AN1269: Dynamic Multiprotocol Development with *Bluetooth*® and Proprietary Protocols on RAIL in GSDK v3.x and Higher

This version of AN1269 has been deprecated.

For the latest version, see docs.silabs.com.

****

This application note provides details on how to develop a multiprotocol application running Bluetooth and a proprietary protocol at the same time, using SDKs from Gecko SDK Suite v3.x. First, the criteria for the coexistence of Bluetooth and a proprietary protocol are discussed. Then the application note guides you through how to create a new DMP application, how to configure Bluetooth and your proprietary protocol, and how to transmit and receive proprietary packets while Bluetooth is running. Finally, a Light/Switch DMP example is introduced in more details. For background on Dynamic Multiprotocol Application development in general and about Bluetooth task priorities and scheduling, see *UG305: Dynamic Multiprotocol User's Guide*.

KEY POINTS

- Generic guidelines for protocol coexistence
- Generating and configuring a new Bluetooth/Proprietary DMP project
- Sending and receiving proprietary packets
- Using RAIL priorities
- Building and understanding the Light/Switch DMP example

1. Introduction

UG305: Dynamic Multiprotocol User's Guide provides information about the Dynamic Multiprotocol solution, where two protocols are running on the same device in parallel, and includes general background as well as information on Bluetooth task priorities and time scheduling. This application note introduces the Bluetooth / Proprietary multiprotocol solution. It assumes that the reader is familiar with the principles of Dynamic Multiprotocol and with all the terms related to it. The Dynamic Multiprotocol projects require an RTOS for scheduling. Currently, the Micrium RTOS and the FreeRTOS are supported. The FreeRTOS is included in the SDK.

1.1 Requirements

To be able to use all the features discussed in this document, you will need the followings installed on your computer:

- Bluetooth SDK version 3.0.0 or higher
- (optional*) Micrium OS-5 kernel

To be able to run the Light/Switch example, you will need the following installed on your computer:

- Bluetooth SDK version 3.0.0 or higher
- Flex SDK version 3.0.0 or higher
- (optional*) Micrium OS-5 kernel
- (optional*) IAR Embedded Workbench for ARM (IAR-EWARM) (required for the Flex (RAIL) - Switch application). See the release notes for the Bluetooth SDK for the required IAR-EWARM version.

*Required only when the Micrium RTOS is used.

2. Guidelines for Bluetooth and Proprietary Coexistence

When you start implementing a Bluetooth / Proprietary DMP application, the first thing to consider is if your proprietary protocol is compatible with Bluetooth. Here are some guidelines that you should always consider:

- **Bluetooth is deterministic**. The huge advantage of the Bluetooth protocol in a DMP scenario is that it does not send and receive packets at random times, but at predefined time instances – always at the start of a connection interval. This means, among other things, that Bluetooth does not need a background receive, and ***your proprietary protocol can receive in the background***, of course with some interruptions.
- **Bluetooth needs time accuracy**. The consequence of predefined time instances is that ***Bluetooth radio operations need very accurate timing***. Radio operation timing needs 500 ppm accuracy. If you delay a Bluetooth packet, it will not be received on the other side. So in case of collision with a proprietary packet, either the ***proprietary packet has to be delayed***, or one of the packets has to be dropped.
- **Bluetooth connection is active**. Once a Bluetooth connection is established, the connection is kept alive by sending and receiving at least an empty packet every connection interval. Consequently, your proprietary protocol needs to be prepared to be ***interrupted every connection interval***. You can, however, set the connection interval to a long period if you do not need low Bluetooth latency. You can also use the peripheral latency parameter to make Bluetooth communication less frequent on the peripheral side.
- **Bluetooth uses short packets**. If there is no data to be sent, the Bluetooth connection is kept alive by empty packets. An empty packet takes 80 µs to be sent out on 1M PHY, and 40 µs on 2M PHY. Empty packets sending + inter frame space + empty packet receiving takes 80 + 150 + 80 = 310 µs or 40 + 150 + 40 = ***230 µs***. This is the usual time needed by Bluetooth in every connection interval. The largest Bluetooth packet has a 257 byte payload which takes 2120 µs to be sent on 1M PHY and 1060 µs on 2M PHY. Along with receiving an empty response packet, this takes 2120 + 150 + 80 = ***2350 µs*** on 1M PHY and 1060 + 150 + 40 = ***1250 µs*** on 2M PHY.
- **Bluetooth uses packet chains**. If the data to be sent does not fit into one packet, Bluetooth communication can be extended within a connection interval; that is, you can ***expect that more than one packet is sent and received*** in an interval, but this is rare.
- **Bluetooth is robust**. If a Bluetooth packet cannot be sent, then it will be ***retransmitted in the next connection interval***. If a Bluetooth packet is received with a ***CRC error, it is always signaled by the other side*** by not sending a response packet. Again, the packet will be retransmitted in the next connection interval. The only limit is the supervision timeout. If there is no successful transmission within the supervision timeout, then the connection is dropped. In other words, Bluetooth communication ***can be subdued by higher priority radio tasks for a time interval shorter that the supervision timeout***.

Summary: When implementing your DMP protocol, you have to take into account that Bluetooth will need the radio every connection interval for a short time (230 µs – 2350 µs). Bluetooth needs accurate timing, so Bluetooth packets cannot be delayed. The Bluetooth packets can interrupt both your packet sending and packet receiving. Therefore, the proprietary protocol should implement acknowledgement and retransmission mechanisms, or a deterministic timing that is interleaved with the Bluetooth communication. Bluetooth communication can be subdued by a higher priority radio task for a time interval shorter than the supervision timeout.

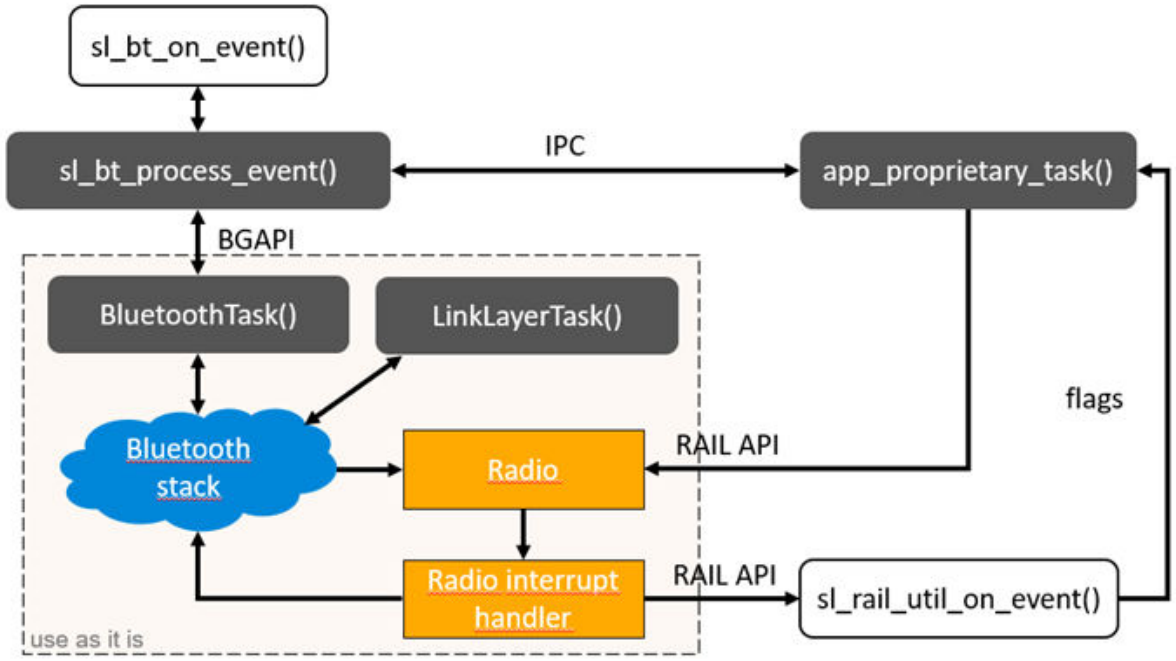
3. Software Architecture of a Bluetooth / Proprietary DMP application

