UG434: Silicon Labs *Bluetooth* ® C Application Developer's Guide for SDK v5.x and Higher

This document is an essential reference for everybody developing C-based applications for the Silicon Labs Wireless Gecko products using the Silicon Labs Bluetooth stack. The guide covers the Bluetooth stack architecture, application development flow, usage, and limitations of the MCU core and peripherals, stack configuration options, and stack resource usage. This version applies to the Silicon Labs Bluetooth Software Development Kit (SDK) version 5.x and higher.

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

KEY POINTS

- Project structure and development flow
- Bluetooth stack and Wireless Gecko configuration
- Interrupt handling
- Event and sleep management
- Resource usage and available resources

Copyright © 2023 by Silicon Laboratories

Table of Contents

1. Introduction

This document is a C developer's guide for the Silicon Labs Bluetooth stack. It covers various angles of development, and is an important reference to everyone 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. Configuring the Bluetooth Stack and a Wireless Gecko Device explains the project include libraries and the actual Wireless Gecko configuration in the application code.
- Section 5. Bluetooth 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 6. Interrupts and section 7. Wireless Gecko Resources touch on the topics of peripherals and the chipset resources, covering what is reserved for the stack usage, how interrupts should be handled, and the stack's memory footprint and available memory for the application.

1.1 About this Version

The current version of Silicon Labs' Bluetooth SDK is 5.x.

Currently supported compilers and IDE versions are:

- **IDE:** Simplicity Studio 5.2 or newer
- **Compiler:** GCC 10.3-2021.10, and IAR 9.20.4

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 the reader is familiar with the quick start guides and with the SDK's examples. Also, the reader should have a basic understanding of Bluetooth technology. For more information, see *UG103.14: Bluetooth Technology Fundamentals*.

For instructions on getting started using example applications in Silicon Labs Simplicity Studio development environment, see *QSG169: Bluetooth® SDK v3.x Quick Start Guide*.

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 connectivity for the end-device.

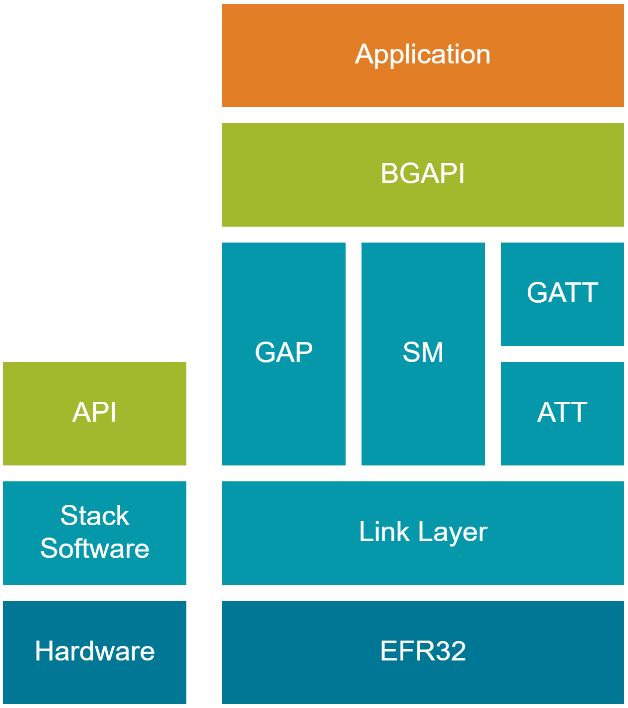


Figure 2.1. Bluetooth Stack Architecture Block Diagram

The Bluetooth stack contains 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, AppLoader starts the application. If the application image is not valid, AppLoader starts the OTA process to try to receive a valid application image. This requires using the Gecko Bootloader. Beginning with Bluetooth SDK version 4.0.x on EFR32[M|B]G2x devices, AppLoader was merged with Gecko Bootloader as a communication plugin.

2.1 Application Build Flow

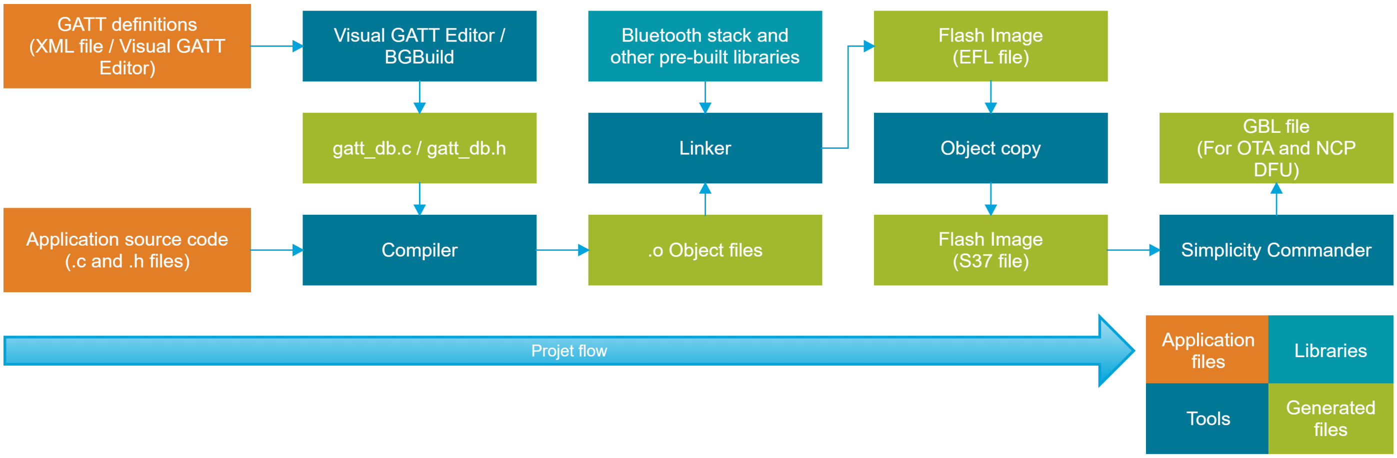


Figure 2.2. Bluetooth Project Build Flow

Building a project starts by defining the Bluetooth services and characteristics (GATT definitions) and by writing the application source code from Silicon Labs-provided examples or an empty project template, as described in *QSG169: Bluetooth® SDK v3.x Quick Start Guide*.

SDK v2.1.0 and later offer two ways to define Bluetooth services and characteristics. The first option is the Visual GATT Editor GUI in Simplicity Studio. This is a graphical tool for designing the GATT and for generating *gatt_db.c* and *gatt_db.h*. Additionally, it can import *.xml* GATT definition files. The Visual GATT Editor is the default tool for GATT definition and generation in Simplicity Studio projects.

The second option is to create an *.xml* according to the *UG118: Blue Gecko Bluetooth® Profile Toolkit Developer's Guide* and then use the BGBuild tool as a pre-compilation step to convert the GATT definition file into .c and .h. This method is used in IAR Embedded Workbench projects.

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.

3. Project Structure

This section explains the application project structure and the mandatory and optional resources that must be included in the project.

3.1 Bluetooth Files

Library Files

The Bluetooth stack libraries are:

- **binapploader.o:** Binary image of the Bluetooth AppLoader for EFR32[M|B]G1x devices, provides the optional OTA (Over-the-Air) functionality.
- **binapploader_nvm3.o:** Binary image of the Bluetooth AppLoader for EFR32[M|B]G1x devices with NVM3 support.
- **libbluetooth.a:** Bluetooth stack library.
- **libpsstore.a:** PS Store functionality for the Bluetooth stack. This is not available on EFR32[M|B]G2x devices. NVM3 must be used instead.

RAIL

The Bluetooth stack uses RAIL to access the radio and RAIL libraries needs to be linked with Bluetooth stack. RAIL has separate libraries for each device family and for single- and multi-protocol environments. RAIL libraries are provided in the Gecko SDK Suite. For more information refer to *UG103.13: RAIL Fundamentals* and other RAIL documentation.

Note: To ensure regulatory compliance of the radio module, the Bluetooth stack for the radio module needs to be linked together with the RAIL library and the configuration library for the radio module. These are librail_module_<soc family><compiler>_release.a and librail_config<modulename>.a.

