AN1134: Dynamic Multiprotocol Development with *Bluetooth*® and Proprietary Protocols on RAIL

This application note provides details on how to develop a multiprotocol application running Bluetooth and a proprietary protocol at the same time. 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 the Light/Switch DMP example demonstrated in *QSG155: Using the Silicon Labs Dynamic Multiprotocol Demonstration Applications* is introduced in more details. For details on Dynamic Multiprotocol Application development that apply to all protocol combinations 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 generic information about the Dynamic Multiprotocol solution, where two protocols are running on the same device in parallel. 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.

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 2.9.0 or higher
- 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 2.7.0 or higher
- Flex SDK version 2.1 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) (required for the RAIL Switch application). See the release notes for the Bluetooth SDK for the required IAR-EWARM version.

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 take into account:

- **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 accuracy**. The consequence of predefined time instances is that ***Bluetooth packets cannot be late*** – their 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 need 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 slave latency parameter to make Bluetooth communication less frequent on the slave 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 1 meg PHY, and 40 µs on 2 meg 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 257 byte payload which takes 2120 µs to be sent on 1 meg PHY and 1060 µs on 2 meg PHY. Along with receiving an empty response packet this takes 2120 + 150 + 80 = ***2350 µs*** on 1 meg PHY and 1060 + 150 + 40 = ***1250 µs*** on 2 meg 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, hence 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.

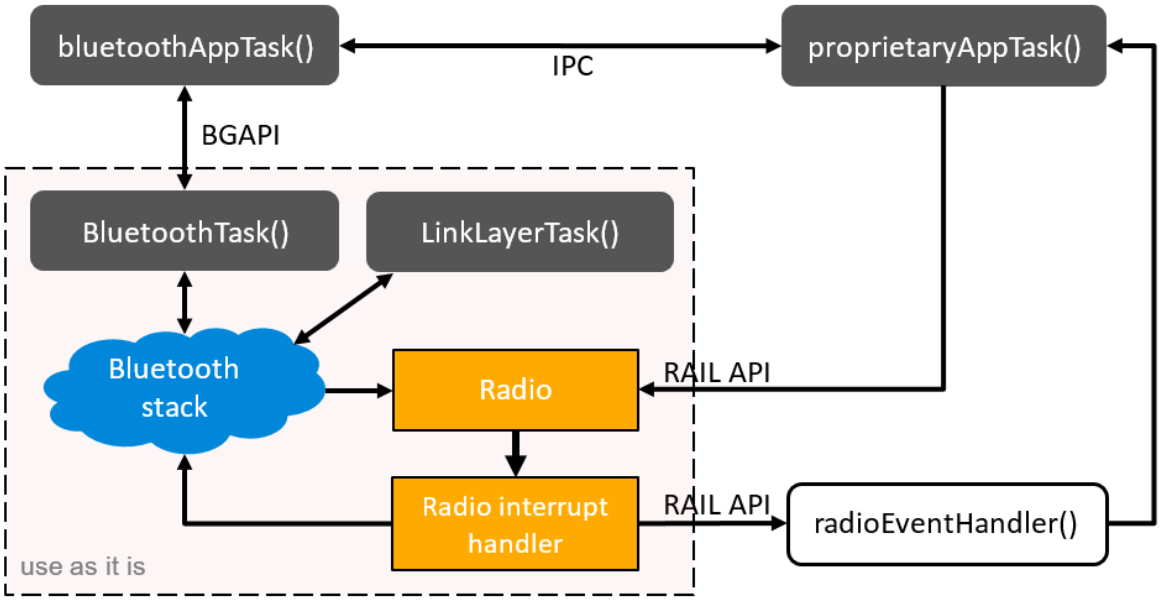
3. Software Architecture of a Bluetooth / Proprietary DMP application

DMP applications are based on Micrium RTOS. The RTOS helps run the Bluetooth and Proprietary protocols in parallel and independently.