DMP applications require an RTOS. The RTOS helps run the Bluetooth and Proprietary protocols in parallel and independently. In this document, the term RTOS refers both to the Micrium RTOS and the FreeRTOS, included with Silicon Labs Bluetooth SDK version 3.1.0. The adaptation layer has been designed to work with Micrium RTOS and FreeRTOS.

Since the Bluetooth stack itself is just a collection of functions, Bluetooth needs separate tasks to run the stack. The `BluetoothTask()` and the `LinkLayerTask()` are responsible for this, and they can be used as they are. The functions of the Bluetooth stack can be accessed through these tasks using BGAPI, as in the case of an RTOS-less or an NCP application. The Bluetooth application (handling Bluetooth events and calling Bluetooth commands) has to be implemented by the developer in `sl_bt_on_event()` , which is (indirectly) called from the `sl_bt_event_handler_task()`. For details, refer to *AN1260: Integrating v3.x Silicon Labs Bluetooth® Applications with Real-Time Operating Systems*.

The proprietary protocol is implemented in the `app_proprietary_task()`. Unlike Bluetooth, the proprietary protocol can access the radio directly through the RAIL API. RAIL events need a callback function – `sl_rail_util_on_event()` – to be defined. This function is called every time a new RAIL event is generated, and can notify the application about the event. Note: `sl_rail_util_on_event()` is called from interrupt context, so only time-critical functions should be implemented in it. Everything else should be done in the application.

Although the Bluetooth and Proprietary applications are independent, they can communicate using inter-process communication (IPC).



4. Developing a Bluetooth / Proprietary DMP Project

Three steps are required when developing a Bluetooth / Proprietary DMP project:

1. Create the project.
2. Configure Bluetooth.
3. Configure the Proprietary Protocol.

4.1 Create a New Project

Silicon Labs Bluetooth SDK (v3.2 or later) include four software samples, which can be used as a starting point for every Bluetooth / Proprietary application.

- Bluetooth - SoC Empty RAIL DMP FreeRTOS
- Bluetooth - SoC Empty RAIL DMP Micrium OS
- Bluetooth - SoC Empty Standard DMP FreeRTOS
- Bluetooth - SoC Empty Standard DMP Micrium OS

Each sample:

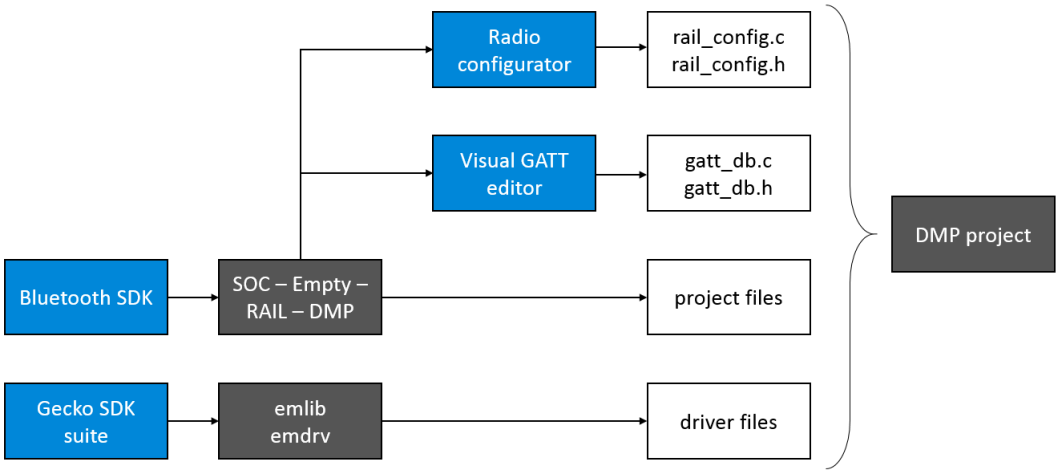
- Includes the multiprotocol RAIL library
- Includes the Bluetooth library
- Includes the selected RTOS
- Has a default Bluetooth GATT database configuration
- Has a default RAIL configuration
- Has a default RTOS configuration
- Implements Bluetooth initialization
- Implements RAIL initialization
- Implements RTOS initialization

The sample types differ in that the 'RAIL' samples contain a radio configurator, so they can be used for any proprietary protocol, whereas the 'standard' samples uses the IEE802.15.4 standard protocol.

The only thing you have to do is to modify the configurations according to your needs and implement the Bluetooth application task and the Proprietary application task. As default, the `app_proprietary_task()` is defined and implemented in the files *app_proprietary.c* and *app_proprietary.h*.

For the Bluetooth part, the default implementation contains the Bluetooth event handler, the `sl_bt_on_event()` function, defined in the *app_bluetooth.c* file.

The GATT database can be configured with the visual GATT Configurator in Simplicity Studio 5, while the RAIL configuration can be generated with the Radio Configurator tool. You may also need to add some **emlib** and **emdrv** files to your project to support peripheral configuration. The general workflow to create a DMP project looks like this:



To create a new project.

1. Open Simplicity Studio 5.
2. Select a connected device in the Debug Adapters view, or select a part in the My Products view.
3. Click **File > New > Silicon Labs Project Wizard**.
4. Review your SDK and toolchain. If you have more than one GSDK version installed, verify that Gecko SDK Suite v3.x is shown. If you wish to use IAR instead of GCC, be sure to change it here. Once you have created a project, it is difficult to change toolchains. Click **NEXT**.
5. On the Example Project Selection dialog, filter on Bluetooth and select **Bluetooth - SoC Empty RAIL DMP FreeRTOS**. Click **NEXT**.
6. Name your project. Click [**FINISH**].

