UG295: Silicon Labs *Bluetooth*® Mesh C Application Developer's Guide for SDK v2.x

This document is an essential reference for anyone developing C-based applications for Silicon Labs Wireless Gecko products using the Silicon Labs Bluetooth Mesh stack. This is a companion to *UG434: Silicon Labs Bluetooth C Application Developers Guide for SDK v3.x* and contains content specific to mesh application development. The guide covers both Bluetooth mesh stack architecture, application development flow, use and limitations of the MCU core and peripherals, stack configuration options, and the stack's resource usage. This version applies to the Silicon Labs Bluetooth Mesh SDK version 2.x.

The purpose of the document is to capture and fill in the blanks between the Bluetooth Mesh Stack API reference, Gecko SDK API reference, and Wireless Gecko reference manuals, when developing Bluetooth mesh applications for the Wireless Geckos. This document exposes details that will help developers make the most out of the available hardware resources.

KEY POINTS

- Bluetooth mesh stack and chip configuration
- Interaction with the Bluetooth LE stack
- Event and sleep management
- Resource usage and availability
- Radio State Monitoring

Table of Contents

1 Introduction 1
1.1 About This Version 1
1.2 Prerequisites 1
2 Application Development Flow 2
2.1 Application Build Flow 3
2.2 Bluetooth Mesh API documentation 3
2.3 Bluetooth Mesh Application Build Flow 4
3 Project Structure 6
3.1 Bluetooth Mesh Library Files 6
3.1.1 Stack 7
3.1.2 Node 7
3.1.3 Provisioner 9
3.1.4 GATT Database 10
3.1.5 Bluetooth Mesh Device Composition Data 11
3.1.6 RTOS Support 11
4 Bluetooth Mesh Stack Event Handling 12
4.1 Bluetooth LE versus Bluetooth Mesh Event 12
5 NVM Layout 13
6 Bluetooth Mesh Features 14
6.1 Proxy 14
6.2 Relay 14
6.3 Friend 14
6.4 Low Power Node 15
7 Bluetooth Mesh Stacks and Wireless Gecko Configuration and Resources 17
7.1 Wireless Gecko MCU and Peripherals Configuration 17
7.2 Wireless Gecko Resources 17
7.2.1 Internal Flash and SRAM 18
7.2.2 Monitoring Radio RX and TX State Using PRS (Peripheral Reflex System) 18
8 Documentation 21

1 Introduction

This document is a C developer's guide for the Silicon Labs Bluetooth Mesh stack. It covers various aspects of development and is an important reference for anyone developing in C for Wireless Gecko products that are running the Bluetooth stack.

The document covers the following topics:

- Section 2 Application Development Flow discusses the application development flow.
- Section 3 Project Structure reviews project structure.
- Section 4 Bluetooth Mesh Stack Event Handling is an important piece for everyone developing with the Silicon Labs Bluetooth stack, as it explains how the application runs in sync with the stack in an event-based architecture.
- Section 5 NVM Layout describes memory allocation for Bluetooth LE and mesh.
- Section 6 Bluetooth Mesh Features reviews functionality provided by Bluetooth mesh features.
- Section 7 Bluetooth Mesh Stacks and Wireless Gecko Configuration and Resources touches on the topics of peripherals and the chipset resources, covers what is reserved for the stack usage, how interrupts should be handled, and the stack's memory footprint and available memory for the application. It also covers radio TX/RX monitoring.

1.1 About This Version

The current version of Silicon Labs' Bluetooth Mesh SDK is 2.1.x. Currently supported compilers and IDE version are:

- IDE: Simplicity Studio 5.0.0 or newer.
- Compiler: IAR v8.50.9 and GCC 10.2.0.

1.2 Prerequisites

This document assumes the current version of Silicon Labs' Bluetooth SDK has been properly installed to the development machine (Windows, MAC OSX, or Linux), and that you are familiar with the *QSG176: Silicon Labs Bluetooth® Mesh SDK v2.x Quick-Start Guide* and with the SDK's examples. Also, you should have a basic understanding of Bluetooth technology. For more information, see *UG103.14: Bluetooth Technology Fundamentals*.

2 Application Development Flow

The following figure describes the high-level firmware structure. The developer creates an application on top of the stack, which Silicon Labs provides as a precompiled object-file, enabling the Bluetooth mesh connectivity for the end-device.

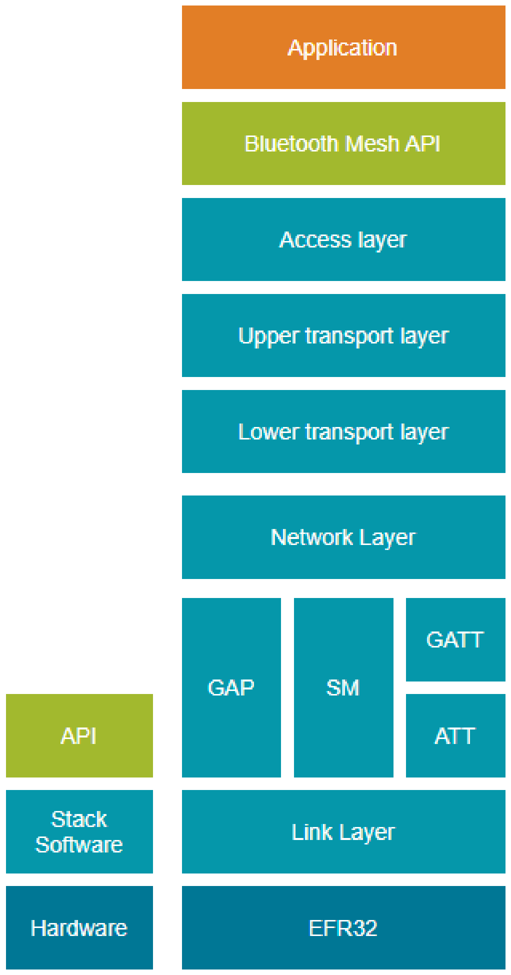


Figure 2.1. Firmware Architecture

The Bluetooth mesh stack contains the following blocks.

- **Bootloader**—The Gecko Bootloader is not part of the stack but is provided with the Bluetooth SDK. See *UG266: Gecko Bootloader User Guide* and *AN1086: Using the Gecko Bootloader with Silicon Labs Bluetooth Applications* for more information. For information on bootloading in general, see *UG103.06: Bootloading Fundamentals*.
- **Bluetooth stack**—Bluetooth functionality consisting of link layer, generic access profile, security manager, attribute protocol, and generic attribute profile.
- **Bluetooth AppLoader**—An application that starts after the bootloader. It checks if the user application is valid and, if it is, starts the application. If the application image is not valid, AppLoader starts the Bluetooth Low Energy OTA process to try to receive a valid application image. This requires using the Gecko Bootloader.
- **Bluetooth Mesh Stack** -The Bluetooth mesh functionality consisting of the network layer, lower and upper transport layer, and access layer. Models are also provided as part of the stack.

2.1 Application Build Flow

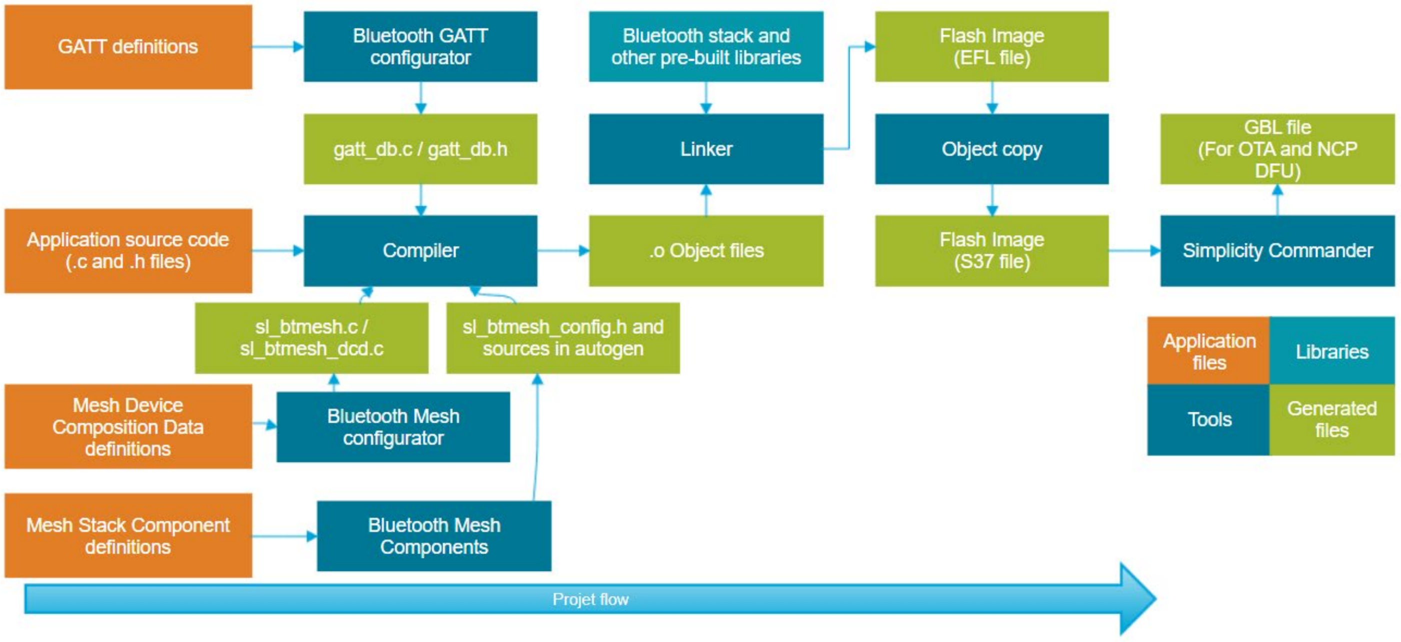

GATT definitions
Bluetooth GATT configurator
Bluetooth stack and other pre-built libraries
Flash Image (EFL file)
gatt_db.c / gatt_db.h
Linker
Object copy
GBL file (For OTA and NCP DFU)
Application source code (.c and .h files)
Compiler
.o Object files
Flash Image (S37 file)
Simplicity Commander
sl_btmesh.c / sl_btmesh_dcd.c
sl_btmesh_config.h and sources in autogen
Application files
Libraries
Tools
Generated files
Mesh Device Composition Data definitions
Bluetooth Mesh configurator
Mesh Stack Component definitions
Bluetooth Mesh Components
Projet flow

Figure 2.2. Bluetooth Mesh Project Build Flow

Building a Bluetooth mesh project starts by selecting the right base project that is to be generated and adding any extra required components. Models and elements can be added to the project using the Bluetooth Mesh Configurator. If the GATT bearer is used, the Bluetooth mesh services and characteristics can be viewed just like any other GATT services and characteristics in the Bluetooth GATT Configurator. For more information on the Bluetooth GATT Configurator and Bluetooth application development, see *QSG169: Bluetooth® SDK v3.x Quick-Start Guide*.