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 the `bluetoothAppTask()`. For details please refer to *AN1114: Integrating Silicon Labs Bluetooth® Applications with the Micrium RTOS*.

The proprietary protocol is implemented in the `proprietaryAppTask()`. Unlike Bluetooth, the proprietary protocol can access the radio directly through the RAIL API. RAIL events need a callback function – `radioEventHandler()` – to be defined. This function is called every time a new RAIL event is generated, and can notify the application about the event. Note: `radioEventHandler()` 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).



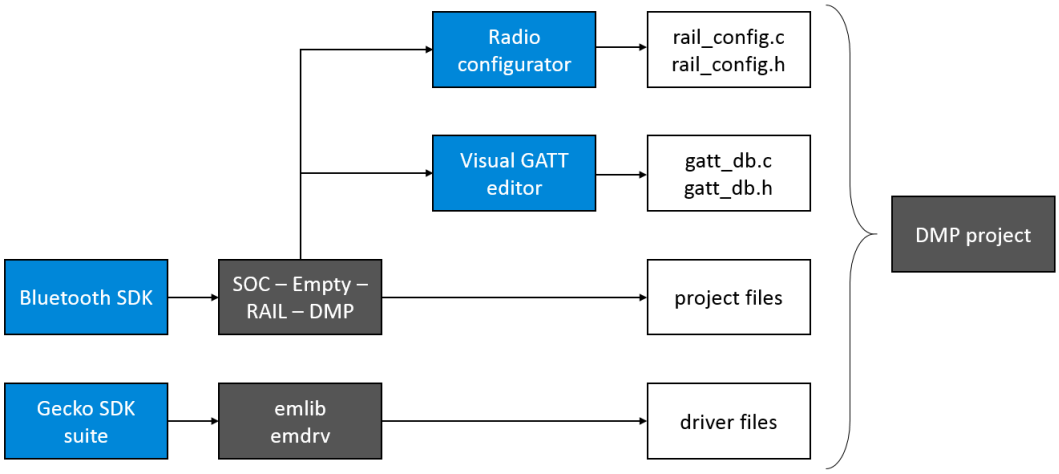
4. Developing a Bluetooth / Proprietary DMP Project

4.1 Create a New Project

Silicon Labs Bluetooth SDK (v2.9 or later) includes the "SOC – Empty – RAIL – DMP" Software Example that should be used as a starting point for every Bluetooth / Proprietary application. This example project:

- Includes the multiprotocol RAIL library
- Includes the Bluetooth library
- Includes the Micrium 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 only thing you have to do is to modify the configurations according to your needs and implement the `bluetoothAppTask()` and the `proprietaryAppTask()`. The GATT database can be configured with the visual GATT editor tool, 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.
2. Select your device on the Devices tab, or on the Solutions tab.
3. Click [**New Project**] in the Launcher perspective of Simplicity Studio, or click **File > New > Project**.
4. Select **Bluetooth SDK**. Click [**Next**].
5. If you have more SDKs installed, select **Bluetooth SDK v2.9.0** or later. Click [**Next**].
6. Select **SOC – Empty – RAIL – DMP** sample application. Click [**Next**].
7. Name your project. Click [**Next**].
8. Check your part number.
9. Select the compiler you want to use. Click [**Finish**].