EMLIB and EMDRV

The Bluetooth stack uses 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 SDK Suite. For more details on EMLIB and EMDRV, see platform EMDRV documentation and EMLIB documentation on https://docs.silabs.com/.

mbed TLS

The Bluetooth stack uses the Mbedtls library for cryptographic operations. The Mbedtls library is provided in source code and must be included in the project. Mbedtls is part of the Gecko SDK Suite. For more details, refer to the Mbedtls documentation.

Sleep Timer

Sleep Timer is a platform component providing software timers, timekeeping, and date functionality. The Bluetooth stack uses it for internal event scheduling, and it must be included in the project. See platform sleeptimer documentation.

Note that Sleep Timer callbacks are called from the interrupt context. BGAPI functions cannot be called from the callback. Instead, the application should implement the timer task handling in the application main loop. Several simple timer components in the SDK, for example, `simple timer` and `simple timer micriumos`, implement helper functionality that also allows calling BGAPI command from its callback.

Power Manager

Power Manager is a platform component that manages the system's energy modes. Its main purpose is to transition the system to a low energy mode when the processor has nothing to execute. See the reference for your MCU on https://docs.silabs.com/ under Modules> Platform Services>Power Manager.

Header Files

sl_bt_version.h

This file contains the Bluetooth stack version.

API Header Files

These files define the Bluetooth stack API.

These files serve two purposes:

1. They contain the actual Bluetooth stack API and the commands and events for the stack.
2. They provide a configuration and event management API to the Bluetooth stack.

sl_bt_types.h

sl_bt_stack_init.h

sl_bt_api.h

sl_bgapi.h

These files contain the Bluetooth stack API and the commands and events for the stack, and a configuration API for the Bluetooth stack.

sl_bt_ncp_host_api.c, sl_bt_ncp_host.c, sl_bt_ncp_host.h and sl_bt_internal.h

These files are used when developing applications for an external host. They provide the API definitions and adaptation layer between the host application and the BGTAPI serial protocol.

3.2 GATT Database

The GATT (Generic Attribute Profile) database is a standardized way of describing the Bluetooth profiles, services, and characteristics of a Bluetooth device. The Silicon Labs' Bluetooth SDK provides two ways to define the GATT database:

- A static GATT database can be defined in compile time with the appropriate tools provided by the Bluetooth SDK, or can be written in XML and passed to the BGBuild tool as a pre-build task. In this case, the database structure is stored in the ROM, which means faster start-up time and lower memory usage.
- A dynamic GATT database can be defined in runtime with the appropriate BGAPI commands using the `bluetooth_feature_dynamic_gattdb` component. In this case, the database structure is stored in the RAM, which makes it more flexible. This is recommended in the NCP use case to avoid re-building the target code that runs on the Wireless Gecko.

You can also combine these two methods. 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, and is auto-generated by BGBuild or by the Visual GATT Editor. The `gatt_db.h` file includes this database and the handles of local characteristics and services. Type definitions of GATT are automatically included from *bg_gatt_db_def.h* to *gatt_db.h*.

When a dynamic GATT database is used, the handles of a local characteristic or service may change if the attribute table is updated at run time. Therefore, it is not recommended to use the constants in `gatt_db.h` unless absolutely certain that the handle values of characteristics and services are not affected by the dynamic GATT database update.

3.3 Device Firmware Upgrade

Device Firmware Upgrade (DFU) is the process of upgrading the application either over a serial link or over-the-air (OTA). In both cases the application needs to add the following file to enable the support for DFU.

application_properties.c

This file includes the application properties struct that contains information about the application image, such as type, version, and security. The struct is defined in `application_properties.h` in the Gecko Bootloader API. A pre-generated file is included in Simplicity Studio projects, which can be modified to include application-specific properties. The application properties can be accessed using the Gecko Bootloader API. The following members can be updated by changing the defines:

```
// Version number for this application (uint32_t)
#define APP_PROPERTIES_VERSION

// Unique ID (e.g. UUID or GUID) for the product this application is built for (uint8_t[16])
#define APP_PROPERTIES_ID
```

When using the OTA process with Bluetooth AppLoader, a pointer to the application properties struct needs to be set to application vector table vector 13. This is enabled automatically when using the default startup file and the struct name is `sl_app_properties`.

3.4 NCP Applications

When developing applications for an external host, the `SL_BT_API_FULL` define needs to be defined to prevent the linker from dropping the BGAPI command implementation from the application. The define includes a full implementation of all enabled BGAPI classes in the application.

3.5 RTOS Support

The Bluetooth stack is usually run on bare metal configuration, but the Bluetooth stack can also run on Micrium RTOS and FreeRTOS. When the application project includes an RTOS kernel, the Bluetooth RTOS adaptation component with the following files is automatically added to the application:

sl_bt_rtos_adaptation.c

sl_bt_rtos_adaptation.h

sl_bt_rtos_config.h

sl_bt_rtos_adaptation.c and sl_bt_rtos_adaptation.h

`sl_bt_rtos_adaptation.c` and `sl_bt_rtos_adaptation.h` provide the RTOS tasks for the IPC (Inter-Process Communication) with the Bluetooth stack and other RTOS tasks using CMSIS-RTOS2.

sl_bt_rtos_config.h

`sl_bt_rtos_config.h` is used to set the Bluetooth RTOS task priorities and the stack sizes.

When the project is generated, the SLC tools will automatically contribute the relevant Bluetooth stack configuration and initialization to enable the Bluetooth stack to work with the RTOS.

3.6 Multiprotocol Support

When the Bluetooth Stack is used in a multiprotocol environment, the application must use the RAIL library with multiprotocol support. When the application uses multiprotocol RAIL library, the SLC tools automatically include relevant initialization for Bluetooth multiprotocol support.

3.7 Platform Components

The SDK v3.x and later rely on many platform components that are part of the underlying Gecko Platform infrastructure of the Gecko SDK Suite. The `autogen` folder contains the source code generated by SLC tools for initializing the hardware and processing events. The `config` folder includes hardware and stack configuration options.

4. Configuring the Bluetooth Stack and a Wireless Gecko Device

To run the Bluetooth stack and an application on a Wireless Gecko, the MCU and its peripherals must be properly configured. The application project configuration consists of selecting the Platform and Bluetooth components that the application needs and setting the configurable values for each selected component. Application projects are generated using Silicon Labs Configurator (SLC) tools that read the project description from a .slcp file and generate the relevant files and build rules. An application can be generated using either the Project Configurator in Simplicity Studio 5 (SSv5) or the SLC-CLI (see *UG520: Software Project Generation and Configuration with SLC-CLI*). The files generated by the SLC tools take care of initializing and setting the configuration for the selected components.

4.1 Wireless Gecko MCU and Peripherals Configuration

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, which are located in the `autogen` folder.

App_init()

The `App_init()` function is used to initialize application-specific features.

4.1.1 Bluetooth Clocks

The clock settings are initialized in the `sl_platform_init()` function in `sl_event_handler.c`. Clock settings include initializations of oscillators (HFXO, LFXO, and LFRCO) with parameters such as tuning, initialization of the clocks (HFCLK, LFCLK, LFA, LFB, LFE), and the assignment of clocks to oscillators. Note: The peripheral clocks (like GPIO clock, TIMER clock) are not enabled in this function. They must be enabled when initializing a peripheral.

HFCLK

HFCLK is used for a radio protocol timer (PROTIMER). HFCLK is a high frequency clock where accuracy must be at least ±50 ppm. This clock needs an external crystal to be sufficiently accurate (HFXO).

The HFXO initialization configures the external crystals for timing-critical connection and sleep management. An HFXO has to be set as the high frequency clock (HFCLK) and physically connected to a Wireless Gecko's HFXO input pins.

LFCLK

LFCLK, the low frequency clock, is used for two purposes. In the Bluetooth stack, it is used for Bluetooth protocol timing. It is also needed to keep track of time during sleep mode.

When a device enters into sleep mode, the current state of PROTIMER is saved. When the device wakes up, it calculates how many ticks of sleep clock have passed and adjusts the PROTIMER accordingly. To the radio it appears that PROTIMER has been constantly ticking.