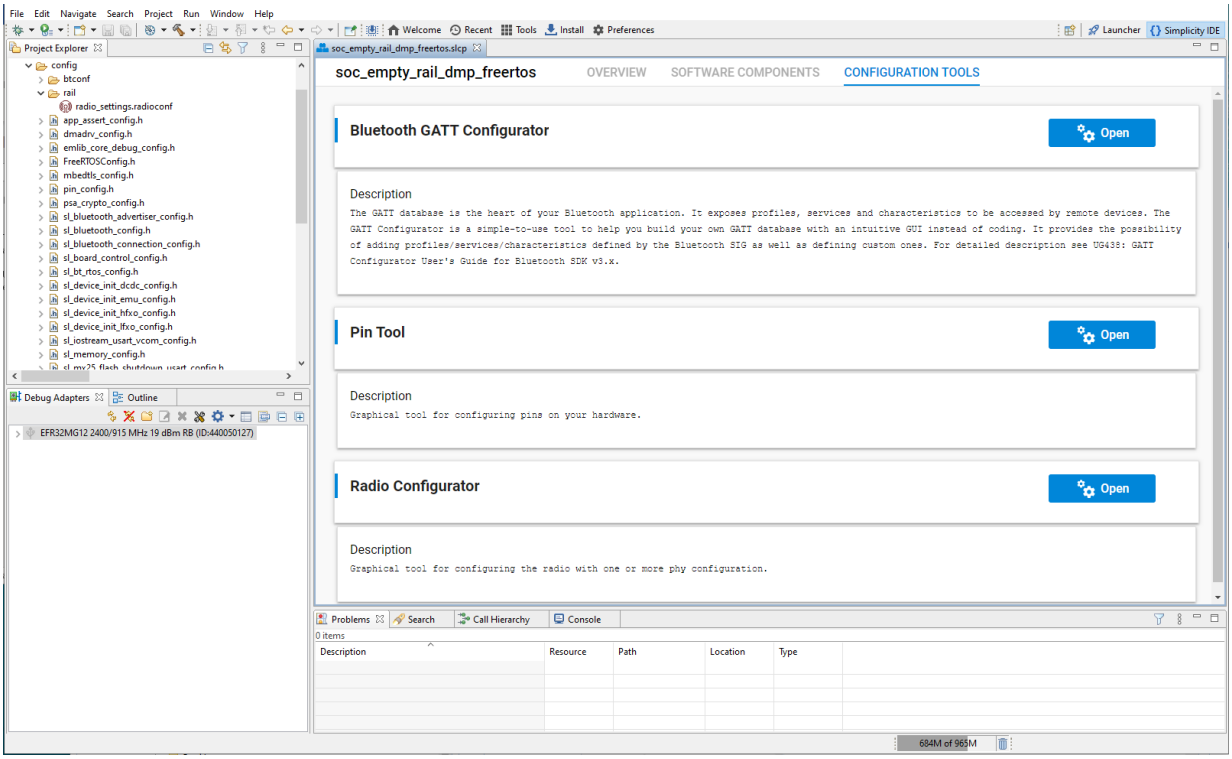
4.2 Configure Bluetooth

Configuring Bluetooth consists of two steps:

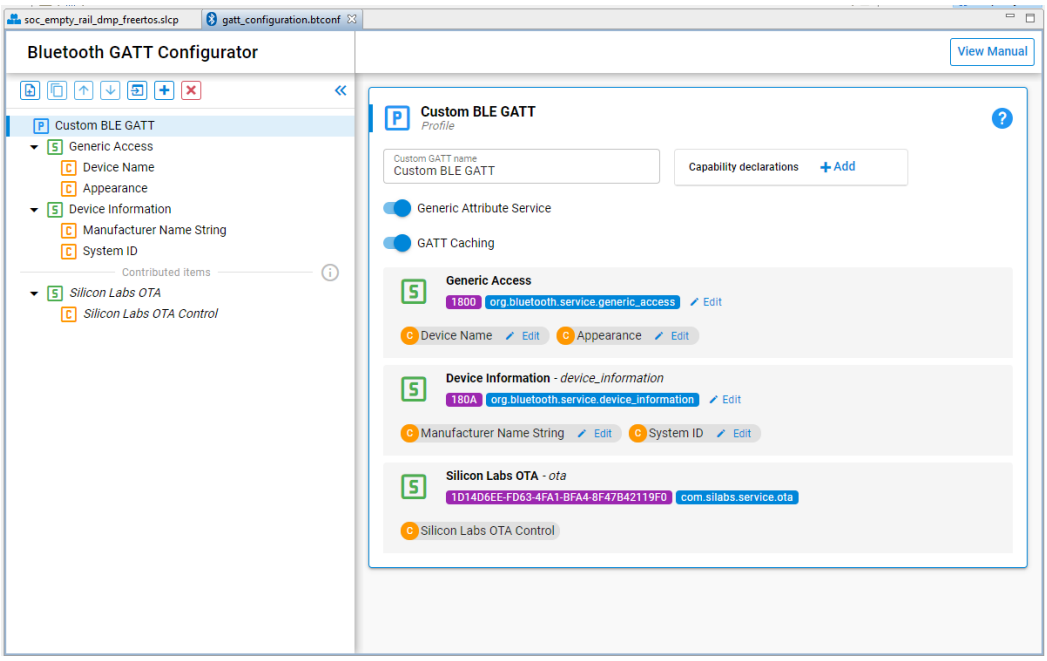
- Configuring the local GATT database
- Configuring the Bluetooth stack

To configure the local GATT database, use Simplicity Studio 5's GATT Configurator.

1. Open the .slcp file in the project (if it is not already open).
2. Click the CONFIGURATION TOOLs tab.
3. Click **Open** next to **Bluetooth GATT Configurator**.

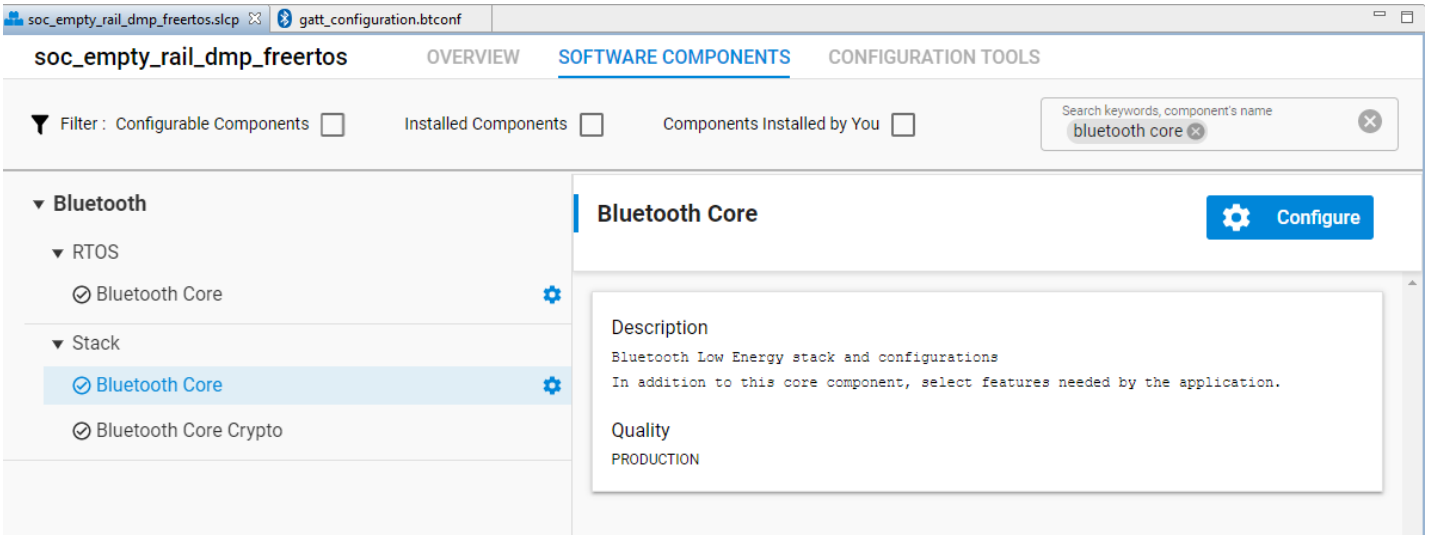


4. Add your custom services and characteristics as described in *QSG169: Bluetooth® SDK v3.x Quick Start Guide* (or use the default GATT database).
5. Your changes are automatically saved and project files are generated.

