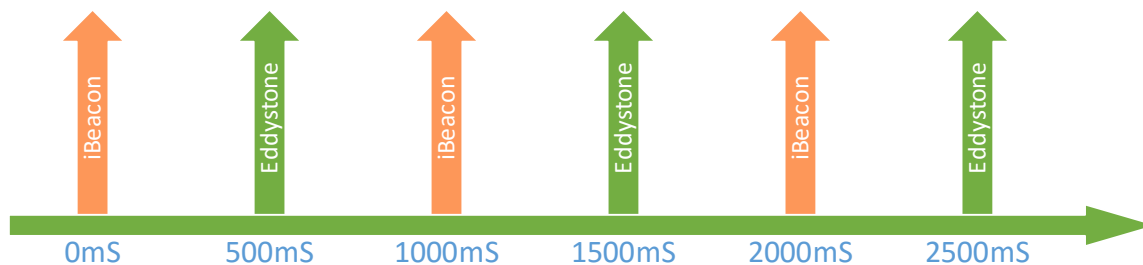
The beacons are set up in `bleAdvertsEventHandler` in the callbacks file.

6.3.3 Bluetooth Event Handling

The Bluetooth stack is initialized as part of the System Task, as shown in the Zigbee implementation section. The Bluetooth task handles the Bluetooth LE link layer messaging and management. The Bluetooth stack's interaction with the user application is through a framework plugin. 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.
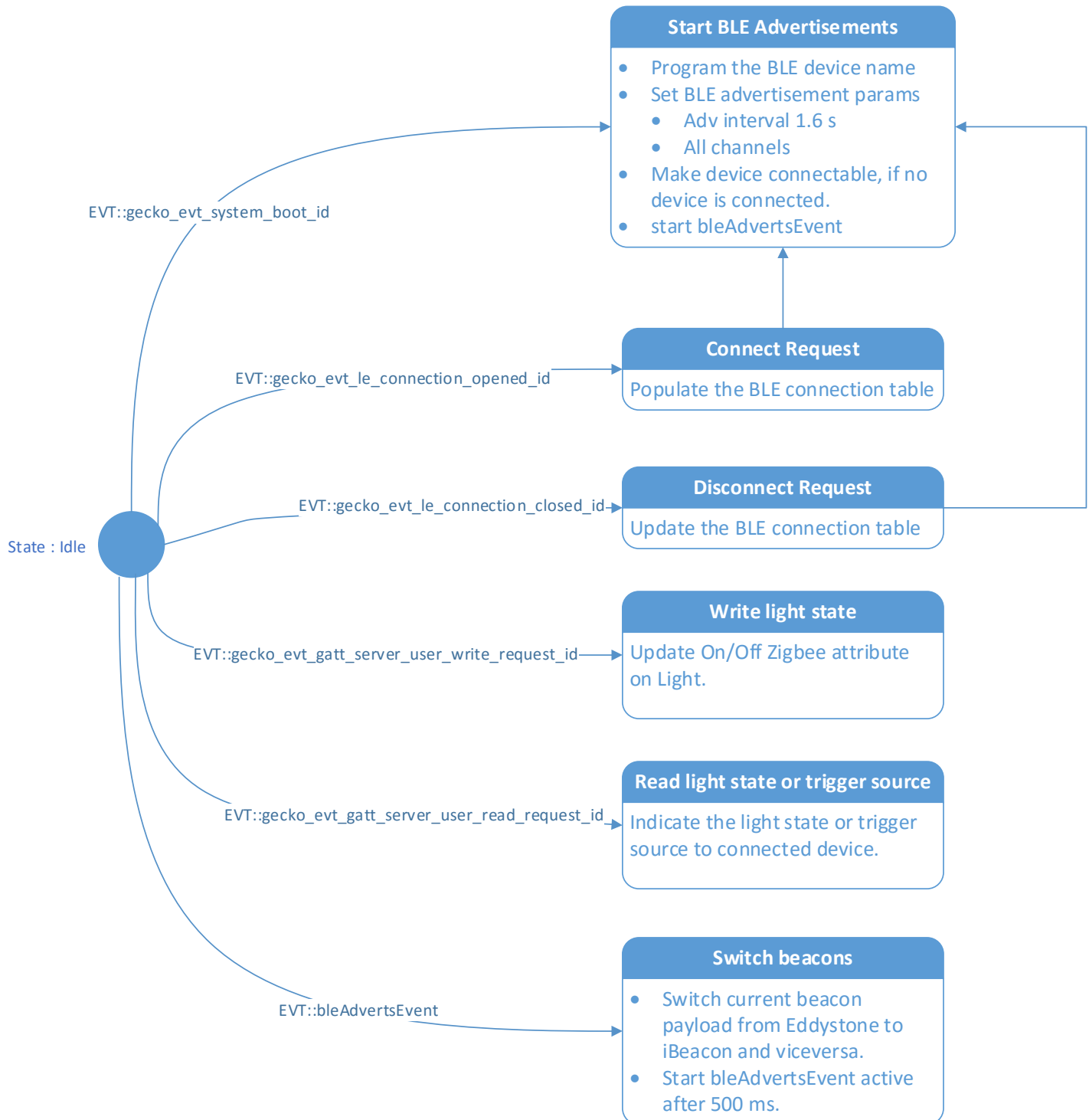
6.3.4 Bluetooth and Zigbee Interaction