The accuracy of this clock depends on the operating mode of the device. When advertising or scanning, accuracy is not that important, but when a connection is open, the accuracy must be at least ±500 ppm. This clock can be driven either by LFXO, PLFRCO (EFR32[B|M]G13 or [M|B]GM13), or LFRCO (EFR32[M|B]G2x or [M|B]GM2x), depending on the accuracy requirements. If applications only require advertising or scanning, LFRCO can be used as the clock source. However, if Bluetooth connections are required, the clock source must be either LFXO, PLFRCO (EFR32[M|B]G13 or [M|B]GM13) or LFRCO with High Precision Mode (EFR32[M|B]G22 or [M|B]GM22). When using PLFRCO or LFRCO, the accuracy of the clock must be configured to ±500 ppm.

In the default configuration, LFXO is connected to the Wireless Gecko and set as the clock source for LFCLK. If the design only has PLFRCO or LFRCO with High Precision Mode, PLFRCO or LFRCO is connected and set as the clock source.

If none of LFXO, PLFRCO, or LFRCO with High Precision Mode is connected in the design, sleeping is disabled automatically if LF clock accuracy does not meet the 500 ppm requirement.

HFRCODPLL

HFRCODPLL is a high frequency clock that is used as a system clock with the Bluetooth stack in EFR32[M|B]G2x devices. On EFR32[M|B]G21x, HFRCODPLL needs to be configured to 80 MHz and set as the system clock source.

```
CMU_HFRCODPLLBandSet(cmuHFRCODPLLFreq_80M0Hz);
CMU_ClockSelectSet(cmuClock_SYSCLK, cmuSelect_HFRCODPLL);
```

CTUNE

The examples have the crystal tune (CTUNE) settings for both HFXO and LFXO set by default to work with all of the Silicon Labs' Bluetooth modules, reference designs, and radio boards. However, in some cases the end-product design requires specific crystal calibration, either per device or per design. The CTUNE value can be adjusted according to the design in the `sl_device_init_hfxo()` function.

For more information on configuring the HFXO and LFXO, refer to the EFR32 Reference Manual.

Default HFXO CTUNE Value

The system checks multiple sources for the default HFXO CTUNE value, using the following logical order:

1. CTUNE PSKEY is set. This key has ID `50` (`32` in hex) and contains 2 bytes of data for the 16 bit CTUNE value. This can be programmed with the BGAPI command `sl_bt_nvm_save`.

2. Calibration value exists in DEVINFO. Some modules contain a factory-programmed value in the DEVINFO-page.

3. Manufacturing token exists in the user data page. This is programmed by the developer, or it can be automatically set by Simplicity Studio if the board EEPROM contains the value. This token consists of 2 bytes, located at offset 0x0100 from the starting address of the User Data page. Refer to the EFR32 Reference Manual for your specific EFR variant for the full flash mapping.

4. If a radio board is selected when generating the project, then use default value from board header file.

5. If nothing else is found, use the default value from CMU header file.

4.1.2 DC-DC Configuration

On devices that have DC-DC, the configuration is set in the `sl_device_init_dcdc()` function in sl_event_handler.c. The examples in the SDK have DC-DC configuration set to work with the Silicon Labs' Bluetooth modules, radio boards, and reference designs, but custom designs might require specific DC-DC settings. These custom settings can be set in `sl_device_init_dcdc_xx.c`..

```
/** DCDC regulator initialization structure. */
typedef struct {
  EMU_DcdcMode_TypeDef              mode;             /**< DCDC mode. */
  EMU_VreginCmpThreshold_TypeDef  cmpThreshold;     /**< VREGIN comparator threshold. */
  EMU_DcdcTonMaxTimeout_TypeDef   tonMax;           /**< Ton max timeout control. */
  bool                              dcmOnlyEn;        /**< DCM only mode enable. */
  EMU_DcdcDriveSpeed_TypeDef       driveSpeedEM01;   /**< DCDC drive speed in EM0/1. */
  EMU_DcdcDriveSpeed_TypeDef       driveSpeedEM23;   /**< DCDC drive speed in EM2/3. */
  EMU_DcdcPeakCurrent_TypeDef      peakCurrentEM01;  /**< EM0/1 peak current setting. */
  EMU_DcdcPeakCurrent_TypeDef      peakCurrentEM23;  /**< EM2/3 peak current setting. */
} EMU_DCDCInit_TypeDef;
```

For more information on configuring the DC-DC, refer to the EFR32 Reference Manual, Chapter 11, and *AN0948: Power Configurations and DC-DC*.

4.1.3 LNA

A low-noise amplifier (LNA) is an electronic amplifier that amplifies a very low-power signal without significantly degrading its signal-to-noise ratio. The LNA improves RF sensitivity.

An LNA is provided on-board in some MGM12P modules as part of front-end module (FEM). To use LNA in these modules, the FEM needs to be correctly configured and enabled. The FEM is configured in *sl_fem_util_config.h*.

FEM is initialized in `sl_fem_util_init()` within the `sl_service_init()` function if the board supports FEM.

4.1.4 PTI

PTI (Packet Trace Interface) is a built-in block in the Wireless Gecko SoCs to route incoming and outgoing radio packets as raw data to the debug interface. These packets can then be captured and displayed in Simplicity Studio's Network Analyzer. Network Analyzer has a decoder for Bluetooth packets and can be used to debug, analyze, and measure Bluetooth networks.

PTI is initialized in `sl_rail_util_pti_init()` within the `sl_stack_init()` function. The baudrate can be set using the SL_RAIL_UTIL_PTI_BAUD_RATE_HZ definition, and pins can be configured using the definitions with the SL_RAIL_UTIL_PTI_DOUT_ and SL_RAIL_UTIL_PTI_DFRAME_ prefix in *sl_rail_util_pti_config.h*.

4.1.5 Transmit Power

Transmit power of Bluetooth depends on the maximum power allowed by the radio, the software configuration, RF path gain compensation, and usage of Adaptive Frequency Hopping (AFH).

The ETSI EN 300 328 standard requires using AFH when transmitter power is +10 dBm and over.

The maximum allowed power is limited to less than +10 dBm if prevented by adaptivity requirements. The ETSI standard requires that at least 15 channels are in use for AFH. This requirement prevents using +10 dBm and over in the following cases: legacy advertising, scan responses, and in connections, when not enough channels are available.

4.1.6 Wi-Fi coexistence

Wi-Fi coexistence (COEX) is a protocol where Bluetooth and Wi-Fi arbitrate which protocol can use the radio for transmitting. When enabled, it improves the performance of Wi-Fi and Bluetooth. The application can include the COEX functionality by including the `rail_util_coex` component and configuring it in *sl_rail_util_coex_config.h*.

4.1.7 Mbedtls

The Mbedtls cryptography library used by the stack is configured using a configuration file that defines which algorithms are supported, and if the implementation uses hardware acceleration or is done on software. EFR32[M|B]G2x devices use the new PSA crypto API for crypto operations, whereas EFR32[M|B]G1x devices continue to use the classic Mbedtls API. In addition to enabling crypto operations, the PSA crypto API enables storing long-term encryption keys encrypted on flash in Vault-enabled devices.

The Mbedtls needs to be initialized with `sl_mbedtls_init()`. The Mbedtls configuration file path is given using `#define MBEDTLS_CONFIG_FILE`. The default configuration files `config/mbedtls_config.h`, `autogen/mbedtls_config_autogen.h`, `config/psa_crypto_config.h`, and `autogen/psa_crypto_config_autogen.h` should be used as a template if the configuration needs to be changed. The two latter config files are only used with EFR32[M|B]G2x devices.

In PSA crypto API, only a certain number of keys can be open at one time. Bluetooth pairing requires that 2 keys are open at the same time. By default, no key slots are reserved for the application to save RAM. If the application uses PSA crypto API, then the `SL_PSA_KEY_USER_SLOT_COUNT` setting must be set to the value of the number of keys the application needs to stay open simultaneously. This can be changed with the `SL_PSA_KEY_USER_SLOT_COUNT` setting located in `config/psa_crypto_config.h`. Each key slot will use 40 bytes of RAM.

With EFR32[M|B]G1x devices, the project must also contain *sl_bt_mbedtls_context.c*, which is provided as source in the SDK. It is used by the stack to get Mbedtls context sizes, which depend on the Mbedtls configuration used.