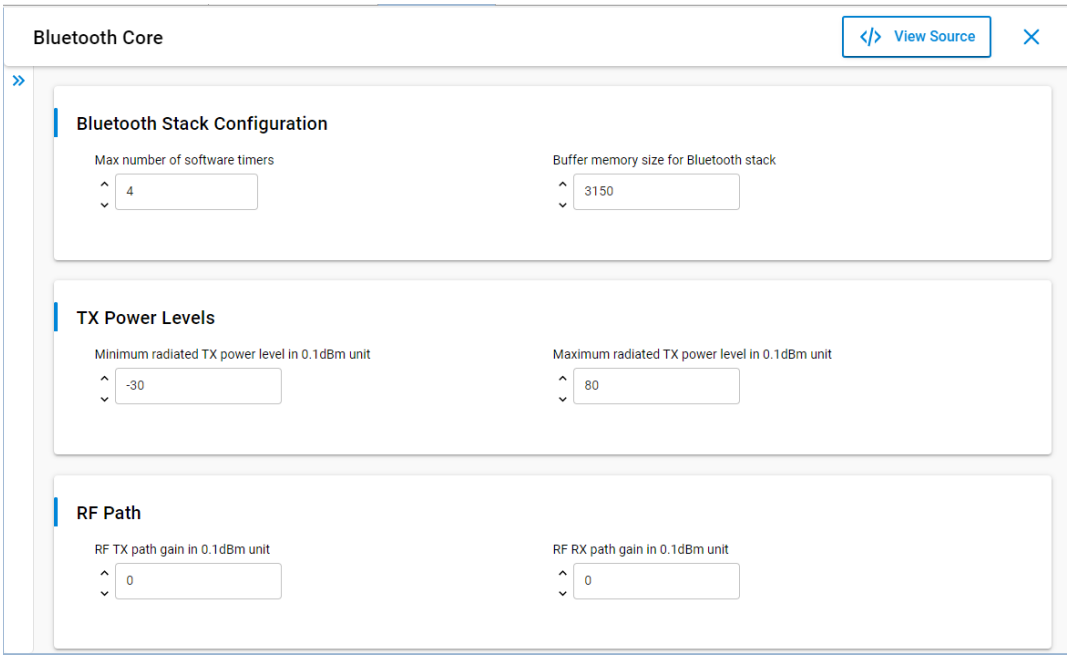


To configure the Bluetooth stack:

1. Go to the SOFTWARE COMPONENTS tab.
2. Find Bluetooth > Stack > Bluetooth Core component.



3. Change the configuration according to your needs. For details, see *UG434: Silicon Labs Bluetooth® C Application Developers Guide for SDK v3.x* (or use the default configuration).



4.3 Configure the Proprietary Protocol

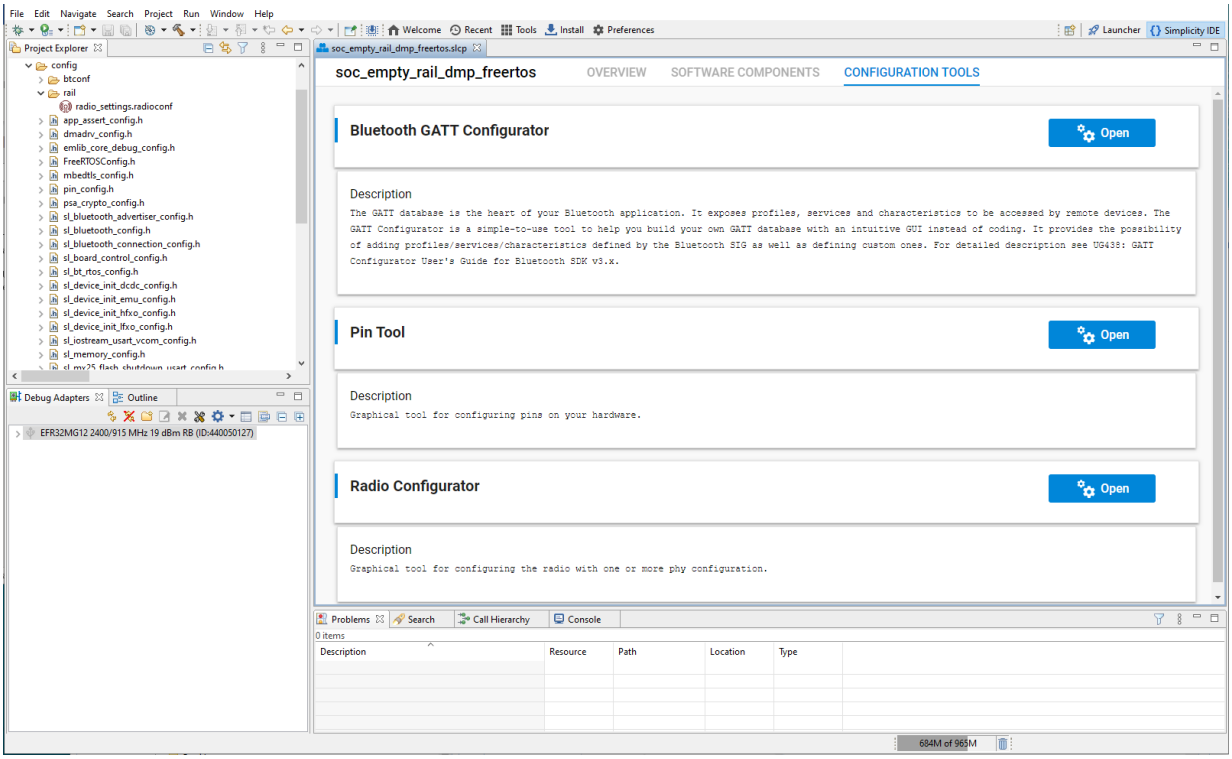
4.3.1 Using the Radio Configurator

Configuring the proprietary protocol consists of two steps:

- Configuring the radio channels (base frequency, modulation, and so on)
- Configuring the RAIL