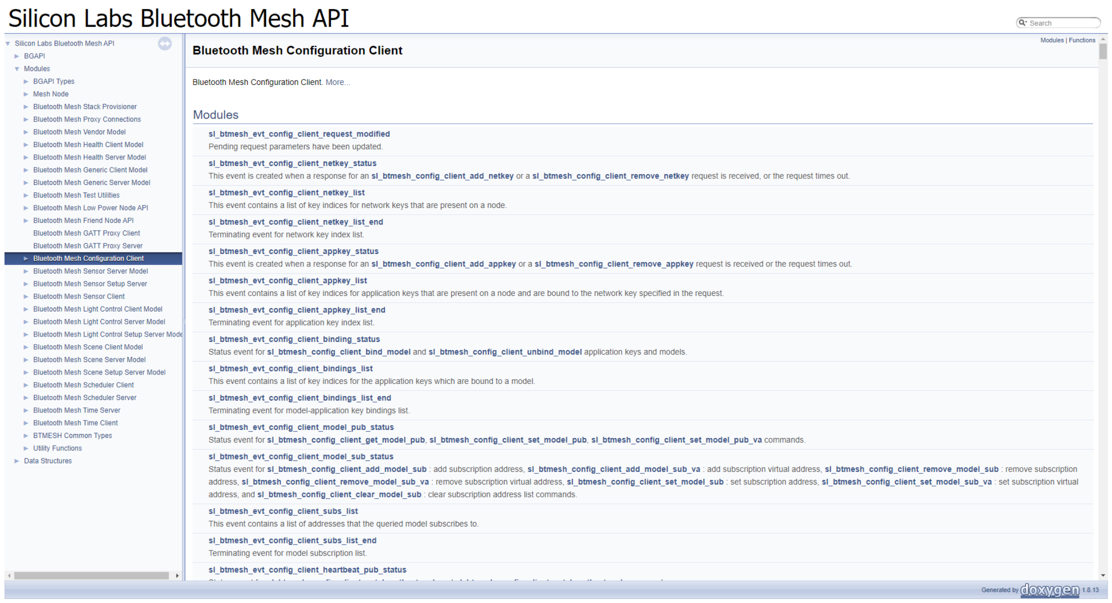
Compiling the project generates an object file, which is then linked with the pre-compiled libraries provided in the SDK. The output of the linking is a flash image that can be programmed to the supported Wireless Gecko devices.

2.2 Bluetooth Mesh API documentation

The Bluetooth Mesh API documentation can be found in HTML format along with all Application Notes and User Guide pdf files in the following default installation directory:

SimplicityStudio\v5\developer\sdks\gecko_sdk_suite\<version>\app\bluetooth\documentation

The API reference is under API_BLUETOOTH_MESH_HTML.



Silicon Labs Bluetooth Mesh API

- Silicon Labs Bluetooth Mesh API
  - BGAPI
  - Modules
    - BGAPI Types
    - Mesh Node
    - Bluetooth Mesh Stack Provisioner
    - Bluetooth Mesh Proxy Connections
    - Bluetooth Mesh Vendor Model
    - Bluetooth Mesh Health Client Model
    - Bluetooth Mesh Health Server Model
    - Bluetooth Mesh Generic Client Model
    - Bluetooth Mesh Generic Server Model
    - Bluetooth Mesh Test Utilities
    - Bluetooth Mesh Low Power Node API
    - Bluetooth Mesh Friend Node API
    - Bluetooth Mesh GATT Proxy Client
    - Bluetooth Mesh GATT Proxy Server
    - Bluetooth Mesh Configuration Client
    - Bluetooth Mesh Sensor Server Model
    - Bluetooth Mesh Sensor Setup Server
    - Bluetooth Mesh Sensor Client
    - Bluetooth Mesh Light Control Client Model
    - Bluetooth Mesh Light Control Server Model
    - Bluetooth Mesh Light Control Setup Server Mode
    - Bluetooth Mesh Scene Client Model
    - Bluetooth Mesh Scene Server Model
    - Bluetooth Mesh Scene Setup Server Model
    - Bluetooth Mesh Scheduler Client
    - Bluetooth Mesh Scheduler Server
    - Bluetooth Mesh Time Server
    - Bluetooth Mesh Time Client
    - BTMESH Common Types
    - Utility Functions
  - Data Structures

Modules | Functions

Bluetooth Mesh Configuration Client

Bluetooth Mesh Configuration Client. More...

Modules

sl_btmesh_evt_config_client_request_modified
Pending request parameters have been updated.

sl_btmesh_evt_config_client_netkey_status
This event is created when a response for an **sl_btmesh_config_client_add_netkey** or a **sl_btmesh_config_client_remove_netkey** request is received, or the request times out.

sl_btmesh_evt_config_client_netkey_list
This event contains a list of key indices for network keys that are present on a node.

sl_btmesh_evt_config_client_netkey_list_end
Terminating event for network key index list.

sl_btmesh_evt_config_client_appkey_status
This event is created when a response for an **sl_btmesh_config_client_add_appkey** or a **sl_btmesh_config_client_remove_appkey** request is received or the request times out.

sl_btmesh_evt_config_client_appkey_list
This event contains a list of key indices for application keys that are present on a node and are bound to the network key specified in the request.

sl_btmesh_evt_config_client_appkey_list_end
Terminating event for application key index list.

sl_btmesh_evt_config_client_binding_status
Status event for **sl_btmesh_config_client_bind_model** and **sl_btmesh_config_client_unbind_model** application keys and models.

sl_btmesh_evt_config_client_bindings_list
This event contains a list of key indices for the application keys which are bound to a model.

sl_btmesh_evt_config_client_bindings_list_end
Terminating event for model-application key bindings list.

sl_btmesh_evt_config_client_model_pub_status
Status event for **sl_btmesh_config_client_get_model_pub**, **sl_btmesh_config_client_set_model_pub**, **sl_btmesh_config_client_set_model_pub_va** commands.

sl_btmesh_evt_config_client_model_sub_status
Status event for **sl_btmesh_config_client_add_model_sub** : add subscription address, **sl_btmesh_config_client_add_model_sub_va** : add subscription virtual address, **sl_btmesh_config_client_remove_model_sub** : remove subscription address, **sl_btmesh_config_client_remove_model_sub_va** : remove subscription virtual address, **sl_btmesh_config_client_set_model_sub** : set subscription address, **sl_btmesh_config_client_set_model_sub_va** : set subscription virtual address, and **sl_btmesh_config_client_clear_model_sub** : clear subscription address list commands.

sl_btmesh_evt_config_client_subs_list
This event contains a list of addresses that the queried model subscribes to.

sl_btmesh_evt_config_client_subs_list_end
Terminating event for model subscription list.

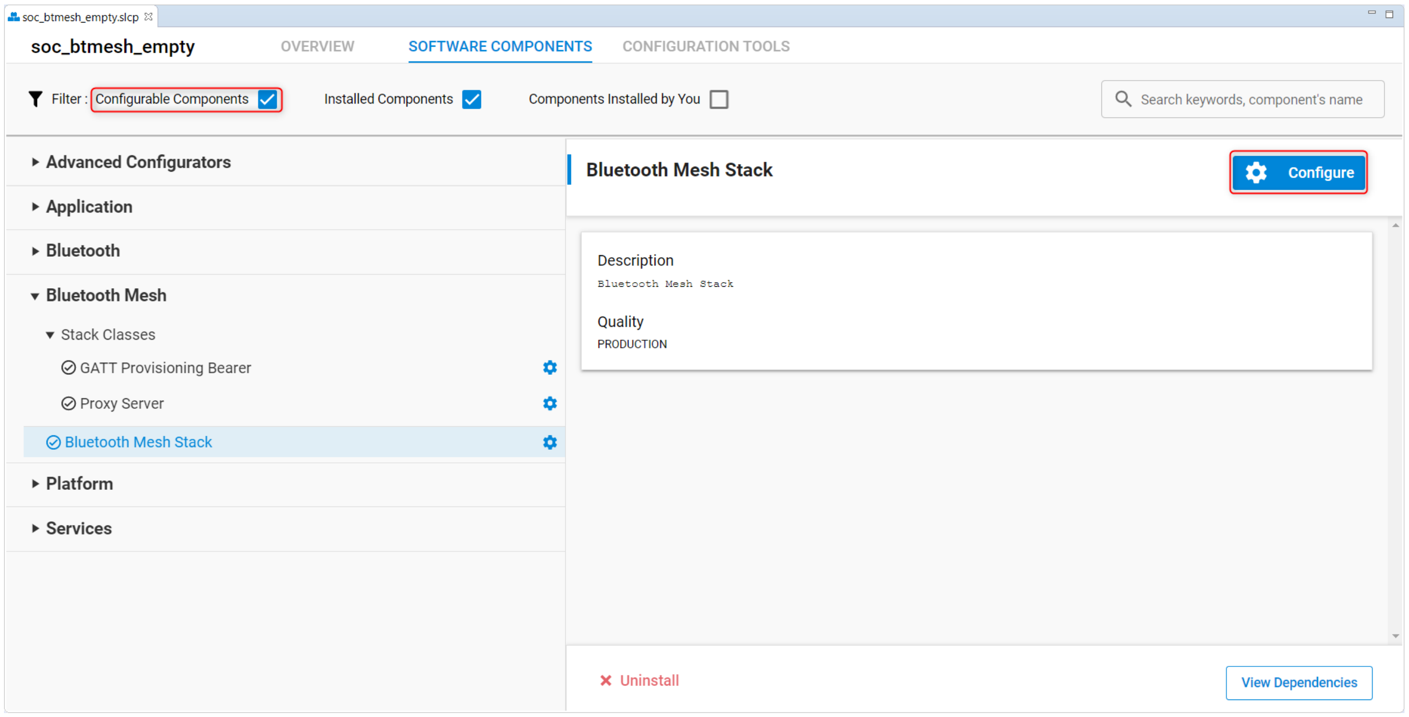
sl_btmesh_evt_config_client_heartbeat_pub_status