If any Mbedtls errors occur when the Bluetooth stack is using crypto operations, `sl_bt_evt_system_error` is sent with the status set as `SL_STATUS_BT_CRYPTO` and the data field containing the actual Mbedtls error code.

Note that the actual Bluetooth connection encryption uses RADIOAES, which does not have DPA countermeasures. RADIOAES only has access to temporary session keys.

4.2 Bluetooth Stack Configuration

The Bluetooth stack core component, `bluetooth_stack`, is available in all versions of the Bluetooth SDK and must always be included when the application wants to use Bluetooth functionality. The Bluetooth feature components represent features and functionalities that an application may optionally include into the build. To minimize the code size and flash usage, applications are encouraged to only include those feature components that the application requires. Most Bluetooth feature components correspond to a BGAPI class in the Bluetooth API. When the corresponding component is not included in the application project, the API class is not available, and its commands will return an error at runtime.

The set of available Bluetooth feature components can vary depending on the Bluetooth SDK version. Consult the API documentation and the components in the SDK that you are working with. The available configuration options may also differ between Bluetooth SDK versions. Some Bluetooth configuration parameters are part of the core component, `bluetooth_stack`, while some parameters are part of the optional feature component. Each component that provides configurable parameters has an associated configuration header. Consult the configuration headers or the Project Configurator in the Simplicity Studio to see the description of each configuration parameter.

The subsections below describe some Bluetooth components or functionalities that affect Bluetooth as a whole or have special configuration considerations.

4.2.1 Bluetooth On-Demand Start

With the Bluetooth on-demand start feature, the application can start and stop the Bluetooth stack from running when needed. The feature is enabled by including the `bluetooth_on_demand_start` component. When this feature is enabled, the Bluetooth stack does not run until `sl_bt_system_start_bluetooth()` is called. The main purpose of this feature is for the DMP use case, where Bluetooth is not needed all the time, and resources need to be freed for other application uses. The Bluetooth stack can be stopped with `sl_bt_system_stop_bluetooth()`, which gracefully restores Bluetooth to an idle state by disconnecting any active connections and stopping any ongoing advertising and scanning. Any resources that were allocated when the stack was started are freed when the stack is stopped. When the Bluetooth stack is not running, all BGAPI classes other than System become unavailable.

If this feature is not enabled, Bluetooth stack is started automatically at boot time.

4.2.2 Bluetooth Buffer Memory

The Bluetooth stack uses memory for buffering API events and the data transmitted in Bluetooth connections, advertising, and scanning. This buffer memory is allocated from the heap when the Bluetooth stack is started. The size of buffer memory in bytes is defined by C-define `SL_BT_CONFIG_BUFFER_SIZE` in *sl_bluetooth_config.h*. The default value is an estimation for achieving adequate throughput and supporting multiple simultaneous connections. Consider increasing this value if the application needs higher data throughput over connections, or uses advertising or scanning with long advertisement data.

4.2.3 Number of Connections

The absolute maximum number of simultaneous Bluetooth connections is 32. The amount of memory that is allocated for connection management further limits the number of connections. The memory is allocated from the heap when the Bluetooth stack is started. C-define `SL_BT_CONFIG_MAX_CONNECTIONS` in *sl_bluetooth_config.h* can be defined to set the number of connections.

4.2.4 Software Timers

The maximum number of available software timers can be configured by C-define `SL_BT_CONFIG_MAX_SOFTWARE_TIMERS` in *sl_bluetooth_config.h*. Each timer needs resources from the stack to be implemented. Increasing the number of soft timers may cause degraded performance in some use cases.

4.2.5 TX Power and RF Path

TX Power

The system scope maximum TX power for Bluetooth can be configured by C-define SL_BT_CONFIG_MAX_TX_POWER. It specifies the maximum TX power for Bluetooth connections, advertising, scanning, and DTM testing.

The C-define SL_BT_CONFIG_MIN_TX_POWER is used only by the LE Power Control feature. It specifies the minimum TX power level for Bluetooth connections and DTM testing.

Gain

The application can define RF path gain values for RX and TX separately. Positive values mean gain on the given RF path, while negative values mean loss.

The Bluetooth stack takes TX RF path gain into account when setting TX power. The TX power is automatically adjusted so that the power radiated from the antenna matches the application request. For example, if maximum power requested by the application is at +10 dBm and path loss is -1 dB,then actual power at the RF pin is set to +11 dBm.

The TX RF path gain must be set with care and correspond to reality. The stack put limits on the TX power to comply with RF regulations. If the TX RF path gain is not set properly, the device may violate the regulations and may not pass RF certification!

Note: This setting is not meant for modules with integral antennas and should be ignored for such devices.

RX RF path gain is used to compensate the RSSI reports from the Bluetooth Stack.

```
.rf.tx_gain = -20; // RF TX path gain in unit of 0.1 dB. -20 means -2 dB loss on the TX RF path.
.rf.rx_gain = -18; // RF RX path gain in unit of 0.1 dB. -18 means -1.8 dB loss on the RX RF path.
```

Output selection

On EFR32[M|B]G21 SoC-based designs, the RF output can be selected.

```
.rf.flags = SL_BT_RF_CONFIG_ANTENNA; // enabling output configuration
.rf.antenna = 0; // desired output,
```

For the correct value refer to the antenna path selection in the RAIL header file `rail_chip_specific.h`.

4.2.6 Security Manager

To enable the Security Manager in the Bluetooth stack, the application needs to include the `bluetooth_feature_sm` component into the application project. It is important to configure the Security Manager properly to ensure that security requirements can be achieved. Use `sl_bt_sm_configure()` to set security flags and device's IO capabilities. The IO capabilities define which pairing methods are possible and should be set to match the device's capabilities. The security flags can be used to enforce certain security settings, such as requiring that pairing always uses bondable mode, pairing only uses secure connections, or whether authentication is required. Security Manager also has a flag to indicate whether authenticated or non-authenticated pairing is preferred if both are possible. If this flag is not set, and even if both devices' IO capabilities allow authenticated bonding, it is not used if neither device requests authentication. Refer to the available options in API documentation, Security Manager.

The number of bondings that can be stored in the bonding database is set with `sl_bt_sm_store_bonding_configuration()`. This command is also used to define what happens if the bonding database becomes full. Pairings are not stored in the bonding database unless both devices are in bondable mode. Enable the bondable mode with `sl_bt_sm_set_bondable_mode()`.

4.2.7 Accept List Filtering

To enable accept list filtering in the Bluetooth stack, the application needs to include the `bluetooth_feature_whitelisting` component in the application project. When the functionality is included, it can be enabled and disabled at runtime by the BGAPI command, `sl_bt_gap_enable_whitelisting()`.

Accept list filtering is used to filter devices. Currently, it is only supported when discovering devices. Connection requests, scan requests from remote devices during advertising, and connection initiations are not restricted by the accept list.

Accept list size matches the configuration for the max number of bonded devices. If the max number of bonded devices is changed when using filtering, the device needs to be reset before the new setting takes effect.

Bonded devices are added to the accept list automatically. Alternatively, they can be added manually with the BGAPI command, `sl_bt_sm_add_to_whitelist()`.

Random address resolving is not supported. Devices using resolvable random addresses will not be visible during scanning. Since most Android and iOS phones use resolvable random addresses, the accept list filtering feature will effectively block these devices during device discovery.

Connections may be restricted to only bonded or accept listed devices separately using `sl_bt_sm_configure()`. This does not require enabling accept list filtering.

4.2.8 Adaptive Frequency Hopping

Bluetooth Stack implements Adaptive Frequency Hopping (AFH), conforming with the ETSI EN 300 328 standard. AFH is required when using transmit power +10 dBm and over. AFH may also provide performance improvement by avoiding congested channels.

To enable AFH in the Bluetooth stack, the application must include the `bluetooth_feature_afh` component. In a central-peripheral connection, both ends can use AFH independent of each other. The central device may be non-adaptive, but the peripheral still may need to be adaptive. The standard allows using control transfer on a blocked channel. For compliance reasons, if the peripheral detects that a blocked channel is in use, it will only send a single packet on that channel to prevent connection timeouts.