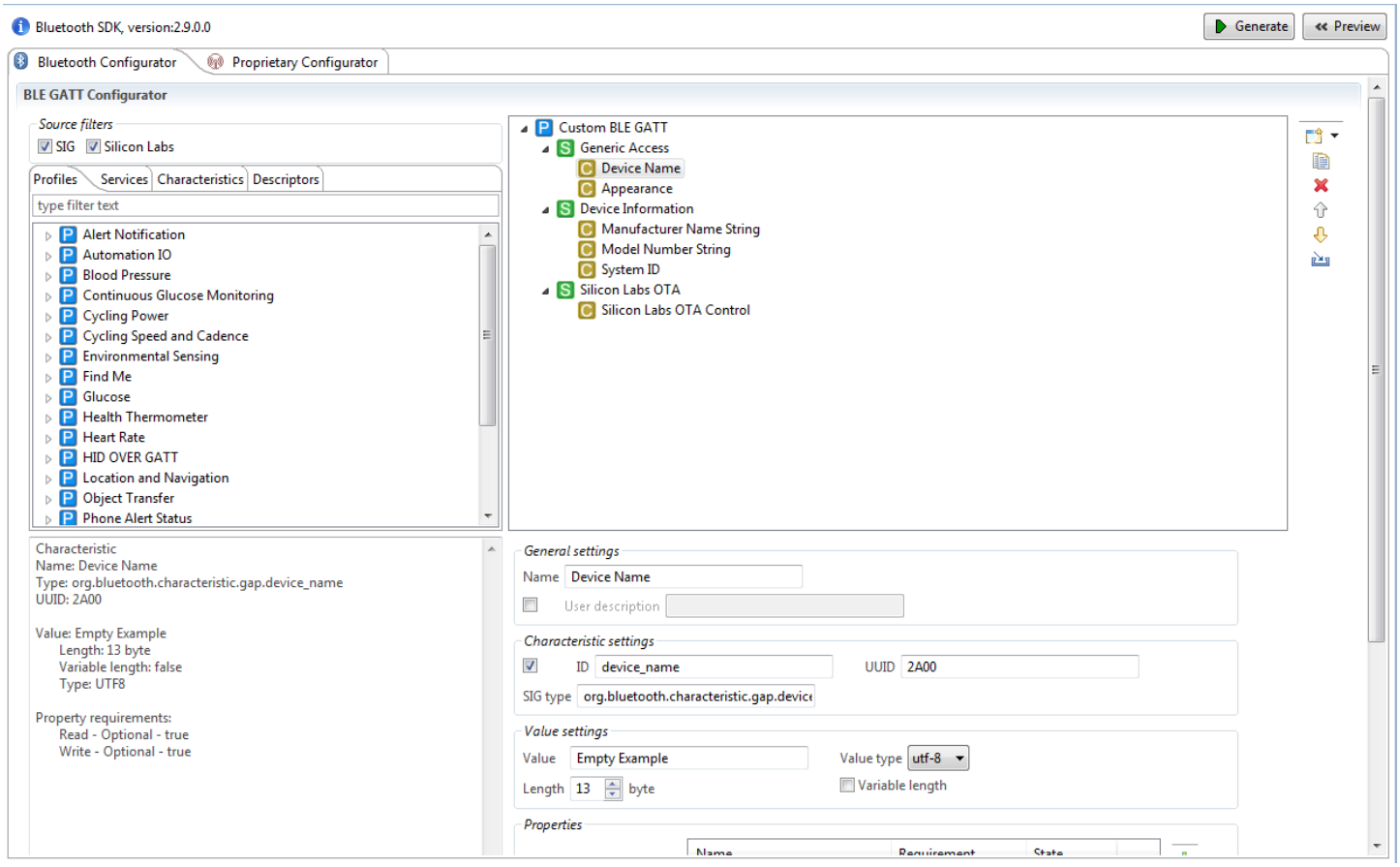
4.2 Configure Bluetooth

Configuring Bluetooth consists of two steps:

- Configuring the local GATT database
- Configuring the Bluetooth stack

To configure the local GATT database, use the Visual GATT editor tool:

1. Open the .isc file in the project (if it is not already open).
2. Click on the Bluetooth Configurator tab.
3. Add your services and characteristics as described in *QSG139: Bluetooth® Development with Simplicity Studio* (or use the default GATT database).
4. Click [**Generate**] to generate gatt.xml, gatt_db.c and gatt_db.h.