Generated by doxygen 1.8.13

Figure 2.3. Bluetooth Mesh API Reference

2.3 Bluetooth Mesh Application Build Flow

Following the Simplicity Studio v5 software approach, the Bluetooth mesh stack is configured using components. Typically, items such as models, stack parameters, and features can be tuned through the software components menu. All component dependencies are handled internally by Simplicity Studio. Components and models can be configured, added, or removed



soc_btmesh_empty.slcp

soc_btmesh_empty OVERVIEW SOFTWARE COMPONENTS CONFIGURATION TOOLS

Filter : Configurable Components ✓ Installed Components ✓ Components Installed by You ☐

Search keywords, component's name

- Advanced Configurators
- Application
- Bluetooth
- Bluetooth Mesh
  - Stack Classes
    - GATT Provisioning Bearer
    - Proxy Server
  - Bluetooth Mesh Stack
- Platform
- Services

Bluetooth Mesh Stack Configure

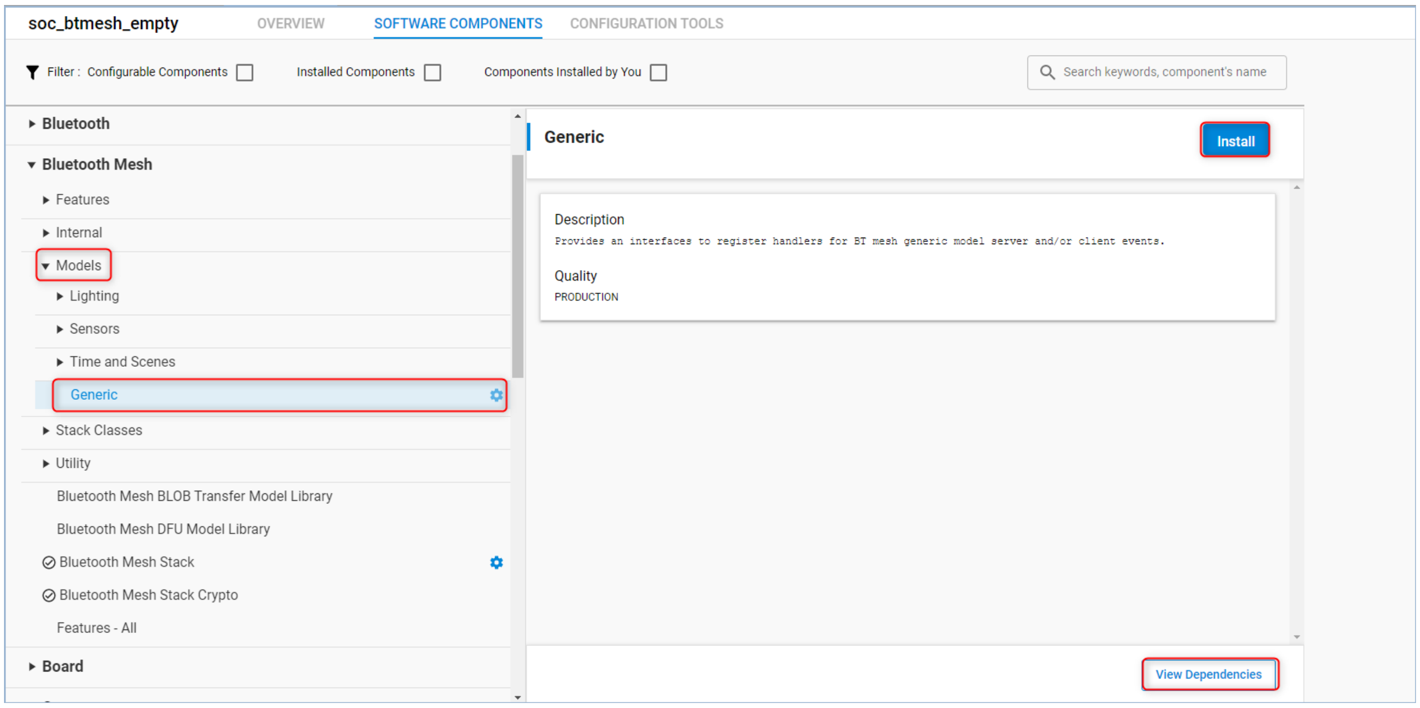
Description
`Bluetooth Mesh Stack`

Quality
PRODUCTION

Uninstall View Dependencies

Figure 2.4. Bluetooth Mesh Configurable Component

Installing a model is shown in the following figure.



Description
Provides an interfaces to register handlers for BT mesh generic model server and/or client events.

Quality
PRODUCTION

Figure 2.5. Bluetooth Mesh Model Installation

3 Project Structure

This section explains the application project structure and the mandatory and optional resources available in the project.

A Bluetooth mesh project is a collection of C source and header files that is built using makefiles. The Simplicity Studio 5 installation generates build files and either generates, copies, or links all SDK or component source files, based on the selection made during project creating. After a project is created, the following directories are created:

- The config directory - This directory is autogenerated and aggregates the component configuration files. Files are all headers and contain macros that are specific to each component. The UI tools used to generate the GATT database as well as the DCD for Bluetooth mesh are in this directory.
- The autogen directory – This directory aggregates the C code generated by Simplicity Studio and the SDK. It is a mix of header and source files that constitute the skeleton of the project. Silicon Labs recommends that you do not edit the files in this directory, as they will be overwritten the next time files are generated.
- The gecko_sdk_3.x directory – This directory copies or links to the SDK resources. Only sources that are relevant to your project and how it is configured are copied or linked.
- The GNU ARM Vxyz – Debug/Release directory – This is the build directory when the GCC compiler is used.

Application code is implemented at the root of the project in app.c/h and main.c. The following table shows the typical layout of a project:

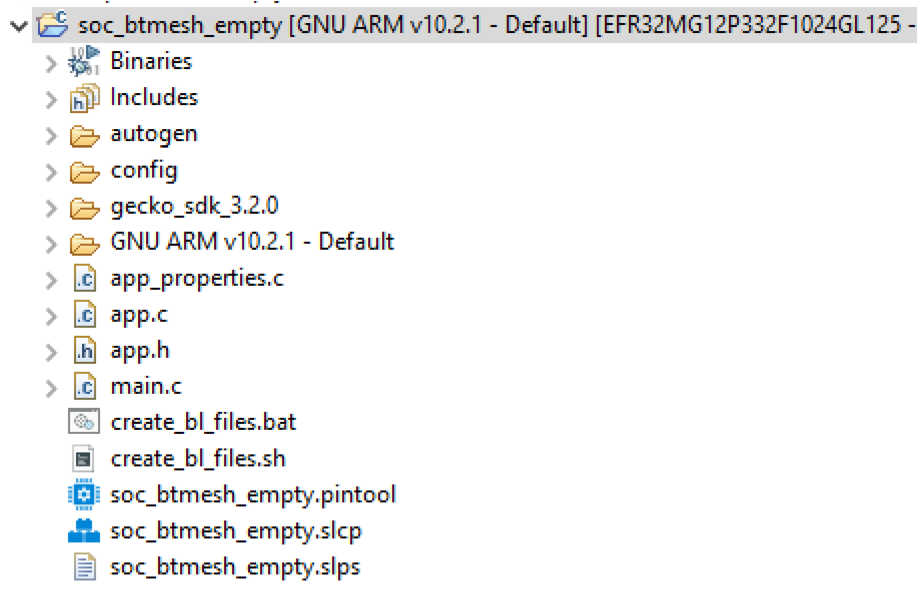

```
soc_btmesh_empty [GNU ARM v10.2.1 - Default] [EFR32MG12P332F1024GL125 -
  Binaries
  Includes
  autogen
  config
  gecko_sdk_3.2.0
  GNU ARM v10.2.1 - Default
  app_properties.c
  app.c
  app.h
  main.c
  create_bl_files.bat
  create_bl_files.sh
  soc_btmesh_empty.pintool
  soc_btmesh_empty.slcp
  soc_btmesh_empty.slps
```

Figure 3.1. Bluetooth Mesh Project Directories

3.1 Bluetooth Mesh Library Files

The Bluetooth stack libraries are:

- **binapploader.o**: Binary image of the Bluetooth AppLoader, provides the optional Bluetooth LE Over-the-Air (OTA) functionality.
- **binapploader_nvm3.o**: Binary image of the Bluetooth AppLoader for Series 1 with NVM3 support.
- **libbluetooth.a**: Bluetooth stack library.
- **libnvm3_CMxx_gcc.a**: NVM3 functionality for Bluetooth LE and Bluetooth mesh stacks. NVM3 is the unique memory management system used for non-volatile memory. For more information how to use NVM3, see *AN1135: Using Third Generation Non-Volatile Memory (NVM3) Data Storage*.
- **libbluetooth_mesh.a**: This library includes the radio driver layer and the Bluetooth LE stack, with the Bluetooth mesh stack built on top of it.