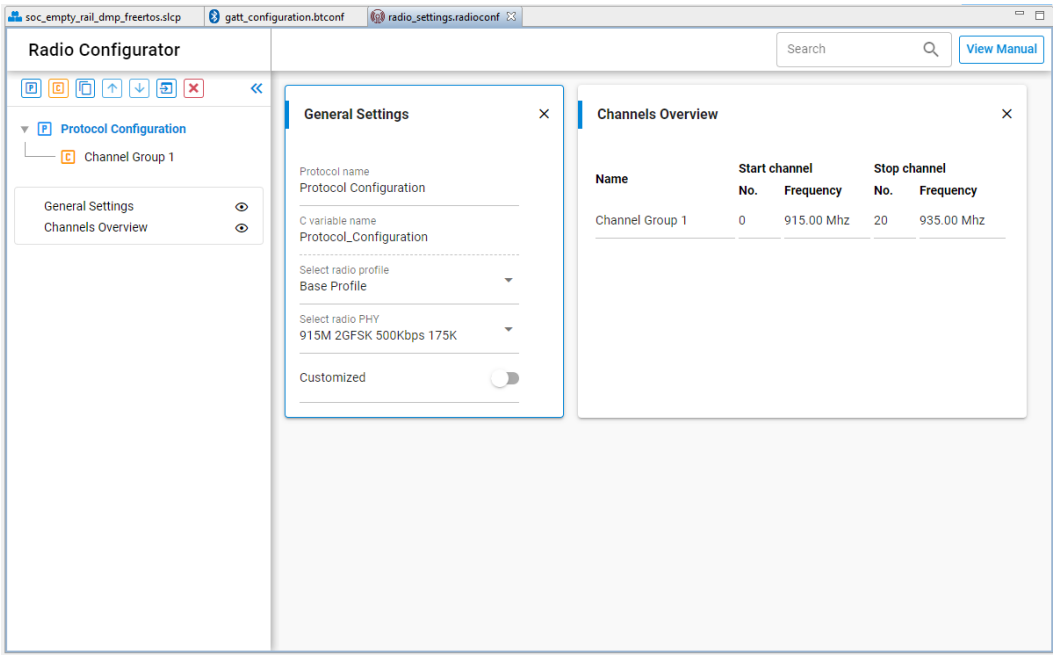
To configure the radio channels, use Simplicity Studio 5's Radio Configurator tool:

1. Open the .slcp file in the project (if it is not already open).
2. Click the CONFIGURATION TOOLS tab.
3. Click **Open** next to **Radio Configurator**.



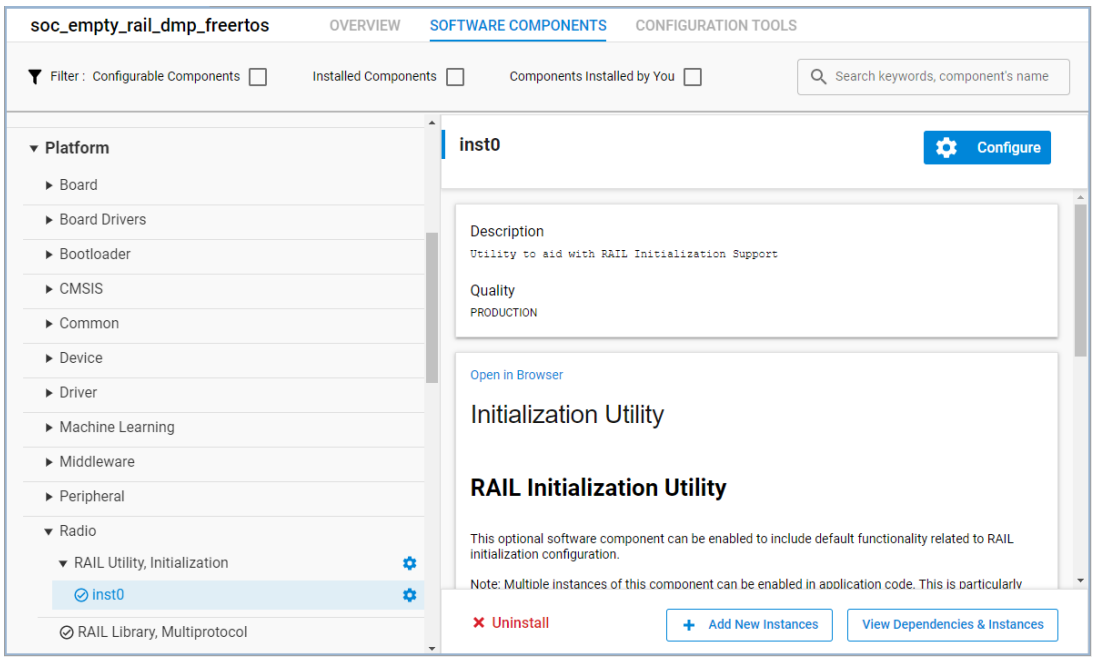
4. Select **Base Profile** from the radio profiles.

5. Select a predefined radio PHY from the list, or select **Customized**, and apply your settings. For details, see *AN1253: EFR32 Radio Configurator Guide for Simplicity Studio 5*.

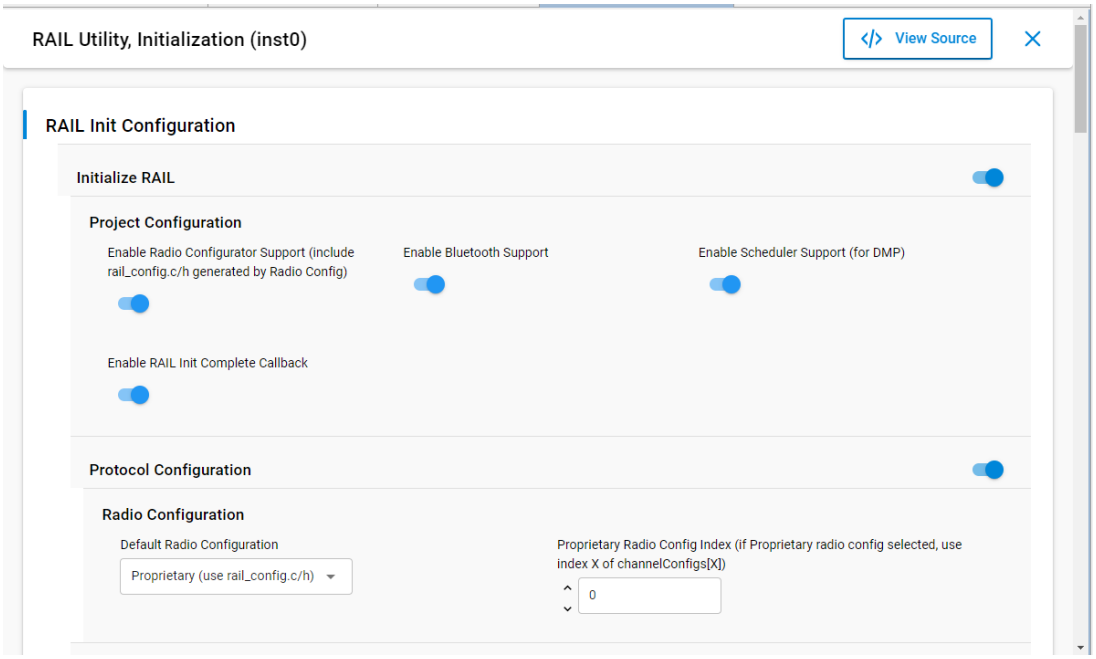


To configure RAIL:

1. On the Software Components tab, select Platform > Radio > RAIL Utility, Initialization > inst0.



2. Click **Configure**. Change configurations as needed.



4.3.2 Using Standard Protocol APIs

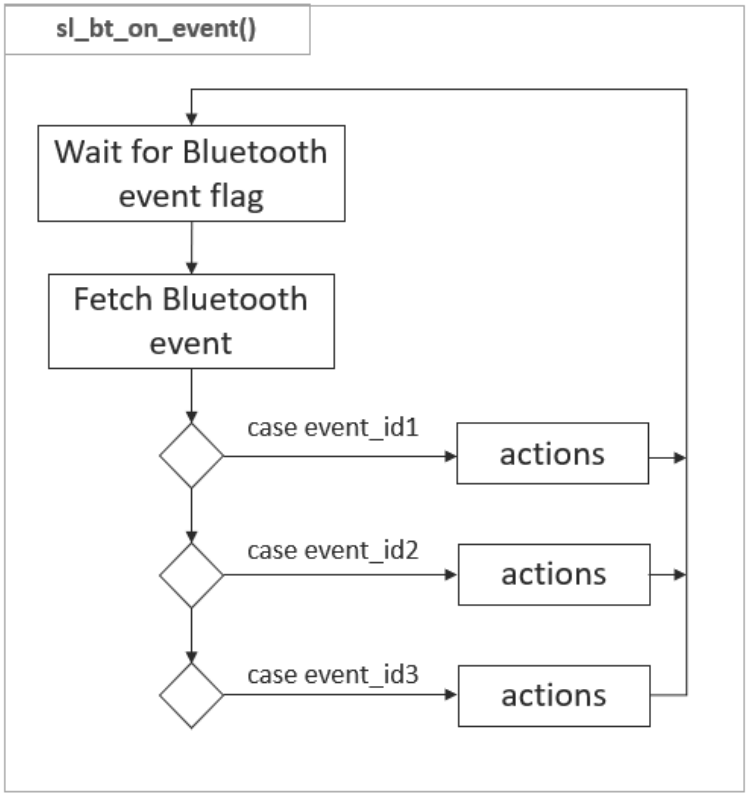
In the “Bluetooth - SoC Empty Standard DMP” sample project, the radio is configured with APIs. The sample project contains a default configuration for the IEE802.15.4 standard protocol. This configuration is set in the function “*app_proprietary_init()*”. For more information about the possible configurations, refer to the API documentation on docs.silabs.com.

4.4 Develop Bluetooth Application

Bluetooth applications have to be implemented the same way as in a non-DMP scenario:

- BGAPI commands can be called from anywhere (except from interrupt context!)
- BGAPI events have to be fetched from the internal event queue of the Bluetooth stack. This is typically done in an infinite loop.

A single protocol Bluetooth application can run with or without RTOS. The DMP Bluetooth application can, however, only run over RTOS. As described in section 3. Software Architecture of a Bluetooth / Proprietary DMP application, you must implement Bluetooth event handling in the Bluetooth application task. The skeleton of this task is implemented in main.c. To handle new Bluetooth events, simply add new case statements with the appropriate event IDs. The general process can be seen in the following figure:



4.5 Develop Proprietary Application

Proprietary application uses RAIL directly:

- RAIL API commands can be called from anywhere.
- RAIL API events have to be handled in the events callback function.

Almost all RAIL APIs can be used in DMP, but a few are incompatible (like `RAIL_HoldRxPacket()`), and a few work slightly differently. For example, automatic state transitions are defined differently due to the concept of background Rx, which is specific on DMP. See *UG305: Dynamic Multiprotocol User's Guide* for details.

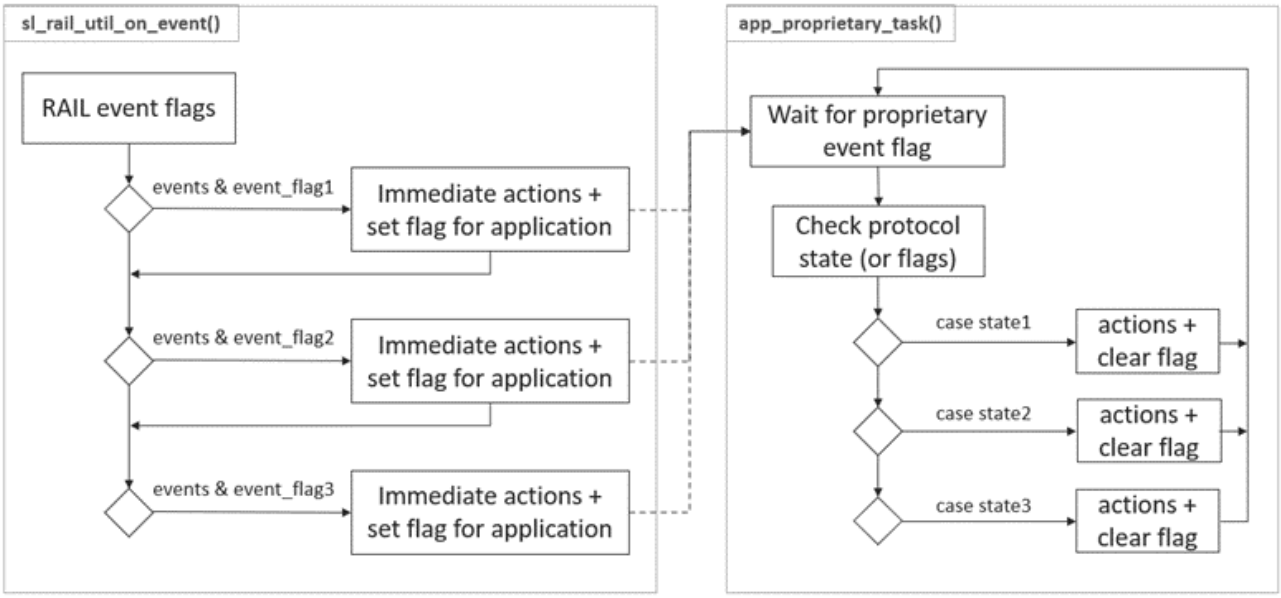
By default, the events callback function is set to `sl_rail_util_on_event()`, just like in a regular RAIL application. An empty `sl_rail_util_on_event()` function is implemented as a weak function in *sl_rail_util_callbacks.c*. It can be overloaded in the application. This function is called every time a new radio event intended for the proprietary protocol is received from RAIL. Each RAIL event sets a specific flag in the 64-bit bitfield. Be aware that multiple flags may be set, so you may have to handle multiple events within one callback. Note: The events callback function is almost always called from an interrupt context, so you have to handle it as an interrupt handler! Do only quick calculations, and set a flag to inform your main loop about the changes.

In the DMP context, you should also prepare for more error events: `RAIL_EVENT_SCHEDULER_STATUS` should be implemented, as that is the event which is triggered if a proprietary radio is interrupted by Bluetooth.

Upon completing a finite radio task (like transmission), `RAIL_YieldRadio()` or `RAIL_Idle()` should be called to let the radio scheduler know that other protocols might use the radio.

The main loop to process the radio events is implemented in the `app_proprietary_task()`, which runs parallel to the `sl_bt_event_handler_task()` that ultimately calls the `sl_bt_on_event()` event handler. It is the developer's job to decide how to communicate between the radio event handler (`sl_rail_util_on_event()`) and the `app_proprietary_task()`, but in general use the services of the RTOS, like semaphores, flags, message queues, and so on.

The general process is shown in the following figure:



4.6 Communication between Bluetooth and Proprietary Protocol

Bluetooth and the proprietary protocol are running parallel in two independent tasks. However, often they need to be synchronized, for example if you want to send out a proprietary packet when a value changed in the local GATT database, or you want to change a value in the local GATT database when you received a proprietary packet.

To notify the proprietary task from the Bluetooth task, or the other way around, the easiest way is to set an RTOS flag. You can define a queue for events and use that to notify the other task. From the proprietary task, you can also set an external event to the Bluetooth stack, using the function `sl_bt_external_signal()`. This will generate an `sl_bt_evt_system_external_signal_id` event in the Bluetooth stack.

5. Examples

5.1 Sending Proprietary Packets

This simple example sends out a proprietary packet every time a specific characteristic in the local GATT database is written.