To configure the Bluetooth stack:

1. Open main.c.
2. Find the gecko configuration structure (gecko_configuration_t config).
3. Change the config according to your needs. For details see *UG136: Silicon Labs Bluetooth® C Application Developer's Guide* (or use the default configuration).

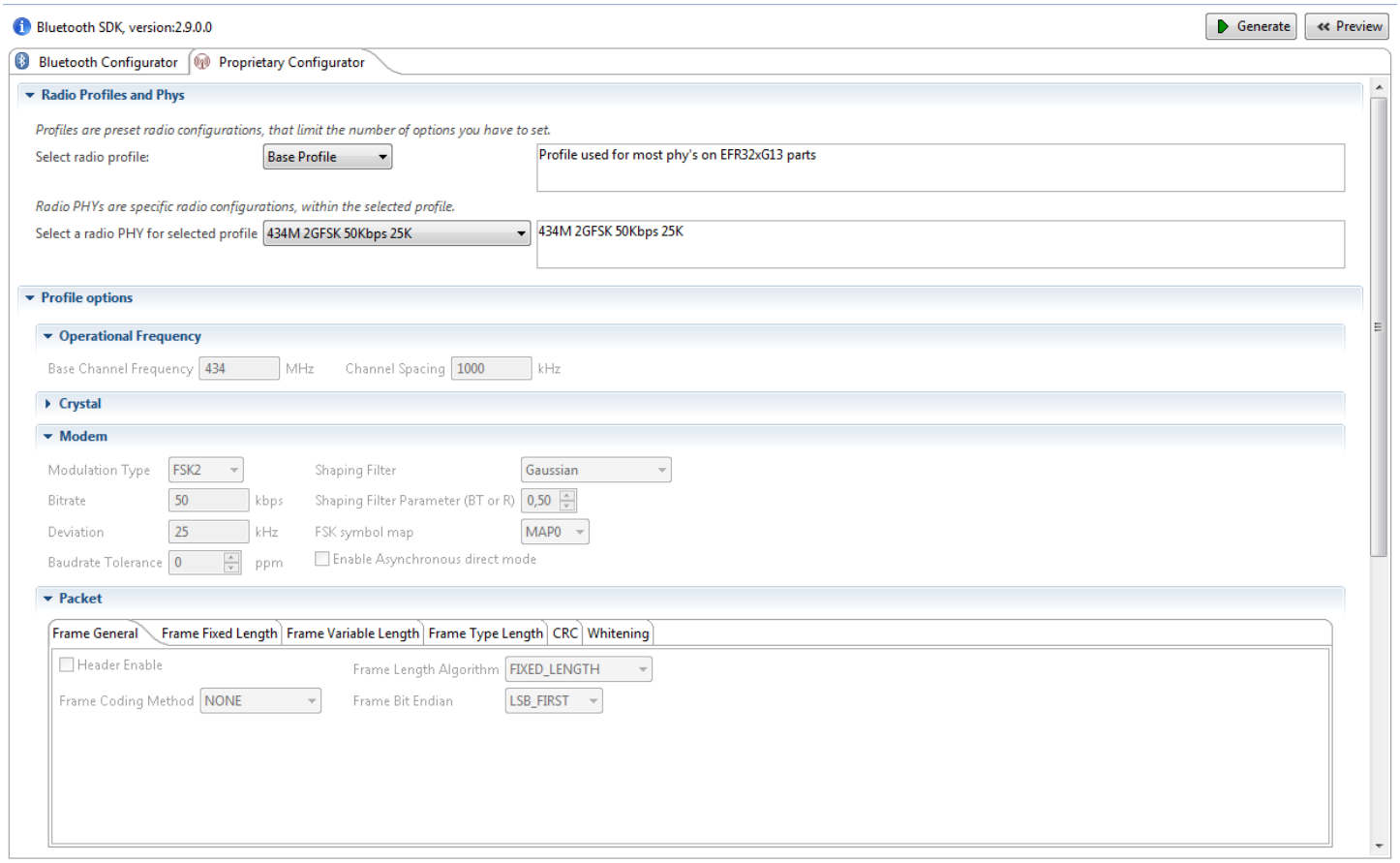
4.3 Configure Proprietary Protocol

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 the Radio Configurator tool:

1. Open the .isc file in the project (if it's not opened yet).
2. Click the Proprietary Configurator tab.
3. Select **Base Profile** from the radio profiles.
4. Select a predefined radio PHY from the list, or select Custom settings, and apply your settings. For details see *AN971: EFR32 Radio Configurator Guide*.
5. Click [**Generate**] to generate rail_config.c and rail_config.h.



To configure RAIL:

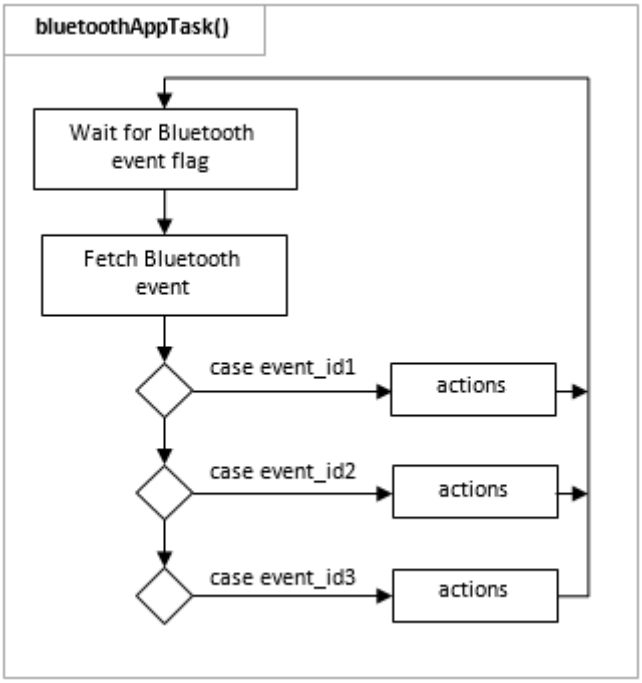
1. Open main.c.
2. Find the `proprietaryAppTask()` function. RAIL init is done here.
3. If you have generated multiple radio configurations, you can select here which one to use, and you can subscribe here for RAIL events.

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 `bluetoothAppTask()`. 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