RAIL (Radio Application Interface Layer). The Bluetooth LE and mesh stacks use RAIL to access the radio. RAIL libraries are linked to the Bluetooth mesh stack under libbluetooth_mesh.a. RAIL has separate libraries for each device family and for single- and multi-protocol environments. RAIL libraries are provided in the Gecko Platform. For more information refer to *UG103.13: RAIL Fundamentals* and other RAIL documentation.

EMLIB and EMDRV. The Bluetooth LE and mesh stacks use EMLIB and EMDRV libraries to access EFR32 hardware. EMLIB and EMDRV peripheral libraries are provided in source code and they must be included in the project. EMLIB and EMDRV are part of the Gecko Platform. For more details on EMLIB and EMDRV, refer to the Gecko Bootloader API reference in <Simplicity Studio Gecko SDK>\platform\bootloader\documentation\Gecko_Bootloader_API_Reference\index.html, along with the documentation in their respective folders under <Simplicity Studio Gecko SDK>\platform\.

mbedTLS. The mbedTLS security library is a C library that implements cryptographic primitives, X.509 certificate manipulation and the SSL/TLS and DTLS protocols. Its small code footprint makes it suitable for embedded systems.

Among other header files generated by the SDK, the following defines the APIs for both the Bluetooth mesh and LE stack. These files serve two purposes: first they contain the actual Bluetooth LE and mesh stack API and the commands and events for the stack, and second they provide configuration, event, and sleep management API to the Bluetooth LE and mesh stack.

sl_btmesh_api.h. This file is part of the SDK directory and defines the Bluetooth mesh API available to the user. It contains all routine definitions as well as types, structures and event definitions needed in order to write a Bluetooth mesh application.

sl_bt_api.h. This file is part of the SDK directory and defines the API available to the user. It contains all routine definitions as well as types, structures and event definitions needed in order to write a Bluetooth LE application.

Note that an application may use both APIs if the developer wished to use both Bluetooth LE and mesh in the project.

3.1.1 Stack

The Bluetooth mesh stack initialization is autogenerated and takes place in sl_btmesh.c under the autogen directory. Sequentially, in the user application code, a system init function `sl_system_init()` is called in main.c. This function is defined in the generated file sl_system_init.c. The stack is then initialized from there.

```
void sl_system_init(void)
{
  sl_platform_init();
  sl_driver_init();
  sl_service_init();
  sl_stack_init();
  sl_internal_app_init();
}
```

The `sl_stack_init()` function initializes the radio transmitter/receiver and makes the call to both the Bluetooth LE and mesh stack initialization functions (sl_event_handler.c).

```
void sl_stack_init(void)
{
  sl_rail_util_pa_init();
  sl_rail_util_pti_init();
  sl_bt_init();
  sl_btmesh_init();
}
```

Additionally, the default Bluetooth LE stack configuration structure and macros can be found under the config directory, in the sl_bluetooth_config.h file. The content of that file is generated by the SDK via the Bluetooth LE and mesh components. For more information on how to configure the Bluetooth mesh stack, refer to *UG472: Bluetooth® Mesh Stack and Bluetooth® Mesh Configurator User's Guide for SDK v2.x*.

```
static sl_bt_configuration_t config = SL_BT_CONFIG_DEFAULT;
sl_status_t err = sl_bt_init_stack(&config);
```

This function takes a single argument - a pointer to a sl_bt_configuration_t struct. Its purpose is to configure and initialize the Bluetooth stack with the parameters provided in the struct. More information on the sl_bt_init_stack() routine is available in the HTML API documentation.

3.1.2 Node

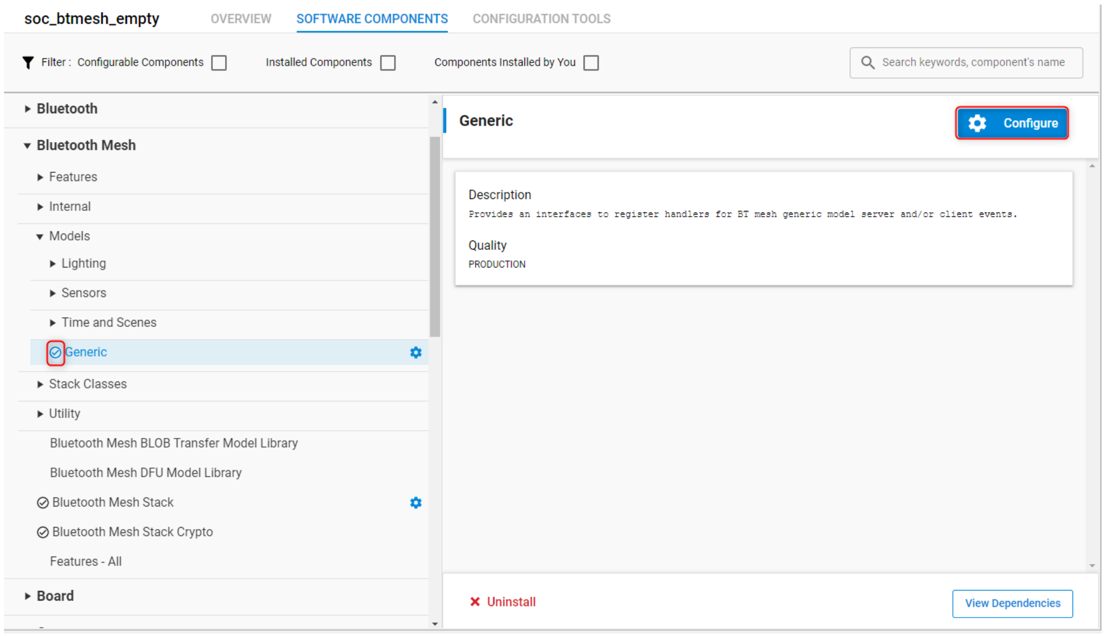
A device configured as a node in a Bluetooth mesh network is initialized by the `sl_btmesh_node_init()` routine. The call to that function is usually present in the Bluetooth mesh user application state machine (switch/case statements) in the app.c file:

```
sl_status_t sl_btmesh_node_init();
```

Each node can be configured in various ways. Some nodes may support a set of models and features that other nodes are not meant to support. This is specific to the user and the network.

In order to make a model or feature functional, make sure the corresponding initialization class routine are called. For example, if a node supports the same generic server and/or client models (On/Off, Level, and so on), follow this procedure.

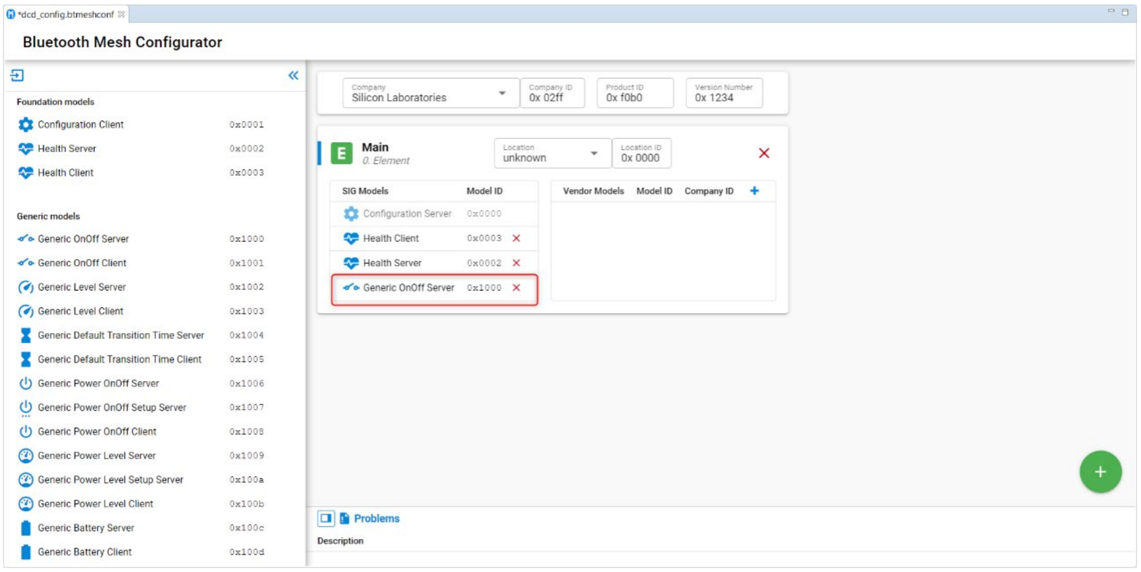
1. Make sure that the corresponding model components are installed in your project. When a component is installed, a blue check symbols appears in front of it. If the component is configurable, a Configure control (gear symbol) is available next to the component name, and as a button in the top right corner.



Description
Provides an interfaces to register handlers for BT mesh generic model server and/or client events.

Quality
PRODUCTION

2. Make sure the models are properly added to your DCD configuration. If the model is not supported as a component, it has to be manually added to the corresponding element using the Bluetooth Mesh Configurator.



Foundation models	
Configuration Client	0x0001
Health Server	0x0002
Health Client	0x0003
Generic models	
Generic OnOff Server	0x1000
Generic OnOff Client	0x1001
Generic Level Server	0x1002
Generic Level Client	0x1003
Generic Default Transition Time Server	0x1004
Generic Default Transition Time Client	0x1005
Generic Power OnOff Server	0x1006
Generic Power OnOff Setup Server	0x1007
Generic Power OnOff Client	0x1008
Generic Power Level Server	0x1009
Generic Power Level Setup Server	0x100a
Generic Power Level Client	0x100b
Generic Battery Server	0x100c
Generic Battery Client	0x100d

Company: Silicon Laboratories
Company ID: 0x 02ff
Product ID: 0x f0b0
Version Number: 0x 1234

Main
0. Element
Location: unknown
Location ID: 0x 0000

SIG Models	Model ID
Configuration Server	0x0000
Health Client	0x0003
Health Server	0x0002
Generic OnOff Server	0x1000

Vendor Models	Model ID	Company ID

3. Make sure the corresponding Bluetooth mesh classes are properly initialized.

```
/** @brief Table of used BGAPI classes */
static const struct sli_bgapi_class * const btmesh_class_table[] =
{
  SL_BTMESH_BGAPI_CLASS(health_server),
  SL_BTMESH_BGAPI_CLASS(proxy),
  SL_BTMESH_BGAPI_CLASS(proxy_server),
  SL_BTMESH_BGAPI_CLASS(node),
  SL_BTMESH_BGAPI_CLASS(generic_server),
  NULL
};
```

Note: This section only describes the common pitfalls that a user might encounter when setting up a node using generic client/server as an illustration. It is not an exhaustive list of steps that are necessary to have a generic On/Off or Battery client/server up and running, which would require a much longer description.