1. Create a new **Soc Empty Rail Dmp** project as described in section 4.1 Create a New Project.
2. In the GATT configurator, add a new characteristic to the GATT database (as described in *QSG169: Bluetooth® SDK v3.x Quick Start Guide*) with the following parameters:
   a. Name: Proprietary characteristic
   b. ID: prop_char
   c. Value type: hex
   d. Length: 16 byte
   e. Properties: Read, Write, Notify
3. Define a CHARACTERISTIC_CHANGED flag. This flag will be used in the communication between `sl-bt-on-event()` and the app_proprietary_task, as part of the proprietary_event_flags flag group.

```
#define CHARACTERISTIC_CHANGED ((OS_FLAGS)0x01)
```

4. Create a Tx FIFO. Define the following in *app_proprietary.c*:

```
#define RAIL_TX_FIFO_SIZE (64)
static uint8_t txFifo[RAIL_TX_FIFO_SIZE];
```

5. In the Bluetooth application task (more precisely in `sl_bt_on_event()`):
   a. Add a new event handler to the switch – case statement to handle characteristic value changes.
   b. Check if it is the prop_char that has changed.
   c. Set a flag to notify the proprietary protocol.

```
case sl_bt_evt_gatt_server_attribute_value_id:
if (evt->data.evt_gatt_server_attribute_value.attribute == gattdb_prop_char)
   {
              OSFlagPost(&proprietary_event_flags,
                                 CHARACTERISTIC_CHANGED,
                                 OS_OPT_POST_FLAG_SET,
                                 &err);
   }
break;
```

6. In the `app_proprietary_task()` – before the infinite loop:
   a. Set up the Tx FIFO for RAIL.
   b. Define scheduler info for the packet to be sent.

```
RAIL_SetTxFifo(railHandle, txFifo, 0, RAIL_TX_FIFO_SIZE);
RAIL_SchedulerInfo_t txSchedulerInfo = (RAIL_SchedulerInfo_t){ .priority = 100,
                                       .slipTime = 100000,
                                       .transactionTime = 800 };
```

7. Within the infinite loop of the `app_proprietary_task()`:
   a. Wait for the CHARACTERISTIC_CHANGED flag.
   b. Copy the content of the characteristic into the Tx FIFO.
   c. Send out the packet.

```
while (DEF_TRUE) {
          RTOS_ERR err;
          OSFlagPend(&proprietary_event_flags,
                     CHARACTERISTIC_CHANGED,
                     (OS_TICK)0,
                      OS_OPT_PEND_BLOCKING       \
                      + OS_OPT_PEND_FLAG_SET_ANY \
                      + OS_OPT_PEND_FLAG_CONSUME,
                      NULL,
                      &err);

          sl_status_t result;;
                result  =  sl_bt_gatt_server_read_attribute_value(gattdb_prop_char,  0,  16,  data_len,
dataPacket);
          RAIL_WriteTxFifo(railHandle, dataPacket, data_len, true);
          RAIL_StartTx(railHandle, 0, RAIL_TX_OPTIONS_DEFAULT, &txSchedulerInfo);
}
```

8. In `sl_rail_util_on_event()`:
   a. Check for the packet_sent event, and do not forget to yield the radio.

```
static void sl_rail_on_event(RAIL_Handle_t railHandle,
                             RAIL_Events_t events)
{
        if (events & RAIL_EVENT_TX_PACKET_SENT) {
            RAIL_YieldRadio(railHandle);
        }
}
```

5.2 Receiving Proprietary Packets

This example implements a receiver for the transmitter implemented in the previous section. Once a proprietary packet is received, the example updates a characteristic in the local GATT database.

To implement a receiver, use the transmitter project described in the previous section and extend it with the following procedure.

1. Define a new flag for signaling packet reception to the proprietary application.

```
#define PACKET_RECEIVED  ((OS_FLAGS)0x02)
```

2. Create an Rx FIFO. Define the following in *app_proprietary.c*:

```
#define RAIL_RX_FIFO_SIZE (64)
static uint8_t rxFifo[RAIL_RX_FIFO_SIZE];
```

3. In the `app_proprietary_task()` – before the infinite loop:
   a. Set Rx transition in order to automatically restore Rx state after packet reception.
   b. Set the Rx priority lower than the Tx priority.
   c. Start Rx (before the infinite loop!).

```
RAIL_StateTransitions_t  stateTransition = (RAIL_StateTransitions_t){
                                                    .success = RAIL_RF_STATE_RX,
                                                    .error = RAIL_RF_STATE_RX  };
RAIL_SetRxTransitions(railHandle,&stateTransition);
RAIL_SchedulerInfo_t rxSchedulerInfo = (RAIL_SchedulerInfo_t){ .priority = 200 };
RAIL_StartRx(railHandle, 0, &rxSchedulerInfo);
```

4. In the radio event handler, such as `sl_rail_util_on_event()`:
   a. Check if a packet was successfully received.
   b. Copy the packet content to your local Rx FIFO.
   c. Set a flag to notify the proprietary protocol about the new packet.

```
if (events & RAIL_EVENT_RX_PACKET_RECEIVED) {
    RAIL_RxPacketInfo_t packetInfo;
    RTOS_ERR err;

    RAIL_GetRxPacketInfo(railHandle,
             RAIL_RX_PACKET_HANDLE_NEWEST,
             &packetInfo);

    if (packetInfo.packetStatus == RAIL_RX_PACKET_READY_SUCCESS) {
       RAIL_CopyRxPacket(rxFifo,&packetInfo);
       OSFlagPost(&proprietary_event_flags,PACKET_RECEIVED,OS_OPT_POST_FLAG_SET,&err);
    }
}
```

5. Within the infinite loop of the `app_proprietary_task()`:
   a. Check for two event flags: CHARACTERISTIC_CHANGED and PACKET_RECEIVED. You can wait for both of them and then check which one was set.
   b. If the PACKET_RECEIVED flag is set then write the content of the received packet into the local GATT database and
   c. Notify the Bluetooth stack that the value has changed (using a Bluetooth external signal).

```
while (DEF_TRUE) {
      RTOS_ERR err;
      OS_FLAGS active_flags = OSFlagPend (&proprietary_event_flags,
                                          CHARACTERISTIC_CHANGED \
                                          + PACKET_RECEIVED,
                                          (OS_TICK)0,
                                          OS_OPT_PEND_BLOCKING       \
                                          + OS_OPT_PEND_FLAG_SET_ANY \
                                          + OS_OPT_PEND_FLAG_CONSUME,
                                          NULL,
                                          &err);

      if (active_flags & CHARACTERISTIC_CHANGED)
      {
          sl_status_t result;
                result  =  sl_bt_gatt_server_read_attribute_value(gattdb_prop_char,  0,  16,  data_len,
dataPacket);
          RAIL_WriteTxFifo(railHandle, dataPacket 16, true);
          RAIL_StartTx(railHandle, 0, RAIL_TX_OPTIONS_DEFAULT, &txSchedulerInfo);
      }

      if (active_flags & PACKET_RECEIVED)
      {
          sl_bt_gatt_server_write_attribute_value(gattdb_prop_char,0,16,rxFifo);
          sl_bt_external_signal(CHARACTERISTIC_CHANGED);
      }
}
```

6. In `sl_bt_on_event()`:
   a. Add a new event handler for the external signal.
   b. Check if you got a CHARACTERISTIC_CHANGED signal.
   c. Send out a notification.

```
case sl_bt_evt_system_external_signal_id:
   if (bluetooth_evt->data.evt_system_external_signal.extsignals &
                                                          CHARACTERISTIC_CHANGED)
   {
       sl_bt_cmd_gatt_server_send_characteristic_notification(0xff, gattdb_prop_char,
                                                              16, rxFifo, &sent_len);
   }
break;
```

5.3 Light/Switch Example

This section provides details on working with the Light/Switch multiprotocol example code.