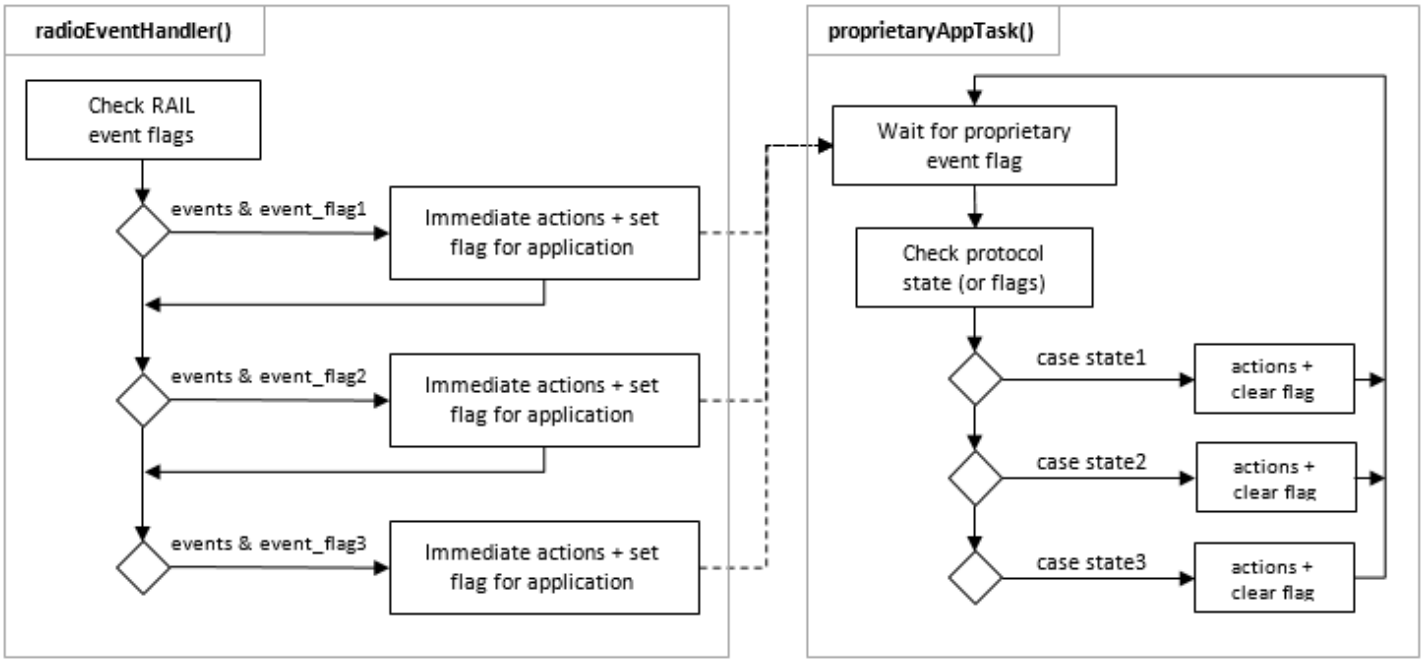
Proprietary application uses RAIL directly:

- RAIL API commands can be called from anywhere.
- RAIL API events have to be handled in the events callback function that was set in `RAIL_Init()`.

By default the events callback function is set to `radioEventHandler()` and an empty `radioEventHandler()` function is implemented in the DMP empty example. This function is called every time a new radio event 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 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.

The main loop to process the radio events is to be implemented in the `proprietaryAppTask()`, which runs parallel to the `bluetoothAppTask()`. It is the developer's job to decide how to communicate between the `radioEventHandler()` and the `proprietaryAppTask()`, but in general it is recommended to 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 Bluetooth task from the proprietary task, the easiest solution is to generate an external Bluetooth event. `gecko_external_signal()` puts a new (external) event in the event queue of the Bluetooth stack. Then you can simply use the gecko_evt_system_external_signal_id event ID in the `bluetoothAppTask()` to check if an external event was received. A 32-bit bitfield can be used to differentiate 32 external events. For more information refer to *UG136: Silicon Labs Bluetooth® C Application Developer's Guide*.

To notify the proprietary task from the Bluetooth task, the easiest way is to set an RTOS flag, in the same way you set flags in the `radioEventHandler()` to notify the application about a new RAIL event.

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 4.1 Create a New Project.
2. Add a new characteristic to the GATT database (as described in QSG139: Bluetooth® Development with Simplicity Studio) 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. Click [**Generate**] to generate the GATT database.
4. Create a new OS Event Flag group to enable communication between the Bluetooth stack and the proprietary protocol.
   a. Declare the global variable for the flag group, and define the flags in the header part of main.c:

```
OS_FLAG_GRP  proprietary_event_flags;
#define CHARACTERISTIC_CHANGED  ((OS_FLAGS)0x01)
```

   b. Find exMainStartTask() in main.c, and initialize the flag group in it.

```
OSFlagCreate(&proprietary_event_flags, "Prop. flags", (OS_FLAGS)0, &err);
```

5. Create a Tx FIFO. Define the following in the header part of main.c:

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

6. In the `bluetoothAppTask()`:
   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 gecko_evt_gatt_server_attribute_value_id:
if (bluetooth_evt->data.evt_gatt_server_attribute_value.attribute == gattdb_prop_char)
   {
              OSFlagPost(&proprietary_event_flags,
                                 CHARACTERISTIC_CHANGED,
                                 OS_OPT_POST_FLAG_SET,
                                 &err);
   }
break;
```

7. In the `proprietaryAppTask()` – 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 };
```