In the case of a generic On/Off server model for example, the following steps would need to be implemented on the server:

- Register a generic server handler.

```
static void init_models(void)
{
  mesh_lib_generic_server_register_handler(MESH_GENERIC_ON_OFF_SERVER_MODEL_ID,
                                          BTMESH_GENERIC_ONOFF_SERVER_MAIN,
                                          onoff_request,
                                          onoff_change,
                                          onoff_recall);
}
```

- Populate the corresponding request/change/recall functions.

```
static void onoff_request(uint16_t model_id,
                         uint16_t element_index,
                                 uint16_t element_index,
                         uint16_t client_addr,
                         uint16_t server_addr,
                         uint16_t appkey_index,
                         const struct mesh_generic_request *request,
                         uint32_t transition_ms,
                         uint16_t delay_ms,
                         uint8_t request_flags)

static void onoff_change(uint16_t model_id,
                         uint16_t element_index,
                         const struct mesh_generic_state *current,
                         const struct mesh_generic_state *target,
                         uint32_t remaining_ms)

static void onoff_recall(uint16_t model_id,
                         uint16_t element_index,
                         const struct mesh_generic_state *current,
                         const struct mesh_generic_state *target,
                         uint32_t transition_ms)
```

Note that this is only an example based on the generic On/Off model. It is commonly more difficult to start from scratch with generic models as it would require a very good understanding of both the Bluetooth mesh technology and the stack.

In order to set up a working node configured as a light client/server model, Silicon Labs recommends using the sample application in Simplicity Studio. For more information, refer to *AN1299: Understanding the Silicon Labs Bluetooth Mesh SDK v2.x Lighting Demonstration*.

3.1.3 Provisioner

To configure a device as a provisioner, first install the **Provisioner** component.

A provisioner must support the configuration client model. As a result the corresponding models should be added in the Bluetooth Mesh Configurator. Check that the configuration client model is part of the Bluetooth mesh init class table, generated by the SDK, in the sl_btmesh.c file:

```
static const struct sli_bgapi_class * const btmesh_class_table[] =
{
…
SL_BTMESH_BGAPI_CLASS(config_client),
NULL
};
```

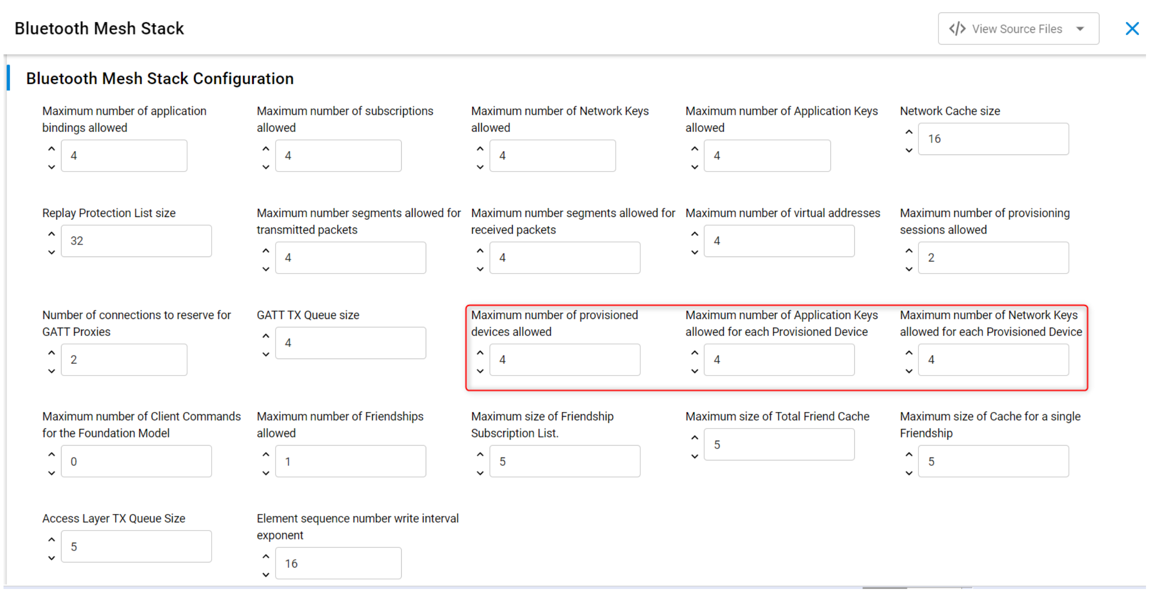
This should also now be visible in the Bluetooth Mesh Configurator user interface.

Additionally, configure the **Bluetooth Mesh Stack** component so that the following parameters are set to values corresponding to your network:

- Maximum number of provisioned devices allowed.
- Maximum number of App keys allowed for each Provisioned device.
- Maximum number of Net keys allowed for each Provisioned device.

These three are required for successful provisioning in any network size, and may be enough for a very small network consisting of one or two nodes. In more complicated networks, provisioner configuration depends on many other parameters. For more information on Bluetooth mesh stack parameters, refer to *UG472: Bluetooth® Mesh Stack and Bluetooth® Mesh Configurator User's Guide for SDK v2.x*.

The following table shows an example. Note that the parameters are set to 4 as an example, not as a recommendation.



Bluetooth Mesh Stack

Bluetooth Mesh Stack Configuration

Maximum number of application bindings allowed	4
Maximum number of subscriptions allowed	4
Maximum number of Network Keys allowed	4
Maximum number of Application Keys allowed	4
Network Cache size	16
Replay Protection List size	32
Maximum number segments allowed for transmitted packets	4
Maximum number segments allowed for received packets	4
Maximum number of virtual addresses	4
Maximum number of provisioning sessions allowed	2
Number of connections to reserve for GATT Proxies	2
GATT TX Queue size	4
Maximum number of provisioned devices allowed	4
Maximum number of Application Keys allowed for each Provisioned Device	4
Maximum number of Network Keys allowed for each Provisioned Device	4
Maximum number of Client Commands for the Foundation Model	0
Maximum number of Friendships allowed	1
Maximum size of Friendship Subscription List.	5
Maximum size of Total Friend Cache	5
Maximum size of Cache for a single Friendship	5
Access Layer TX Queue Size	5
Element sequence number write interval exponent	16

Figure 3.2. Bluetooth Mesh Stack Component - Provisioner Settings

The Provisioner is initialized using the following routine. The call to that function is usually present in the Bluetooth mesh user application state machine (switch/case statement) in the app.c file:

```
sl_status_t sl_btmesh_prov_init();
```

No Bluetooth mesh API routine can be called before this one. Additionally, the node-initializing routine `sl_btmesh_node_init()` must not be called on a provisioner.

3.1.4 GATT Database

The GATT (Generic Attribute Profile) database is a standardized way of describing a Bluetooth device's profiles, services, and characteristics. With the Silicon Labs Bluetooth stack, the GATT definitions are either directly edited in the Bluetooth GATT Configurator in Simplicity Studio or are written in XML. For more information on how to create GATT databases and the syntax, refer to *UG118: Blue Gecko Bluetooth® Smart Profile Toolkit Developer's Guide*.

gatt_db.c and gatt_db.h

The gatt_db.c file defines the GATT database structure and content. It is autogenerated by the Bluetooth GATT configurator. gatt_db.h includes this database and the handles of local characteristics and services. Type definitions of GATT are automatically included from gatt_db_def.h to gatt_db.h. In the case of Bluetooth mesh, only the Bluetooth mesh services are relevant:

- Mesh Proxy service (UUID 0x1828): The Bluetooth mesh Proxy Service is used to enable a server to send and receive Proxy PDUs with a proxy client .
  - Mesh Proxy Data In (UUID 0x2ADD)
  - Mesh Proxy Data Out (UUID 0x2ADE)
- Mesh Provisioning service (UUID 0x1827): The Bluetooth mesh Provisioning Service allows a Provisioning Client to provision a Provisioning server to allow it to participate in the Bluetooth mesh network.
  - Mesh Provisioning Data In (UUID 0x2ADB)
  - Mesh Provisioning Data Out (UUID 0x2ADC)

Those services and characteristics can be thought of as duplex communication channels for provisioning and proxy PDUs. The "data in" and "out" characteristics are then the Tx and Rx channel, respectively.

A device may support the Bluetooth mesh Provisioning Service or the Mesh Proxy Service or both. If both are supported, only one of these services should be exposed in the GATT database at a time.

3.1.5 Bluetooth Mesh Device Composition Data

The device composition data, or DCD, is a set of data indicating which features are supported, how many elements are present on a node with their description, and a set of identifiers defining the model layout across the elements of the node.

The Composition Data state contains information about a node, the elements it includes, and the supported models. The Composition Data is composed of a number of pages of information. Composition Data Page 0 is mandatory. All other pages are optional. All Composition Data Pages not defined in this specification are reserved for future use. The size of the state must not exceed the maximum useful access payload size.