Note: Legacy advertising does NOT use Adaptive Frequency Hopping. Legacy advertising uses 3 channels, and AFH needs a minimum of 15 channels to fulfill the requirements of the ETSI standard. Extended advertising must be used to enable AFH with advertising.

4.2.9 Even Connection Distribution Algorithm

The even connection distribution algorithm is designed to be used especially with applications that involve several concurrent connections. The algorithm tries to distribute the connections such a way that they are distributed over time as evenly as possible without overlapping, and all connections should get an equal share of the air interface resource.

For optimal performance, the algorithm user should:

- Initiate the first connection with the longest connection interval if all connections do not have the same interval.
- Set the connection intervals of the other connections such that they are, or allow (via min-max range), integer fractions of the first interval.
- Make the first interval long enough such that all connections would fit within the interval with a reasonable transmission time.

The algorithm and the connections can be expected to work if the above recommendations are not followed, but performance will not likely be optimal.

By default, the link layer uses the legacy Random Connection Distribution algorithm. The Even Connection Distribution algorithm can be enabled by including the component `bluetooth_feature_ll_even_scheduling` or calling link layer function `ll_connSchAlgorithmEvenEnable()` during the software initialization phase. As the even connection scheduling mechanism is meant to be used with multiple (up to 32) concurrent connections, the buffer and heap sizes are recommended to be increased as follows.

```
SL_BT_CONFIG_BUFFER_SIZE 20160
SL_HEAP_SIZE 22520
```

4.2.10 Multiprotocol Priority Configuration

When the Bluetooth stack is used with other protocols in a multiprotocol environment, it may become necessary to change the Bluetooth priority settings for RAIL to optimize certain use cases.

The application needs to allocate the configuration struct and provide it for the Bluetooth stack:

```
sl_bt_bluetooth_ll_priorities  custom_priorities;
static const sl_bt_configuration_t config = {
  //
  .bluetooth.linklayer_priorities = &custom_priorities,
  //
};
```

The `sl_bt_bluetooth_ll_priorities` struct must be initialized to default state by the `SL_BT_BLUETOOTH_PRIORITIES_DEFAULT` constant.

The `sl_bt_bluetooth_ll_priorities` struct contains following fields:

- `scan_min, scan_max, scan step` - The priority range for scan operation.
- `adv_min, adv_max, adv step` - The priority range for advertisement operation.
- `conn_min & conn_max` - The priority range for connection packets.
- `init_min & init_max` - The priority range for connection initiation.
- `rail_mapping_offset` - The RAIL priority level where Bluetooth priorities are located.
- `rail_mapping_range` - The RAIL priority range where Bluetooth priorities are located.

For each priority range, 0 is the maximum priority, and 0xff is the minimum priority. Bluetooth priorities are different from RAIL priorities. That is, Bluetooth has its own space between 0 and 0xff where all Bluetooth priorities are located. To map Bluetooth priorities to RAIL priorities, the values in fields `rail_mapping_offset` and `rail_mapping_range` are used to form single-degree equation:

```
RAIL_priority=(BT_priority/0xFF)*rail_mapping_range+rail_mapping_offset
```

4.2.11 Sleep

Wireless Gecko's sleep mode EM2 (energy mode two) is managed by the platform power manager component. Including the power manager component or calling the `sl_power_manager_init()` function automatically enables deep sleep.

The sleep modes require that an accurate 32 kHz low-frequency clock (LFCLK) is present in the hardware. If an accurate sleep clock is not available for the Bluetooth stack and the application must support Bluetooth connections or periodic advertising synchronizations, then low power sleep modes cannot be entered. For applications where low power sleep modes are not needed, the LFXO or LFRCO can be left out.

Disabling Sleep at Runtime

If the application needs to disable sleep at runtime, it can be done by implementing bool `app_is_ok_to_sleep()` function. The function is called when the device wants to sleep. While EM2 is disabled (/blocked), the stack will switch between EM0 and EM1. For more information, refer to Power Manager documentation.

4.2.12 PA

On EFR32 SoC-based designs, the Power Amplifier (PA) configuration comes from component `rail_util_pa`, the utility to aid with RAIL RF PA Support.

4.2.13 NVM3 Error Codes

The Bluetooth stack maps NVM3 error codes to the corresponding `sl_status` code if one exists. Other NVM3 error codes are mapped using base value 0x480 + NVM3 error value. The NVM3 error values can be found from platform/emdrv/nvm3/inc/nvm3.h. For example, `ECODE_NVM3_ERR_ALIGNMENT_INVALID` would be mapped as 0x481.

4.3 OTA Configuration

4.3.1 OTA Configuration for EFR32[M|B]G1x devices

Bluetooth Over-the-Air (OTA) firmware upgrades are supported because part of the firmware upgrade is handled by the Bluetooth AppLoader application. On EFR32[M|B]G1x devices, enable OTA configuration with the `bluetooth_feature_ota_config` component. On EFR32[M|B]G2x devices, AppLoader was merged with Gecko Bootloader.

The OTA mode can be configured using the `sl_bt_ota_set_configuration()` function, which can, for example, set OTA to use a static random address, instead of a public address. For other options, refer to the BGAPI document.

When the Wireless Gecko is in AppLoader's OTA mode, its device name and the device name length can be configured with the `sl_bt_ota_set_device_name()` function. The advertisement data used in OTA mode can be set to use custom data instead of the default one with `sl_bt_ota_set_advertising_data()`.

If the device is not using the default RF path, it can be configured for OTA mode with `sl_bt_ota_set_rf_path()`.

Finally, setting the device to OTA DFU mode should be secured so that only trusted devices have that capability.

For more details about OTA firmware updates, refer to *UG266: Silicon Labs Gecko Bootloader User's Guide* and *AN1086: Using the Gecko Bootloader with Silicon Labs Bluetooth Applications*.

4.3.2 OTA Configuration for EFR32[M|B]G2x devices

On EFR32[M|B]G2x devices, configuration of the AppLoader is done directly by modifying the provided AppLoader source file `btl_apploader_common.c` in the bootloader project.

Note that setting the device to OTA DFU mode should be secured so that only trusted devices have that capability.

For more details about OTA firmware updates, refer to *UG266: Silicon Labs Gecko Bootloader User's Guide* and *AN1086: Using the Gecko Bootloader with Silicon Labs Bluetooth Applications*.

4.4 Radio Co-Processor Mode

The Bluetooth controller can be used in Radio Co-Processor (RCP) mode with HCI interface. The link layer must be configured with vendor-specific HCI commands to allocate dynamic memory structures before the link layer can be used.

The following commands are only available when the link layer is using HCI interface. Setting the value to 0 deallocates the respective objects.

`VS_SiliconLabs_Deinit` deinitializes all memory allocated by the link layer. It does not send a response to the command as there is no memory to send it. The reset command must be sent before sending this command.

Command	Opcode	Parameter
VS_SiliconLabs_Allocate_ Connections	0xfc20	uint8_t num_connections
VS_SiliconLabs_Allocate_ Advertisers	0xfc21	uint8_t num_advertisers
VS_SiliconLabs_Allocate_ Addresses	0xfc22	uint8_t num_addresses
VS_SiliconLabs_Allocate_ PeriodicAdv	0xfc23	uint8_t num_periodicadv
VS_SiliconLabs_Allocate_ PeriodicScan	0xfc24	uint8_t num_periodicscan
VS_SiliconLabs_Deinit	0xfc25	(no parameters)

5. Bluetooth Stack Event Handling

The Bluetooth stack for the Wireless Geckos is an event-driven architecture, where events are handled in the main while loop of the application on bare metal.

5.1 Event Listener with RTOS

By default, the event handling with RTOS calls `sl_bt_on_event()` when events are received the same way as without RTOS.

If the application needs to define its own Bluetooth event handler it needs to define SL_BT_DISABLE_EVENT_TASK. The application can then use `sl_bt_rtos_has_event_waiting()` to check if any events are waiting. To process events, call `sl_bt_rtos_get_event()` and `sl_bt_rtos_set_event_handled()` is used mark the event has been handled.