8. Within the infinite loop of the `proprietaryAppTask()`:
   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);

          struct gecko_msg_gatt_server_read_attribute_value_rsp_t *result;
          result = gecko_cmd_gatt_server_read_attribute_value(gattdb_prop_char,0);
          RAIL_WriteTxFifo(railHandle, &(result->value.data[0]), 16, true);
          RAIL_StartTx(railHandle, 0, RAIL_TX_OPTIONS_DEFAULT, &txSchedulerInfo);
}
```

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

```
static void radioEventHandler(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 the header part of main.c:

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

3. In the `proprietaryAppTask()` – 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 `radioEventHandler()`:
   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 `proprietaryAppTask()`:
   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)
      {
          struct gecko_msg_gatt_server_read_attribute_value_rsp_t *result;
          result = gecko_cmd_gatt_server_read_attribute_value(gattdb_prop_char,0);
          RAIL_WriteTxFifo(railHandle, &(result->value.data[0]), 16, true);
          RAIL_StartTx(railHandle, 0, RAIL_TX_OPTIONS_DEFAULT, &txSchedulerInfo);
      }

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

6. In the `bluetoothAppTask()`:
   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 gecko_evt_system_external_signal_id:
   if (bluetooth_evt->data.evt_system_external_signal.extsignals &
                                                        CHARACTERISTIC_CHANGED)
   {
       gecko_cmd_gatt_server_send_characteristic_notification(0xff, gattdb_prop_char,
                                                              16, rxFifo);
   }
break;
```