The following code snippet illustrate how this is defined for the Bluetooth mesh stack:

```
const uint8_t __mesh_dcd[] = {
  U16TOA(0x02ff), /* Company ID */
  U16TOA(0xf0b0), /* Product ID */
  U16TOA(0x1234), /* Version Number */
  U16TOA(SL_BTMESH_CONFIG_RPL_SIZE), /* Capacity of Replay Protection List */
  U16TOA(SL_BTMESH_FEATURE_BITMASK), /* Features Bitmask */
  /* Main */
    U16TOA(0x0000), /* Location */
    0x04, /* Number of SIG Models = 4 */
    0x00, /* Number of Vendor Models = 0 */
    /* SIG Models */
      U16TOA(0x0000), /* Configuration Server */
      U16TOA(0x0002), /* Health Server */
      U16TOA(0x0001), /* Configuration Client */
      U16TOA(0x0003), /* Health Client */
};

const uint8_t *__mesh_dcd_ptr = __mesh_dcd;
```

The structure called __mesh_dcd is passed via a pointer to the C Bluetooth mesh library.

After provisioning, the Provisioner typically retrieves the DCD of the newly provisioned node in order to determine the node’s features and functionalities so that it can be configured to operate in the network.

3.1.6 RTOS Support

Note that multiprotocol applications are not supported by the Bluetooth mesh protocol. RTOS is available only for the Bluetooth LE stack.

4 Bluetooth Mesh Stack Event Handling

The Bluetooth mesh stack for the Wireless Geckos is an event-driven architecture, where events are handled in the main while loop. The Bluetooth mesh stack runs on top of the Silicon Labs Bluetooth stack.

4.1 Bluetooth LE versus Bluetooth Mesh Event

The Bluetooth mesh protocol is built on top of Bluetooth Low Energy. This mean that a device running a Bluetooth LE stack will be able to receive a Bluetooth mesh PDU but will not be able to interpret the data that it contains. Nevertheless, a node or device can run both. In effect, it is possible to have Bluetooth LE and Bluetooth mesh events treated separately in an application.

In this particular case, Bluetooth LE and Bluetooth mesh event application state machines need to be separate. In other words, the Bluetooth LE and Bluetooth mesh events cannot be treated in a unique switch-case statement.

At the application level, the Silicon Labs Bluetooth Mesh API provides a way to differentiate Bluetooth mesh events from Bluetooth LE events. This is done through the Bluetooth mesh listener.

The Bluetooth mesh events in the stack are handled similarly to the regular Bluetooth LE events. In a freshly created project, the Bluetooth mesh switch case statement is performed by the routine `sl_btmesh_step()` and `sl_btmesh_process_event()`.

For more information on how events are processed in both the Bluetooth LE and mesh stacks, refer to the ‘Bluetooth Stack Event Handling’ section in *UG434: Silicon Labs Bluetooth C Application Developer’s Guide for SDK v3.x*.

5 NVM Layout

The non-volatile memory management system, called non-volatile memory 3 (NVM3), is a data storage driver for storing persistent data primarily, but not only, in internal flash. The term "non-volatile" and "NVM3" are synonymous in this section.

AN1135: Using Third Generation Non Volatile Memory Data Storage describes in detail how NVM3 operates. The NVM3 subsystem allocates a certain range of address to both the user and the Bluetooth LE and Bluetooth mesh stacks, among other things. The following table shows what key ranges are dedicated to the stack and the user:

Domain	NVM3 Key Range
User	0x00000 - 0x0FFFF
Bluetooth stack	0x40000 - 0x4FFFF

The Bluetooth stack key range is shared between regular Bluetooth Low Energy and Bluetooth mesh. Within this key range, the distribution is laid out as follow:

Domain	NVM3 Key Range
Bluetooth internal stack data (bonding, etc.)	0x40000 - 0x40FFF
Bluetooth mesh stack data	0x41000 - 0x44000
Reserved for future use	0x48000 - 0x4FFFF

6 Bluetooth Mesh Features

Optional networking or energy features may also be implemented in a Bluetooth mesh application. The Bluetooth mesh profile specification refers to those simply as features. There are four Bluetooth mesh features: Proxy, Relay, Friends, and Low Power nodes. This section describes all four.

Note: These features are specific to nodes, that is, devices in a Bluetooth mesh network. The provisioner of a network is not subject to features support.

6.1 Proxy

The Proxy feature allows a node to receive and transmit Bluetooth mesh messages between GATT and advertising bearers. The proxy feature is used to forward Network packets received by a node between GATT bearer and advertising bearers. This feature is optional and can be enabled/disabled at runtime. When this feature is enabled, the corresponding GATT Proxy service must be exposed.

The proxy feature defines two roles, the proxy client and the server. The proxy server is a node that supports both the GATT bearer and the advertising bearer. In practice, the proxy client is a node that supports only the GATT bearer.

For a proxy feature to run, the corresponding proxy routine class must be part of the stack initialization table. The following code snippet gives a proxy server example.

```
static const struct sli_bgapi_class * const btmesh_class_table[] =
{
  SL_BTMESH_BGAPI_CLASS(health_server),
  SL_BTMESH_BGAPI_CLASS(proxy),
  SL_BTMESH_BGAPI_CLASS(proxy_server),
  SL_BTMESH_BGAPI_CLASS(node),
  NULL
};
```

6.2 Relay

The Relay feature allows a node to receive and retransmit Bluetooth mesh messages over the advertising bearer to enable larger networks.

The provisioner can enable the relay feature on a particular node (if supported) via the following routine:

```
sl_status_t sl_btmesh_config_client_set_relay(uint16_t enc_netkey_index,
                                               uint16_t node_address,
                                               uint8_t  value,
                                               uint8_t  retransmit_count,
                                               uint16_t retransmit_interval_ms,
                                               uint32_t * handle)
```

A getter function is also available. For more details, refer to the Bluetooth Mesh Configuration Client section of the API html documentation.

Note: In large networks, it is in general a good practice to limit the number of nodes supporting the relay feature. Otherwise, the data traffic can increase very rapidly to undesired levels.

6.3 Friend

The Friend feature allows a node to help a node supporting the Low Power feature to operate by storing messages destined for that node. Friendship is used by Low Power Nodes to limit the amount of time that they need to listen.

The application code for nodes supporting that feature need to enable it using:

- `sl_status_t sl_btmesh_friend_init(void)` for enabling the feature.
- `sl_status_t sl_btmesh_friend_deinit(void)` for disabling the feature.

For more information, refer to the HTML API Reference delivered with the SDK.

6.4 Low Power Node

The Low Power Node (LPN) feature allows a node to operate within a Bluetooth mesh network at significantly reduced receiver duty cycles, only in conjunction with a node supporting the Friend feature.

Similarly to the Friend feature, the application code for nodes supporting the Low Power Node feature needs to enable it:

- `sl_status_t sl_btmesh_lpn_init(void)`
- `sl_status_t sl_btmesh_lpn_deinit(void)`

When the feature has been enabled on a node, and if a node offering friendship is within radio range, the friendship can be established and terminated using the following routines with the associated network key index:

- `sl_status_t sl_btmesh_lpn_establish_friendship(uint16_t netkey_index)`
- `sl_status_t sl_btmesh_lpn_terminate_friendship(uint16_t netkey_index)`

The Silicon Labs Bluetooth Mesh API allows the user to configure the time interval at which the LPN will poll the friend as well as other time variables:

- `sl_status_t sl_status_t sl_btmesh_lpn_config(uint8_t setting_id, uint32_t value)`

The following arrays describes the setting id enum used by the stack to aggregates the LPN configuration values `(enum sl_btmesh_lpn_settings_t):`

`sl_btmesh_lpn_queue_length (0x00)`	Minimum queue length the friend must support in bytes. The value is rounded up to the nearest power of 2. Default is 2. Range is 2..128.
`sl_btmesh_lpn_poll_timeout (0x01)`	Poll timeout in milliseconds, which is the longest time that LPN sleeps in between querying its friend for queued messages. Default is 50 ms. The value is rounded to the nearest multiple of 100 ms. The range is 1 s to 95 h 59 min 59 s 900 ms.
`sl_btmesh_lpn_receive_delay (0x02)`	Receive delay in milliseconds. Receive delay is the time between the LPN sending a request and listening for a response. Receive delay allows the friend node time to prepare the message and the LPN to sleep. Range: 10 ms to 255 ms. The default receive delay is 10 ms. The default value is 10.
`sl_btmesh_lpn_request_retries (0x03)`	Request retry is the number of retry attempts to repeat, for example, how many times to repeat the friend poll message if the friend update was not received by the LPN. Range is from 0 to 10. The default value is 6 (initial attempt plus 5 retries).
`sl_btmesh_lpn_retry_interval (0x04)`	Time interval between retry attempts in milliseconds. Range is 0 to 100 ms. The default value is 100 ms.

Additionally, a friend poll request can be sent from the LPN at any time using the flowing routine with the appropriate network key index:

`sl_status_t sl_btmesh_lpn_poll(uint16_t netkey_index)`

However, it is not required for correct operation, because the procedure will be performed automatically before the poll timeout expires.

For more information on the friend and LPN API, refer to the HTML API documentation.

sl_btmesh_dcd.c

From a practical standpoint, the device composition data of each node contains a 2-byte field indicating the supported features. The following array illustrates the features field:

Bit	Feature	Notes
0	Relay	Relay feature supported if set to 1. 0 otherwise.
1	Proxy	Proxy feature supported if set to 1. 0 otherwise.

Bit	Feature	Notes
2	Friend	Friend feature supported if set to 1. 0 otherwise.
3	Low Power Node (LPN)	LPN feature supported if set to 1. 0 otherwise.
4 - 15	Reserved for future use.	Reserved for future use.

As mentioned in section 3.1.5 Bluetooth Mesh Device Composition Data, each node, after provisioning, sends its composition data page 0 to the provisioner. In the code example presented in that section, the macro SL_BTMESH_FEATURE_BITMASK is used with the default value of 3:

```
#define SL_BTMESH_FEATURE_BITMASK 3
```

In this example, that macro enables the relay and proxy features (3) in the Bluetooth mesh stack. This is the default setting.