5.2 Commands from Multiple Tasks

It is possible to send Bluetooth commands from multiple OS tasks. Beginning with Gecko SDK v3.1.2, all BGAPI command functions have automatic locking to make them thread-safe. Using `sl_bt_bluetooth_pend()` and `sl_bt_bluetooth_post()` is therefore no longer required for individual calls to the BGAPI.

The application only needs to use `sl_bt_bluetooth_pend()` and `sl_bt_bluetooth_post()` to protect sections of code where multiple commands need to be performed atomically in a thread-safe manner. This includes cases such as using `sl_bt_system_data_buffer_write()` to write data to the system buffer followed by a call to `sl_bt_extended_advertiser_set_long_data()` to set that data to an advertiser set. To synchronize access to the shared system buffer, the application needs to lock by calling `sl_bt_bluetooth_pend()` before `sl_bt_system_data_buffer_write()`, and release the lock by calling `sl_bt_bluetooth_post()` after `sl_bt_extended_advertiser_set_long_data ()`.

6. Interrupts

Interrupts create events in their respective interrupt handlers, be it radio interrupts or interrupts from IO pins. The events are later processed in the main event loop from the message queue. The application should always minimize the processing time within an interrupt handler, and leave the processing for event callbacks or to the main loop.

In general, the interrupt scheme is according to any event-based programming architecture, but a few unique and important exceptions apply to the Bluetooth stack:

- BGAPI commands cannot be called from interrupt context.
- Only the `sl_bt_external_signal()` function can be called from interrupt context.

6.1 External Event

An external event is used to capture all peripheral interrupts as an external signal to be passed to the main event loop and to be processed within that loop. The external event interrupt can come from any of the peripheral interrupt sources, for example IOs, comparators, or ADCs, to name a few. The signal bit array is used for notifying the event handler of what external interrupts have been issued.

- The main purpose of the external signal is to trigger an event from the interrupt context to the main event loop.
- The BGAPI event `sl_bt_evt_system_external_signal` can be generated by calling the `void sl_bt_external_signal(uint32 signals)` function.
- The function `sl_bt_external_signal` can be called from the interrupt context.
- The `signals` parameter of the `sl_bt_external_signal` function is passed to the `sl_bt_evt_system_external_signal` event.

```
/***************************************************************************//**
* Bluetooth stack event handler.
*
* @param[in] evt The event coming from the Bluetooth stack
*****************************************************************************/
void sl_bt_on_event(sl_bt_msg_t *evt)
{
  ...
  switch (SL_BT_MSG_ID(evt->header)) {
    ...

    case sl_bt_evt_system_external_signal_id:
      // External signal indication (comes from the interrupt handler)
      // Handle GPIO IRQ and do something
      // External signal command’s parameter can be accessed using
      // evt->data.evt_system_external_signal.extsignals
      break;

    ...
  }
}

/**
* Handle GPIO interrupts and trigger system_external_signal event
*/
void GPIO_ODD_IRQHandler()
{
  static bool radioHalted = false;

  uint32_t flags = GPIO_IntGet();
  GPIO_IntClear(flags);

  //Send gecko_evt_system_external_signal_id event to the main loop sl_bt_external_signal(...);
 }
}
```

6.2 Priorities

It is highly recommended that the radio should have the highest priority interrupts. This is the default configuration, and other interrupts are handled with lower priority. Interrupt priorities for radio is 4, for Link Layer the priority is 5, and other interrupts have default priority of 7. Smaller value is higher priority interrupt.

If the application needs to disable interrupts, it is recommended that the `BASEPRI` register is used instead of the `PRIMASK` register. The `BASEPRI` register disables with interrupt priority, whereas `PRIMASK` disables all interrupts. EMLIB Core can be configured to use the `BASEPRI` register, and it can then be used with the `CORE_ENTER_ATOMIC()` and `CORE_EXIT_ATOMIC()` macros, which will disable interrupt priorities 3 and lower. See Core Interrupt documentation for more information.

Without RTOS, Link Layer uses PendSV for achieving priority over the application software. With RTOS the Link Layer will not use PendSV, but Link Layer task will have higher priority over application task. RTOS scheduler will then give priority to Link Layer task over application task.

The following table describes the three different components within the Bluetooth stack that run in different operating contexts, and their maximum time to disable interrupts in order for each component to assure connections.

Component	Description	Timing Accuracy	Operating Context	Maximum IRQ Disable	If Timing Requirements are Ignored
Radio	Time-critical low level TX/RX radio control	Microseconds	Radio IRQ	< ~10 µs	Packets are not transmitted or received, which will eventually cause supervision timeout and Bluetooth link loss.
Link layer	Time-critical connection management procedures and encryption	Milliseconds	PendSV IRQ (1)	< ~20 ms	If the link control procedure is not handled in time, Bluetooth link loss may happen. Peripheral-side channel map update and connection update timings are controlled by central device.
Host Stack	Bluetooth Host Stack, Security Manager, GATT	Seconds	Application	< 30 s	SMP and GATT have a 30 s timeout and if operations are not handled within that timeout Bluetooth link loss will occur.

(1) PendSV interrupt is only used without RTOS

7. Wireless Gecko Resources

The Bluetooth 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 (in red) 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 timings	The sleep timer uses RTCC in the default configuration. In EFR32[M|B]G13 and EFR32[M|B]G22, RTCC can be used by applications if the sleep timer is configured to use another resource. See platform sleeptimer documentation.
	PROTIMER	Bluetooth	The application does not have access to PROTIMER.
Radio	**RADIO**	Bluetooth	Always used and all radio registers are reserved for the Bluetooth stack.
GPIO	NCP	Host communication.	2 to 6 x I/O pins can be allocated for the NCP usage depending on used features (UART, RTS/CTS, wake-up and host wake-up). Optional to use, and valid only for NCP use case.
	PTI	Packet trace	2 to N x I/O pins. Optional to use.
	TX enable	TX activity indication	1 x I/O pin. Optional to use.
	RX enable	RX activity indication	1 x I/O pin. Optional to use.
	COEX	Wi-Fi coexistence	4 x I/O pin. Optional to use.
CRC	GPCRC	PS Store	Can be used in application, but application should always reconfigure GPCRC before use, and GPCRC clock must not be disabled in CMU.
Flash	MSC	PS Store	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.
	RADIOAES	Bluetooth link encryption	The application does not have access to RADIOAES

7.1 Flash

The application and Bluetooth stack are executed from the flash memory. The flash can be split into blocks for the bootloader, the Bluetooth AppLoader, application + Bluetooth stack, and non-volatile memory, as shown in the following figure.

- The bootloader is essential to enable Bluetooth stack and application upgradeability. The bootloader has been designed to be future-proof for bootloader improvements and feature additions. On devices with separate bootloader flash the bootloader is located there.
- The Bluetooth AppLoader provides OTA upgradability for the application. This is an optional feature, but using it requires that the bootloader is also used. On EFR32[M|B]G2x devices, AppLoader was merged with Gecko Bootloader beginning Bluetooth SDK version 4.0.x.
- PS Store and NVM3 are a non-volatile data stores (NVM), where both the Bluetooth stack and the application can store permanent data, such as Bluetooth bonding keys, application configuration data, hardware configurations, and so on. These cannot be used simultaneously. PS Store is only supported on Series 1 devices.
- The application is located between the Bluetooth AppLoader and NVM. The Bluetooth stack is a library that is linked with the application. The Bluetooth stack includes the actual Bluetooth firmware, including link layer, GAP, SM, ATT, and GATT layers.
- Manufacturing tokens storage is used for storing manufacturing tokens. On EFR32[M|B]G2x devices it is located at end of main flash.