The primary purpose of the example application 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.

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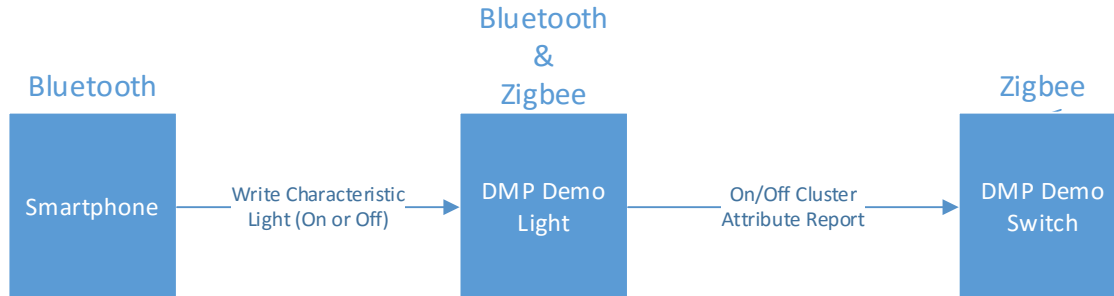
The application's services and characteristics are pre-selected in the GATT editor in Simplicity Studio. Upon generation the characteristics are #define in the gatt_db.h. Using the #define reference, the characteristics can then be coupled to read and write Bluetooth requests. For example the Light characteristic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

```
static const AppCfgGattServerUserWriteRequest_t appCfgGattServerUserWriteRequest[] =
{
  { gattdb_light_state, writeLightState },
  { 0, NULL }
};
```

The application implements the Zigbee attribute write and a Bluetooth write response in the `writeLightState` function as follows

```
static void writeLightState(uint8_t connection, uint8array *writeValue)
{
  lightDirection = DMP_UI_DIRECTION_BLUETOOTH;
  emberAfWriteAttribute(emberAfPrimaryEndpoint(),
                        ZCL_ON_OFF_CLUSTER_ID,
                        ZCL_ON_OFF_ATTRIBUTE_ID,
                        CLUSTER_MASK_SERVER,
                        (int8u *)&writeValue->data[0],
                        ZCL_BOOLEAN_ATTRIBUTE_TYPE);
  gecko_cmd_gatt_server_send_user_write_response(
    connection,
    gattdb_light_state,
    ES_WRITE_OK
    );
}
```

The `emberAfWriteAttribute()` 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 `emberAfOnOffClusterServerAttributeChangedCallback()` 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