6. Light/Switch Example

This section provides details on working with the Light/Switch example code that results in the example user interface documented in *QSG155: Using the Silicon Labs Dynamic Multiprotocol Demonstration Applications*.

6.1 Working with the Light/Switch Example

To work with Light/Switch dynamic multiprotocol example 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*.

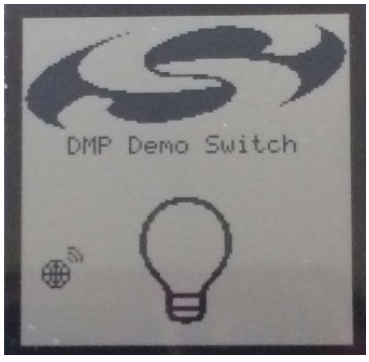
Note: In a demonstration configuration with multiple RAIL/Bluetooth dynamical protocol light devices and a single switch device, unpredictable behavior may occur. We recommend testing with a single light device and a single switch device.

The following summary procedures are provided for your convenience.

6.1.1 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.

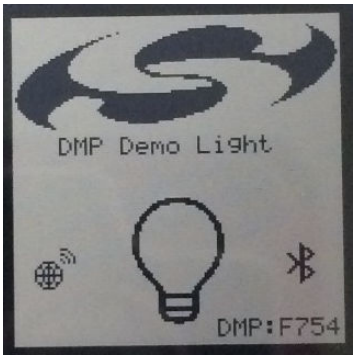


6.1.2 Building the Bluetooth Light Application

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.

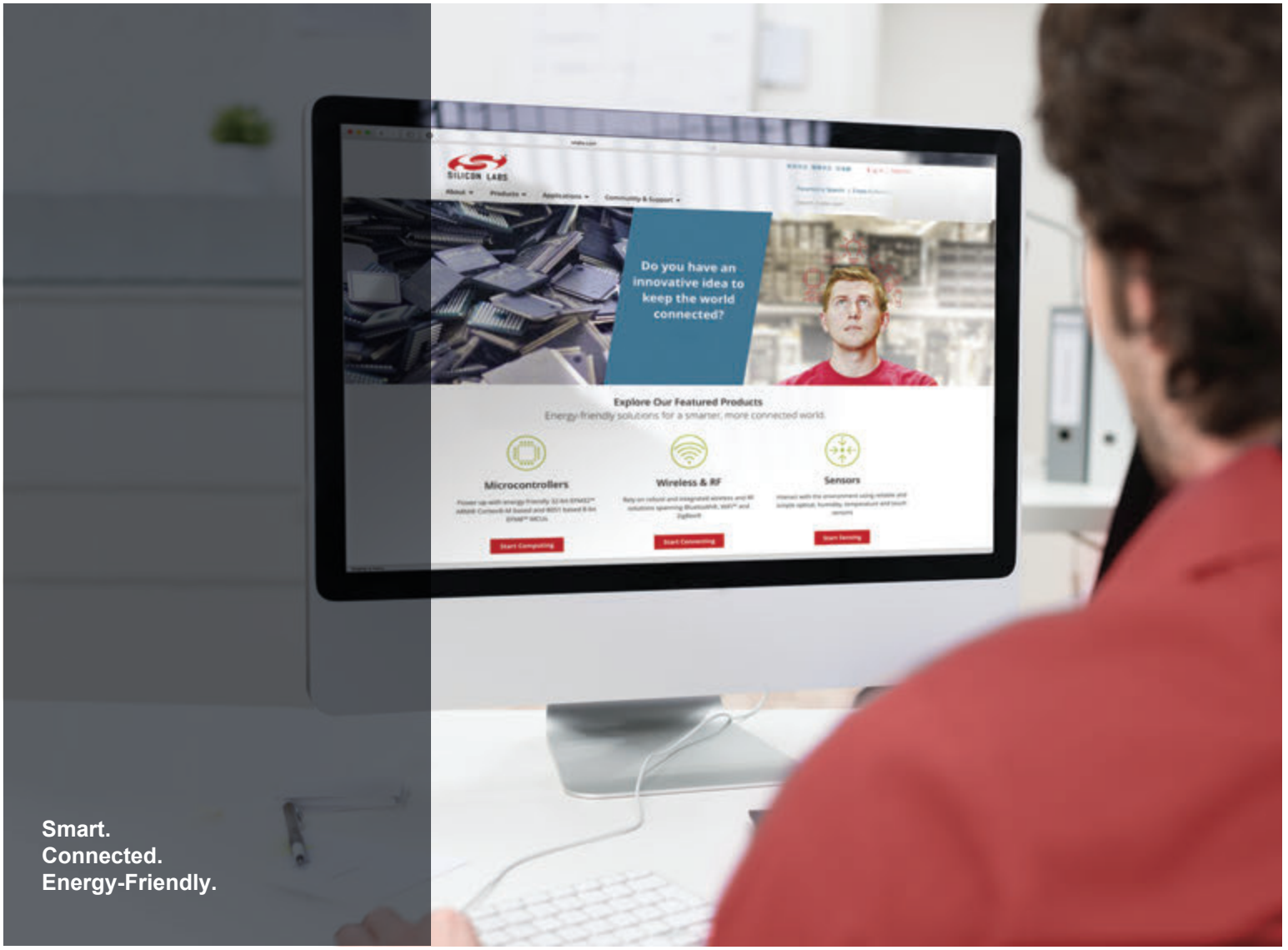


6.1.3 Changing the PHY Configuration

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:

1. Open the **RAIL:Switch** project.
2. Open the .isc file in the project.
3. Select the Radio Configuration tab.
4. Select a new PHY.
5. Click [**Generate**] to generate new rail_config.c and rail_config.h.
6. Open the **SOC - Light - RAIL - DMP** project.
7. Copy rail_config.c and rail_config.h from the RAIL:Switch project to the SOC - Light - RAIL – DMP project.
8. Make sure that you overwrite the old rail_config.c and rail_config.h files.
9. Rebuild and flash both projects as you would normally.



Products
www.silabs.com/products

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

SILICON LABS®

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

http://www.silabs.com