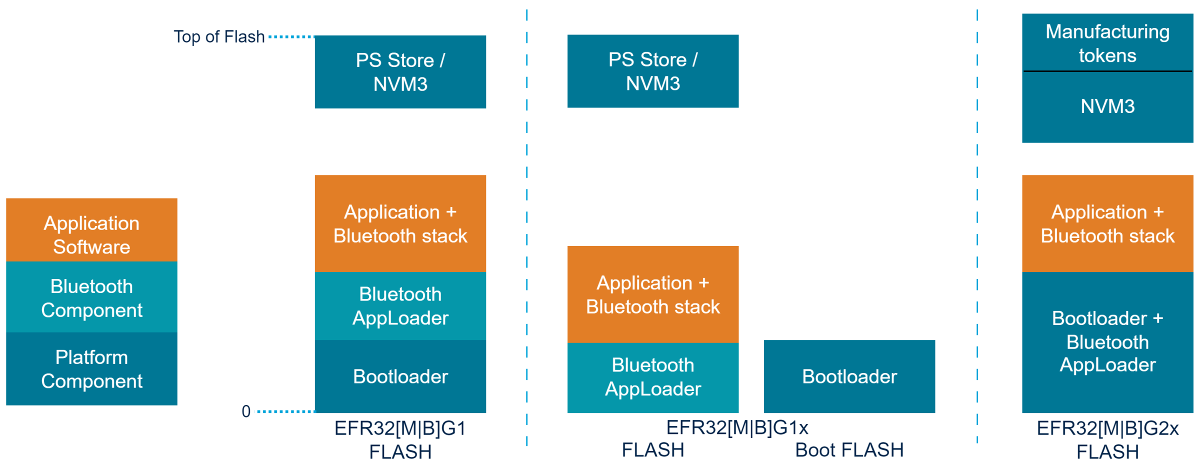


Figure 7.1. Flash Usage With and Without Separate Bootloader Flash

The following table shows the flash usage for each block. The estimates can vary between use cases, configurations, application resources, or SDK version.

	Compiler	EFR32[M|B]G1	EFR32[M|B]G12	EFR32[M|B]G13	EFR32[M|B]G21	EFR32[M|B]G21 with Vault	EFR32[M|B]G22	EFR32[M|B]G24	EFR32[M|B]G27
Bootloader (1)	-	16	16	16	16+56	16+56	24+48	24+48	24+48
Bluetooth AppLoader (1)	-	42	48	50	-	-	-	-	-
soc-empty (2)	GCC	148	154	157	163	166	172	180	173
"	IAR	148	154	157	163	166	172	181	173
PS Store	-	4	4	4	-	-	-	-	-
NVM3 (3)	-	10	10	10	40	40	40	40	40
Manufacturing tokens	-	-	-	-	8	8	8	8	8

(1) Bluetooth AppLoader was merged with bootloader on EFR32[M|B]G2x devices. The bootloader size includes the AppLoader communication plugin. The size in the table for EFR32[M|B]G2x devices is the size reserved for Bootloader and AppLoader by default. The actual size can be smaller.

(2) *soc-empty* is an example application provided in the Bluetooth SDK. It is compiled with high size optimizations. GCC uses the `-Os` flag, and IAR the `-Ohz` flag.

(3) NVM3 is an alternative to PS Store. They cannot be used simultaneously. NVM3 requires a minimum of 3 flash pages; the default configuration in the Bluetooth sample applications is 5 pages in the SDK. Please refer to *AN1135: Using Third Generation Non-Volatile Memory (NVM3) Data Storage* for further information about NVM3.

7.1.1 Optimizing Flash Usage

Dead Code Elimination

Bluetooth stack libraries are designed to benefit from the linker's dead code elimination optimization. With this optimization all unused code will be removed from application.

To fully utilize this optimization feature, it is important not to call any function that is not needed for application. These include all initialization functions for the Bluetooth stack.

Selective Initialization of Bluetooth Stack Components

Each required stack component must be individually initialized. For more information, see section 4.2 Bluetooth Stack Configuration.

7.1.2 Bluetooth Bonding Database

Bluetooth bonding database is stored in NVM. NVM3 size must be set so that the required number of bondings can fit to it. The following table shows how much NVM3 space each bonding will require at maximum in bytes including NVM3 overheads. EFR32[M|B]G1x devices still use use the old bonding database, whereas EFR32[M|B]G2x devices use new PSA ITS (internal trusted store) for storing the keys.

	EFR32[M|B]G1x	EFR32[M|B]G2x	EFR32[M|B]G2x with Vault
Secure Connections Pairing	107	211	299
Legacy Pairing	155	311	443

Note that in EFR32[M|B]G2x devices, during the first boot, the device tries to import keys from the old bonding database used in the SDK v3.1.1 and older into PSA ITS. If IRK (privacy key) import fails, all existing bondings are deleted, because IRK is shared with bonded devices. If importing certain bonding fails, that bonding is erased and importing will continue with the next one.

When deployed to Secure Vault High devices, sensitive keys such as the Long Term Key (LTK) are protected using the Secure Vault Key Management functionality. The table below shows the protected keys and their storage protection characteristics.

Wrapped Key	Exportable/Non-Exportable	Notes
Remote Long Term Key (LTK)	Non-Exportable	-
Local Long Term Key (legacy only)	Non-Exportable	-
Remote Identity Resolving Key (IRK)	Exportable	Must be Exportable for future compatibility reasons
Local Identity Resolving Key (IRK)	Exportable	Must be Exportable because the key is shared with other devices

Wrapped keys that are “Non-Exportable” can be used, but cannot be viewed or shared at runtime. Wrapped keys that are “Exportable” can be used or shared at runtime, but remain encrypted while stored in flash.

When the `bluetooth_feature_external_bonding_database` component is used, the application is responsible for storing the persistent bonding information. The Bluetooth stack will only store the local identity resolving key internally in this case.

7.2 Linking

The Bluetooth stack is delivered as a set of library files. The application links the Bluetooth stack libraries with the rest of application. The linker will then create an ELF-file, which contains the application code and data ready to be loaded into flash.

For generating OTA DFU files, the application's code and data must be linked into their own section in the ELF-file. This is automatically done with the linker files provided with the Gecko Platform.

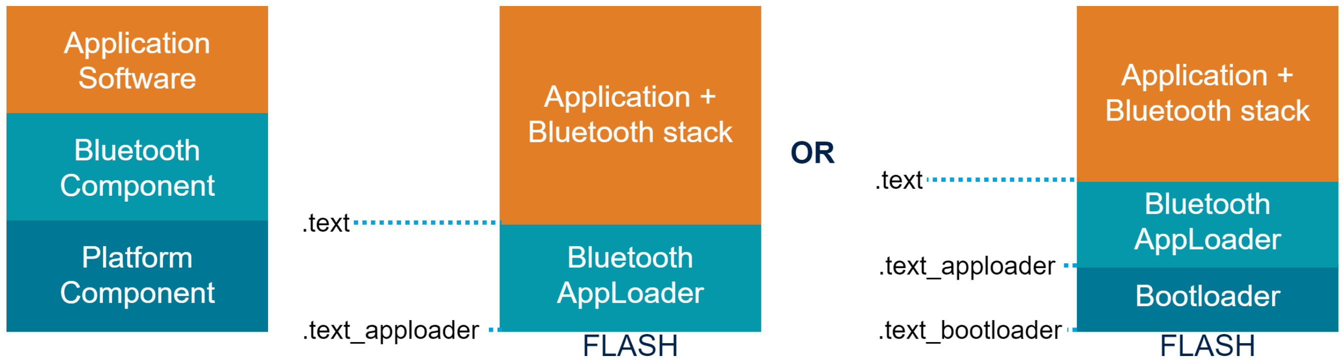


Figure 7.2. Sections Defined in the Linker File and Their Placement

Depending on the device used, the bootloader is placed on separate flash memory or, if no separate bootloader flash exists, the linker file reserves some memory from main flash for the bootloader. On EFR32[M|B]G1x devices, Bluetooth AppLoader is placed at the beginning of main flash and the application with all libraries start from the next free flash page.

For more information on the OTA updates and how to enable them, please refer to *UG266: Silicon Labs Gecko Bootloader User's Guide* and *AN1086: Using the Gecko Bootloader with Silicon Labs Bluetooth Applications*.

7.3 RAM

The Bluetooth stack reserves part of the RAM from the Wireless Gecko and leaves the unused RAM for the application.

RAM consumption of the Bluetooth functionality is divided into:

- Bluetooth stack
- Bluetooth object pools
- Bluetooth buffer memory
- Bluetooth GATT database
- C STACK
- C HEAP

The following table shows the RAM allocations that are done statically at link time.