5.3.1 Working with the Light/Switch Example

The **Flex (RAIL) - Switch** and **Bluetooth - SoC Ligh /RAIL DMP** 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 *QSG168: Proprietary Flex SDK v3.x Quick-Start Guide*.
- To see details about installing Simplicity Studio and the Bluetooth SDK and building an example application, see *QSG169: Bluetooth® SDK v3.x Quick-Start Guide*.

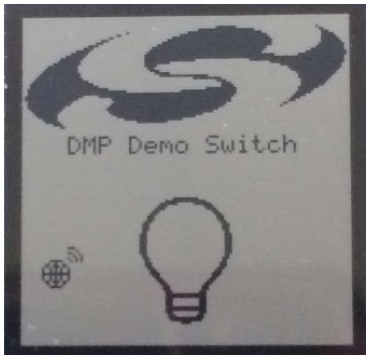
Note: In a demonstration configuration with multiple RAIL/Bluetooth dynamical protocol light devices and a single switch device, unpredictable behavior may occur. We recommend testing with a single light device and a single switch device.

The following summary procedures are provided for your convenience.

5.3.2 Building the RAIL:Switch Application

1. Open Simplicity Studio 5.
2. Select a connected device in the Debug Adapters view.
3. Select File > New > Silicon Labs Project Wizard ...
4. Review the SDK and toolchain, and change as necessary. Click **NEXT**.
5. On the Example Project Selection dialog, filter on Proprietary and select **Flex (RAIL) - Switch**. Click **NEXT**.
6. Name your project. Click [**FINISH**].
7. Either automatically compile and flash using the debug button, or manually compile and then load.

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

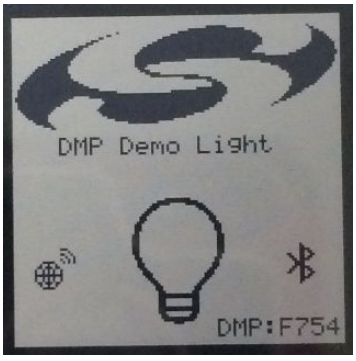


5.3.3 Building the Bluetooth Light Application

The Bluetooth Light application requires the Gecko Bootloader to be loaded on the device. The Gecko Bootloader is loaded when you load the precompiled **SOC-Light-Rail-Dmp** demonstration. Alternatively you can build and load your own Gecko Bootloader combined image (called <projectname>-combined.s37), as described in *UG266: Silicon Labs Gecko Bootloader User's Guide for GSDK 3.2 and Lower* or *UG489: Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher*.

1. Open Simplicity Studio 5.
2. Select the connected device in the Debug Adapters view.
3. Select File > New > Silicon Labs Project Wizard ...
4. Review the SDK and toolchain, and change as necessary. Click **NEXT**.
5. On the Example Project Selection dialog, filter on Bluetooth and select **Soc Light Rail Dmp**. Click **NEXT**.
6. Name your project. Click [**FINISH**].
7. Either automatically compile and flash using the debug button, or manually compile and then load.

Application load success indicators are code-dependent. With the **Bluetooth - SoC Light RAIL DMP** example, the LCD displays a light bulb.



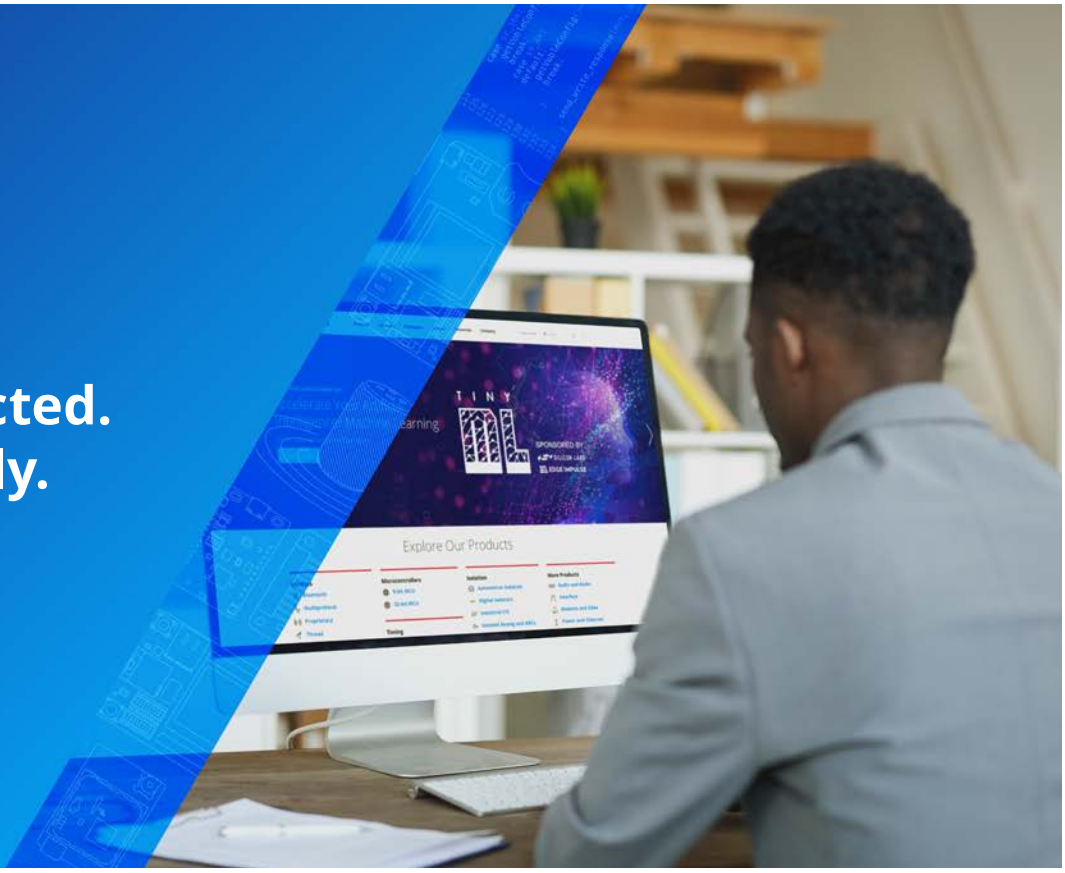
5.3.4 Changing the PHY Configuration

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:

1. Open the **Flex (RAIL) - Switch** project.
2. Open the .slcp file in the project, and click the **Configuration Tools** tab.
3. Click **Open** next to Radio Configurator.
4. Select a new PHY.
5. The new config will be generated into the folder *autogen*, with the names of rail_config.c and rail_config.h.
6. Open the **Bluetooth - SoC Light RAIL DMP** project.
7. Import the modified radio configuration file (radio_settings.radioconf) from the Switch project.
8. Rebuild and flash both projects as you would normally.

Smart. Connected. Energy-Friendly.



IoT Portfolio
www.silabs.com/products

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