7 Bluetooth Mesh Stacks and Wireless Gecko Configuration and Resources

To run the Bluetooth stack and an application on a Wireless Gecko, the MCU and its peripherals have to be properly configured. Once the hardware is initialized, the stack also has to be initialized using the `sl_btmesh_init()` function as described in section 3.1.1 Stack. This process is automated by the SDK.

sl_system_init()

The `sl_system_init()` function is used to initialize the system. It will call platform, driver, service, stack, and internal app init functions, located in the autogen folder.

app_init()

This function is used to initialize application-specific features.

7.1 Wireless Gecko MCU and Peripherals Configuration

When the configuration is relevant to Bluetooth mesh, the information is the same as provided in the 'Wireless Gecko MCU and Peripherals Configuration' section of *UG434: Silicon Labs Bluetooth C Application Developer's Guide for SDK v3.x*. The following table follows the order in that document and provides cross-references where appropriate.

Configuration	Bluetooth mesh	Reference
Adaptive Frequency Hopping	Not supported when using the advertising bearer, as all data traffic uses on the primary advertising channels. If the GATT bearer is used, Bluetooth mesh data are sent and received via the Proxy protocol, which uses a Bluetooth Low Energy connection with dedicated Bluetooth mesh services.	Mesh profile specification
Bluetooth Clocks	Supported.	UG434, 'Bluetooth Clocks'
DC-DC Configuration	Supported.	UG434, 'DC-DC Configuration'
LNA	Supported.	UG434, 'LNA'
Periodic Advertising	Not supported. Legacy advertising (31 bytes long) only, as per the profile specification.	
PTI	Supported.	UG434, 'PTI'
Transmit Power	Supported.	UG434, 'Transmit Power'
Filter Accept List	Supported.	UG434, 'Accept List Filtering'
Wi-Fi Coexistence	Supported.	UG434, 'Wi-Fi Coexistence'
OTA Configuration	Supported using Bluetooth LE services.	UG434, 'OTA Configuration'
Even Connection Distribution Algorithm	Not supported.	
Interrupts	Supported	UG434, 'Interrupts'

7.2 Wireless Gecko Resources

The Bluetooth mesh stack uses some of the Wireless Gecko's resources, which are not available to the application. The following table lists the resources and describes their use by the stack. The first four resources are always used by the Bluetooth stack.

Category	Resource	Used in software	Notes
PRS	PRS7	PROTIMER RTC synchronization	PRS7 always used by the Bluetooth stack.
Timers	RTCC	EM2	The sleep timer uses RTCC in the default configuration.

Category	Resource	Used in software	Notes
“	PROTIMER	Bluetooth (LE and mesh)	The application does not have access to PROTIMER.
Radio	RADIO	Bluetooth	Always used and all radio registers are reserved for the Bluetooth LE and mesh stack.
GPIO	NCP	Host communication	Up to 4 I/O pin. Optional.
“	PTI	Packet trace	2 to N I/O pins. Optional.
“	TX ACTIVE	TX activity indication	1 I/O pin. Optional.
“	RX ACTIVE	RX activity indication	1 I/O pin. Optional.
“	COEX PTA	Wi-Fi Coexistence	Up to 4 I/O pins. Optional.
CRC	GPCRC	NVM3	Can be used in application, but application should always reconfigure GPCRC before use, and GPCRC clock must not be disabled in CMU.
Flash	MSC	NVM3	Can be used by the application.
Crypto	CRYPTO	Bluetooth Link encryption	The CRYPTO peripheral can only be accessed through the mbedTLS crypto library, not through any other means. The library should be able to do the scheduling between the stack and application access.
“	RADIO AES	Bluetooth Link encryption	The application does not have access to RADIOAES

7.2.1 Internal Flash and SRAM

For more information, refer to the Wireless Gecko Resources section in *UG434: Silicon Labs Bluetooth C Application Developer’s Guide for SDK v3.x*.

7.2.2 Monitoring Radio RX and TX State Using PRS (Peripheral Reflex System)

It is sometimes useful, for debugging purposes, to monitor the state of the radio transmitter/receiver. This can be done by outputting on pins the RX_ACTIVE and TX_ACTIVE signals. An example is provided here on how to do that on series 2 devices (EFR32xG21-based Wireless Gecko starter kit).

First, make sure the **PRS** component is installed in the project. Then the following code example indicates how PRS can be used to output the RX_ACTIVE and TX_ACTIVE signals.

Note: As indicated in the table in section 7.2 Wireless Gecko Resources, PRS channel 7 is used by the Bluetooth LE stack and cannot be used in this example.

```
#include "em_prs.h"
#include "em_cmu.h"

/* Enable TX_ACT signal through GPIO PD03 */
#define _PRS_CH_CTRL_SOURCESEL_RAC2 0x00000031UL
#define PRS_CH_CTRL_SOURCESEL_RAC2 (_PRS_CH_CTRL_SOURCESEL_RAC2 << 8)
#define _PRS_CH_CTRL_SIGSEL_RACRX 0x00000003UL
#define PRS_CH_CTRL_SIGSEL_RACRX (_PRS_CH_CTRL_SIGSEL_RACRX << 0)
#define _PRS_CH_CTRL_SIGSEL_RACTX 0x00000004UL
#define PRS_CH_CTRL_SIGSEL_RACTX (_PRS_CH_CTRL_SIGSEL_RACTX << 0)

/* RACPAEN Enable (TX_ACT) signal through GPIO PD03 */
#define TX_ACTIVE_PRS_SOURCE PRS_CH_CTRL_SOURCESEL_RAC2
#define TX_ACTIVE_PRS_SIGNAL PRS_CH_CTRL_SIGSEL_RACTX
#define TX_ACTIVE_PRS_CHANNEL 10
```

```
#define TX_ACTIVE_PRS_PORT gpioPortD
#define TX_ACTIVE_PRS_PIN 3

/* Enable RX_ACT signal through GPIO PD02 */
#define RX_ACTIVE_PRS_SOURCE PRS_CH_CTRL_SOURCESEL_RAC2
#define RX_ACTIVE_PRS_SIGNAL PRS_CH_CTRL_SIGSEL_RACRX
#define RX_ACTIVE_PRS_CHANNEL 11
#define RX_ACTIVE_PRS_PORT gpioPortD
#define RX_ACTIVE_PRS_PIN 2
```

This snippet of codes defines which PRS signals coming from the radio source (RAC, 0x31 on xG21) should be used. Those signals will then be routed to the desired pins, in this case PD3 for the TX_ACTIVE signal and PD2 for the RX_ACTIVE signal.

Then the following functions set up the pins and configure the PRS module:

```
static void initGpio(void)
{
  // Set RX/TX active pins
  GPIO_PinModeSet(TX_ACTIVE_PRS_PORT, TX_ACTIVE_PRS_PIN, gpioModePushPull, 0);
  GPIO_PinModeSet(RX_ACTIVE_PRS_PORT, RX_ACTIVE_PRS_PIN, gpioModePushPull, 0);

  /* Set up GPIO clock */
  CMU_ClockEnable(cmuClock_GPIO,true);
}

static void initPrs(void)
{
  /* Enable PRS clock */
  CMU_ClockEnable(cmuClock_PRS, true);

  /* Use RAC, PAEN as PRS source */
  PRS_SourceAsyncSignalSet( TX_ACTIVE_PRS_CHANNEL, PRS_RAC_PAEN, PRS_RAC_PAEN);


  /* Use RAC, RX_ACT as PRS source */
  PRS_SourceAsyncSignalSet( RX_ACTIVE_PRS_CHANNEL, PRS_RAC_RX, PRS_RAC_RX);

  /* Route output to PC01. No extra PRS logic needed here. */
  PRS_PinOutput(TX_ACTIVE_PRS_CHANNEL,prsTypeAsync, TX_ACTIVE_PRS_PORT , TX_ACTIVE_PRS_PIN);
  PRS_PinOutput(RX_ACTIVE_PRS_CHANNEL,prsTypeAsync, RX_ACTIVE_PRS_PORT , RX_ACTIVE_PRS_PIN);
}
```

The `initGpio()` routine sets the previously-defined pins as output and enables the GPIO clock. The `initPrs()` routine enables the PRS module clock, sets the asynchronous channels, and routes the signals to the pins.

The two functions need to be called in the user application code as such:

```
SL_WEAK void app_init(void)
{
  /* Set up GPIOs */
  initGpio();

  /* Set up PRS */
  initPrs();
}
```

The radio state can then be monitored using the defined pins on a logic analyzer. In this example, the radio is running a simple Bluetooth LE advertisement example. On each of the three primary advertising channels, data is first transmitted (long logic high) then the radio switches to the receive state (short logic high), which is repeated on each channel.