The flow in the other direction, that is a change in the Light state from Zigbee connected device, is shown in the following figure.
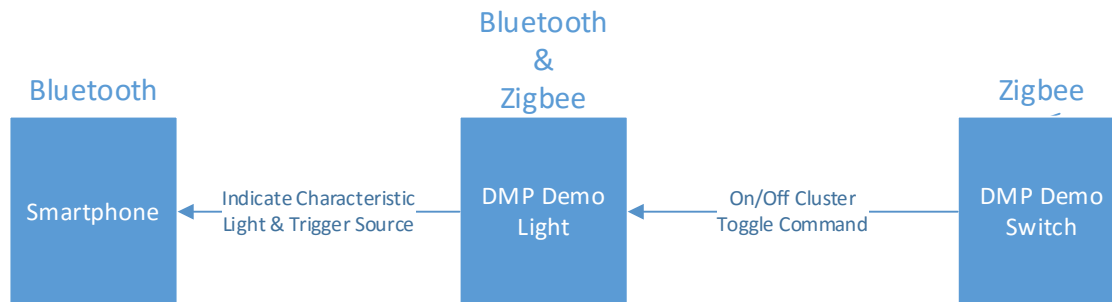
Any on-off client on the same network as the Light can send an on-off cluster's On, Off or Toggle command to the Light to change its state. Once such a command is received over the Zigbee interface the Silicon Labs Zigbee framework interprets it and calls an appropriate handler to change the value of on-off attribute of the on-off server cluster. In the example **DynamicMultiprotocolDemoSwitch-SoC** application the on-off client sends a Toggle command to the Light, which toggles the value of the on-off attribute and triggers the `emberAfOnOffClusterServerAttributeChangedCallback()`. 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.

```
void emberAfOnOffClusterServerAttributeChangedCallback(int8u endpoint,
                                                       EmberAfAttributeId attributeId)
{
  EmberStatus status;
  int8u data;

  if (attributeId == ZCL_ON_OFF_ATTRIBUTE_ID) {
    status =  emberAfReadAttribute(endpoint,
                                   ZCL_ON_OFF_CLUSTER_ID,
                                   ZCL_ON_OFF_ATTRIBUTE_ID,
                                   CLUSTER_MASK_SERVER,
                                   (int8u*)&data,
                                   sizeof(data),
                                   NULL);

    if (status == EMBER_ZCL_STATUS_SUCCESS) {
      if (data == 0x00) {
        halClearLed(BOARDLED0);
        halClearLed(BOARDLED1);
        dmpUiLightOff();
        notifyLight(currentConnection, 0);
      } else {
        halSetLed(BOARDLED0);
        halSetLed(BOARDLED1);
        notifyLight(currentConnection, 1);
        dmpUiLightOn();
      }
      if ( (lightDirection == DMP_UI_DIRECTION_BLUETOOTH)
           || (lightDirection == DMP_UI_DIRECTION_SWITCH) ) {
        dmpUiUpdateDirection(lightDirection);
      } else {
        lightDirection = DMP_UI_DIRECTION_ZIGBEE;
        dmpUiUpdateDirection(lightDirection);
      }
      ble_lastEvent = lightDirection;
      lightDirection = DMP_UI_DIRECTION_INVALID;

      if (ble_lastEvent != DMP_UI_DIRECTION_INVALID) {
        if ( (ble_lightState_config != GAT_RECEIVE_INDICATION) && (ble_lastEvent_config ==
GAT_RECEIVE_INDICATION)) {
          notifyTriggerSource(currentConnection, ble_lastEvent);
        }
      }
    }
  } else {
  }
}}
```
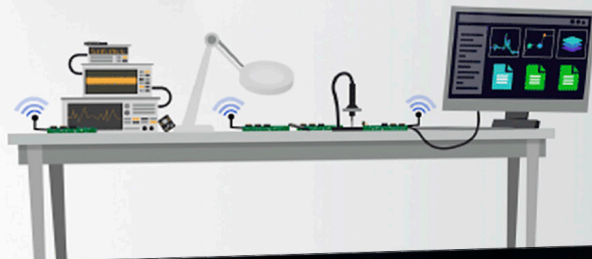
7 Future Enhancements

Dynamic Multiprotocol is still an evolving technology. Silicon Labs is working on enhancing the current operations, as well as adding support for additional wireless protocols operating in a Dynamic Multiprotocol environment.

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 and Community
community.silabs.com

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

SILICON LABS

http://www.silabs.com