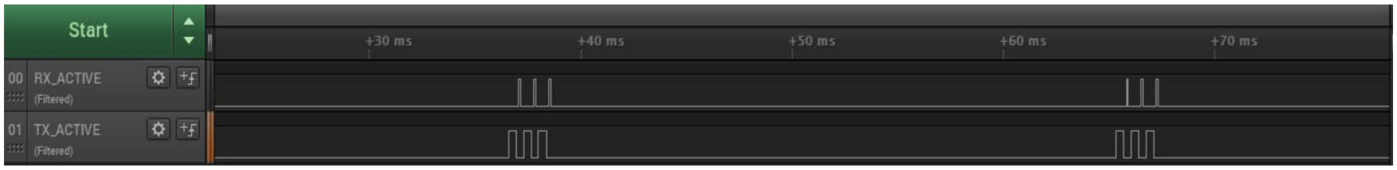


Figure 7.1. Radio State Monitored (Bluetooth LE Advertisement)

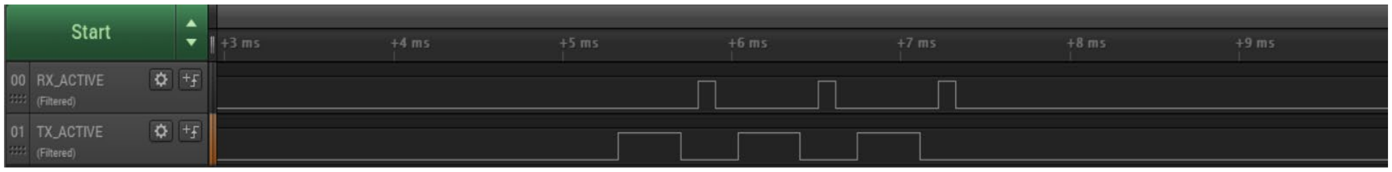
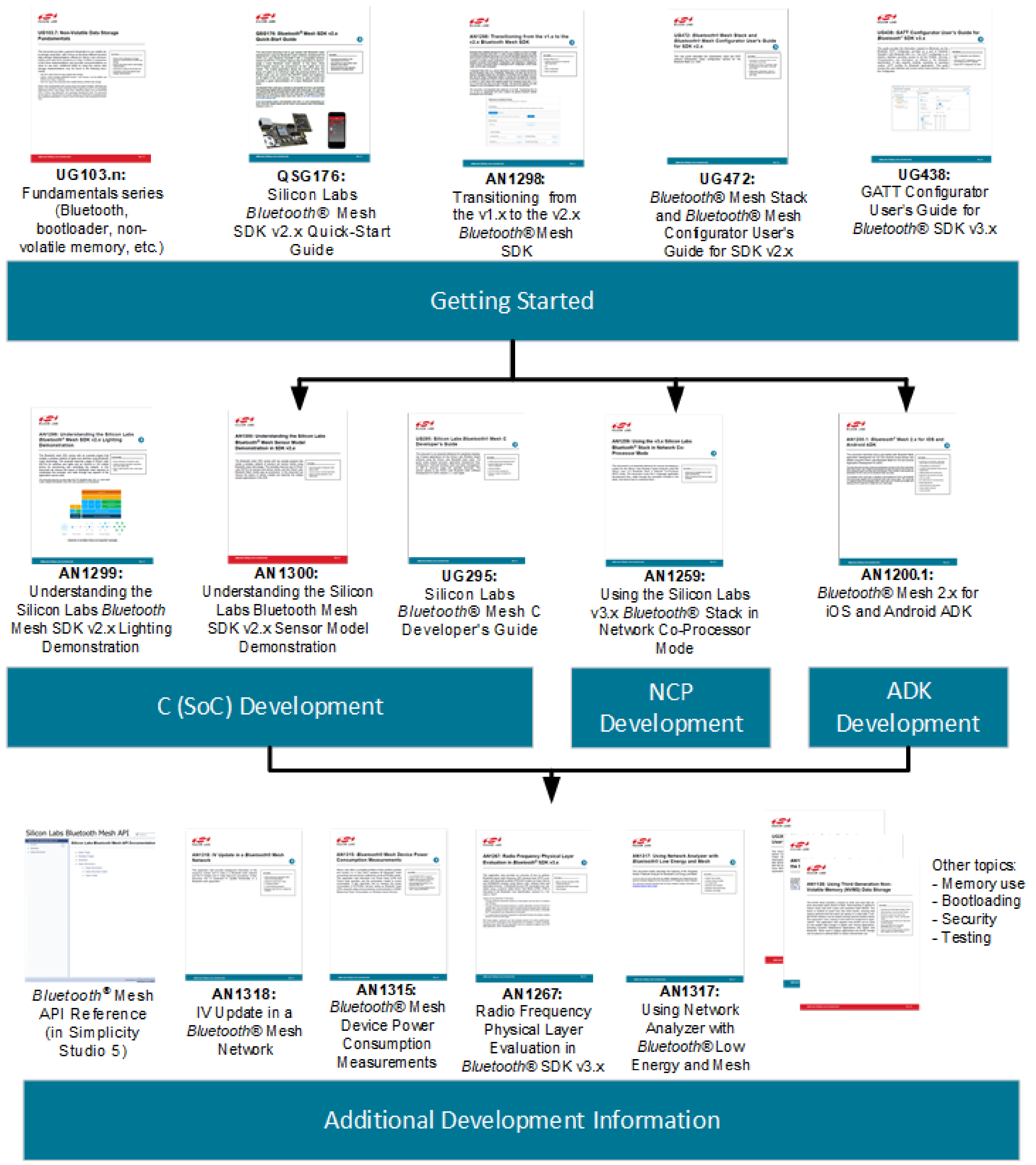
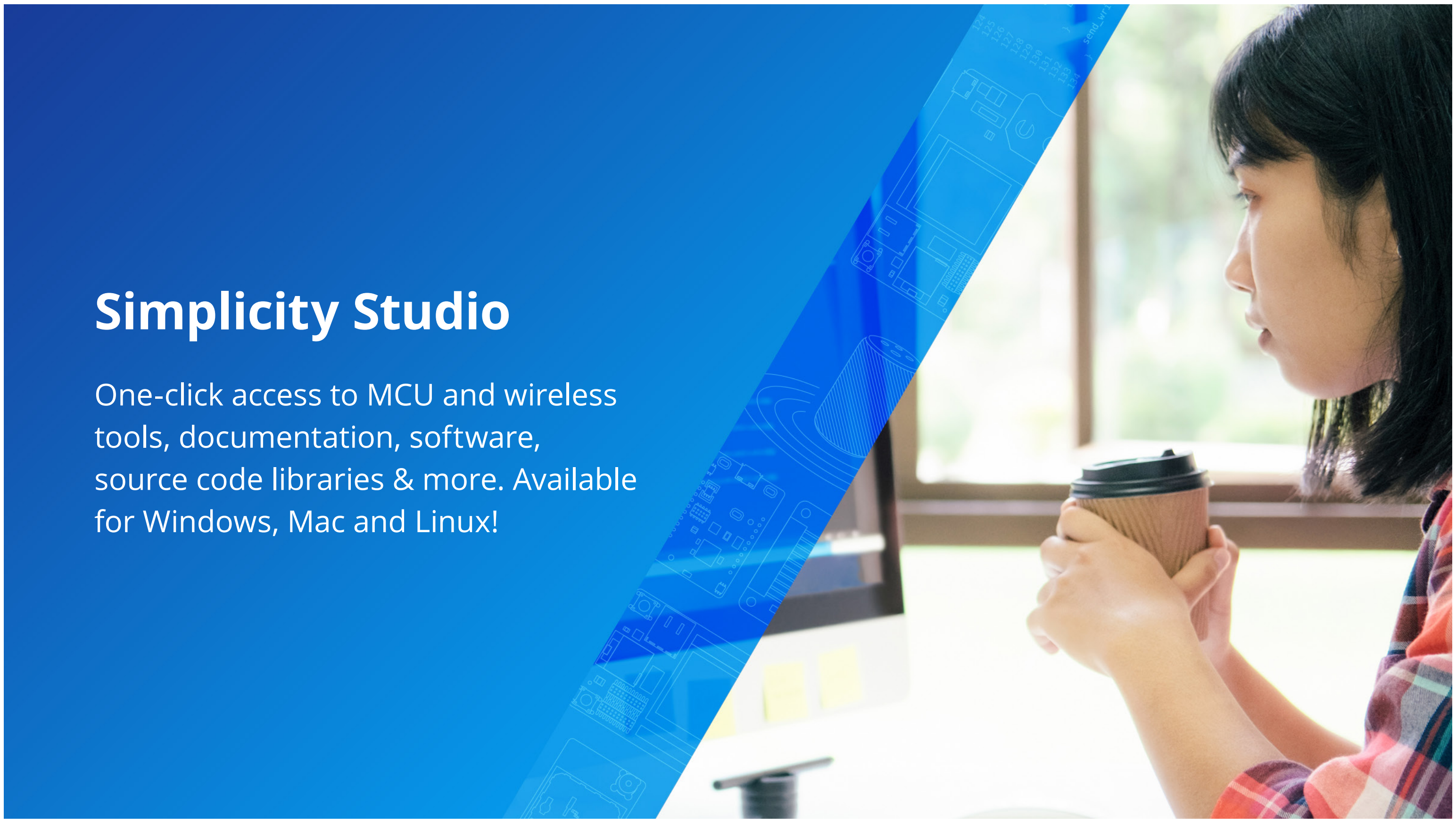


Figure 7.2 Radio State Monitored (Zoomed)

8 Documentation


UG103.n: Fundamentals series (Bluetooth, bootloader, non-volatile memory, etc.)
QSG176: Silicon Labs Bluetooth® Mesh SDK v2.x Quick-Start Guide
AN1298: Transitioning from the v1.x to the v2.x Bluetooth® Mesh SDK
UG472: Bluetooth® Mesh Stack and Bluetooth® Mesh Configurator User's Guide for SDK v2.x
UG438: GATT Configurator User's Guide for Bluetooth® SDK v3.x
Getting Started
AN1299: Understanding the Silicon Labs Bluetooth Mesh SDK v2.x Lighting Demonstration
AN1300: Understanding the Silicon Labs Bluetooth Mesh SDK v2.x Sensor Model Demonstration
UG295: Silicon Labs Bluetooth® Mesh C Developer's Guide
AN1259: Using the Silicon Labs v3.x Bluetooth® Stack in Network Co-Processor Mode
AN1200.1: Bluetooth® Mesh 2.x for iOS and Android ADK
C (SoC) Development
NCP Development
ADK Development
Bluetooth® Mesh API Reference (in Simplicity Studio 5)
AN1318: IV Update in a Bluetooth® Mesh Network
AN1315: Bluetooth® Mesh Device Power Consumption Measurements
AN1267: Radio Frequency Physical Layer Evaluation in Bluetooth® SDK v3.x
AN1317: Using Network Analyzer with Bluetooth® Low Energy and Mesh
Other topics:
- Memory use
- Bootloading
- Security
- Testing
Additional Development Information


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

IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support & Community
www.silabs.com/community

SILICON LABS

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

www.silabs.com