Component	Static allocation at link time	Configurable by
Bluetooth stack	6 kB	
Bluetooth GATT database	Application-dependent (20 to 200 bytes)	
Call stack	2752 bytes	SL_STACK_SIZE
Heap memory	9200 bytes	SL_HEAP_SIZE

The following table shows the RAM allocations that are done dynamically from the heap at run time.

Component	Dynamic heap allocation at run time	Configurable by
Bluetooth stack	2 kB	
Bluetooth connection objects	1600 bytes = 400 bytes * 4	SL_BT_CONFIG_MAX_CONNECTIONS
Bluetooth advertiser objects	160 bytes = 160 bytes * 1	SL_BT_CONFIG_USER_ADVERTISERS
Bluetooth periodic advertising synchronization objects	0 bytes = 168 bytes * 0	SL_BT_CONFIG_MAX_PERIODIC_ADVERTISING_SYNC
Bluetooth software timers	160 bytes = 40 bytes * 4	SL_BT_CONFIG_MAX_SOFTWARE_TIMERS
Bluetooth buffer memory	3150 bytes	SL_BT_CONFIG_BUFFER_SIZE

7.3.1 Bluetooth Stack

The Bluetooth stack allocates around 6 kB of static RAM and 2 kB of heap for its internal use. It includes Bluetooth stack software with low-level radio drivers and the application programming interface.

7.3.2 Bluetooth Object Pools

The Bluetooth stack uses memory to store the necessary context for objects such as connections, advertisers, and periodic advertisement synchronizations. The number of these objects depends on the configuration. The table in section 7.3 RAM summarizes the memory usage in the default configuration and shows which configuration items affect the number of objects allocated.

7.3.3 Bluetooth Buffer Memory

The Bluetooth stack uses memory for buffering API events and the data transmitted in Bluetooth connections, advertising, and scanning. This buffer memory is allocated from the heap by the Bluetooth stack when the Bluetooth stack is started. The size of buffer memory in bytes is defined by the C-define `SL_BT_CONFIG_BUFFER_SIZE` in *sl_bluetooth_config.h*. The default value is an estimation for achieving adequate throughput and supporting multiple simultaneous connections. Consider increasing this value if the application needs higher data throughput over connections or uses advertising or scanning with long advertisement data.

7.3.4 Bluetooth GATT Database

The Bluetooth GATT database uses statically-allocated RAM. The amount of RAM used depends on the user-defined GATT database and cannot be generalized. All characteristics with write enabled use as much RAM as their length defined. Plus, every attribute in GATT needs a few bytes of RAM for maintaining the Attribute permissions. Typical RAM usage is approximately 20 to 200 bytes.

7.3.5 Call Stack

The Bluetooth stack requires at minimum a call stack to be reserved from RAM as summarized in the table in section 7.3 RAM. Application developers must allocate RAM for the application call stack on top of the memory required by the stack. The size of the call stack is configured by SL_STACK_SIZE in *sl_memory_config.h*.

7.3.6 Heap memory

The Bluetooth stack uses the heap to allocate storage for object contexts and the stack internal state as summarized in the table in section 7.3 RAM. In addition to these allocations, the Bluetooth stack requires heap memory for asymmetric encryption operations using the elliptic curve algorithms during Bluetooth pairing.

The C-define `SL_HEAP_SIZE` in *sl_memory_config.h* defines the minimum heap size that is allocated from the physical RAM at link time. The actual heap size at runtime can end up being larger than the minimum to make use of any available physical memory that would otherwise have remained unallocated.

The default minimum heap size is sufficient for running the Bluetooth examples with the default Bluetooth configuration. The application should configure the minimum heap size to account for the Bluetooth configuration used and any extra heap that the application may require.

8. Application ELF-file

ELF (Executable and Linkable Format) is a standard file format for executable files. This chapter describes the sections in the ELF file related to the application and the Bluetooth stack.

Some linkers provide output describing the consumed flash, but what it contains is not obvious. A Bluetooth project might contain a bootloader and the Bluetooth AppLoader, and the device might have separate flash for the bootloader. The ELF-file provides exact information about RAM and flash usage.

Simplicity Studio provides the GCC toolchain, which contain command line tool *objdump*. This tool can be used to get section information from the ELF-file.

objdump requires input ELF-file. If the parameter `-h` is used, *objdump* dumps the section header information.

IAR

Calling *objdump* from the command line for an example application:

```
arm-none-eabi-objdump -h ewarm-iar/exe/soc_empty.out
```

objdump then gives the following output:

```
Sections:
Idx Name           Size      VMA       LMA       File off  Algn
  0 .text_apploader rw 0000dfc0  00006000  00006000  00000034  2**13
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 application us 000289f0  00014000  00014000  0000dff4  2**8
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 storage_regions rw 0000a000  00074000  00074000  000369e4  2**13
                  ALLOC
  3 application_ram rw 000045c8  20000000  20000000  000369e4  2**9
                  ALLOC
  4 .debug_abbrev 00000015  00000000  00000000  000369e4  2**0
                  CONTENTS, READONLY, DEBUGGING
  5 .debug_aranges 0000001c  00000000  00000000  000369fc  2**0
                  CONTENTS, READONLY, DEBUGGING
  6 .debug_frame  0003bf4f  00000000  00000000  00036a18  2**0
                  CONTENTS, READONLY, DEBUGGING
  7 .debug_info   00000056  00000000  00000000  00072968  2**0
                  CONTENTS, READONLY, DEBUGGING
  8 .debug_line   00000096  00000000  00000000  000729c0  2**0
                  CONTENTS, READONLY, DEBUGGING
  9 .iar.debug_frame 00015f9d  00000000  00000000  00072a58  2**0
                  CONTENTS, READONLY
 10 .comment      000e14ee  00000000  00000000  000889f8  2**0
                  CONTENTS, READONLY
 11 .iar.rtmodel  00000069  00000000  00000000  00169ee8  2**0
                  CONTENTS, READONLY
 12 .ARM.attributes 0000002e  00000000  00000000  00169f54  2**0
                  CONTENTS, READONLY
```

`.text_apploader` contains the Bluetooth AppLoader.

`.text_signature` is the space reserved for the AppLoader signature.

`.text` contains the application code and read-only data. Size of the application in this example is 0x289f0 in hexadecimal, and 166384 bytes in decimal.

`.stack` is a RAM section for the call stack.

`.data` is the RAM section for initialized variables.

`.bss` is the RAM section for uninitialized variables.

`.heap` is the RAM section for heap.

Refer to IAR documentation for description of the remaining sections.

GCC

Calling *objdump* from the command line for an example application:

```
arm-none-eabi-objdump -h build/debug/soc_empty.out
```

objdump then gives the following output:

```
Sections:
Idx Name           Size      VMA       LMA       File off  Algn
  0 .text_apploader 0000dfc0  00006000  00006000  00006000  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .text_signature 00000040  00013fc0  00013fc0  00013fc0  2**0
                  ALLOC
  2 .text         000289dc  00014000  00014000  00014000  2**8
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .ARM.exidx    00000008  0003c9dc  0003c9dc  0003c9dc  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .stack        00000800  20000000  20000000  00050000  2**3
                  ALLOC
  5 .data         00000318  20000800  0003c9e4  00040800  2**2
                  CONTENTS, ALLOC, LOAD, CODE
  6 .bss          00001a74  20000b18  0003ccfc  00040b18  2**9
                  ALLOC
  7 .heap         00001f40  20002590  20002590  00040b18  2**3
                  CONTENTS
  8 .nvm          0000a000  0003c9e4  0003c9e4  00042a58  2**0
                  CONTENTS
  9 .ARM.attributes 00000036  00000000  00000000  0004ca58  2**0
                  CONTENTS, READONLY
 10 .comment      00000076  00000000  00000000  0004ca8e  2**0
                  CONTENTS, READONLY
 11 .debug_frame  000003c0  00000000  00000000  0004cb04  2**2
                  CONTENTS, READONLY, DEBUGGING
```

`.text_apploader` contains the Bluetooth AppLoader.

`.text_signature` is the space reserved for the AppLoader signature.

`.text` contains the application code and read-only data. The size of the application in this example is 0x289dc in hexa-decimal and 166364 bytes in decimal.

`.ARM.exidx` is used for debugging.

`.stack` is a RAM section for the call stack

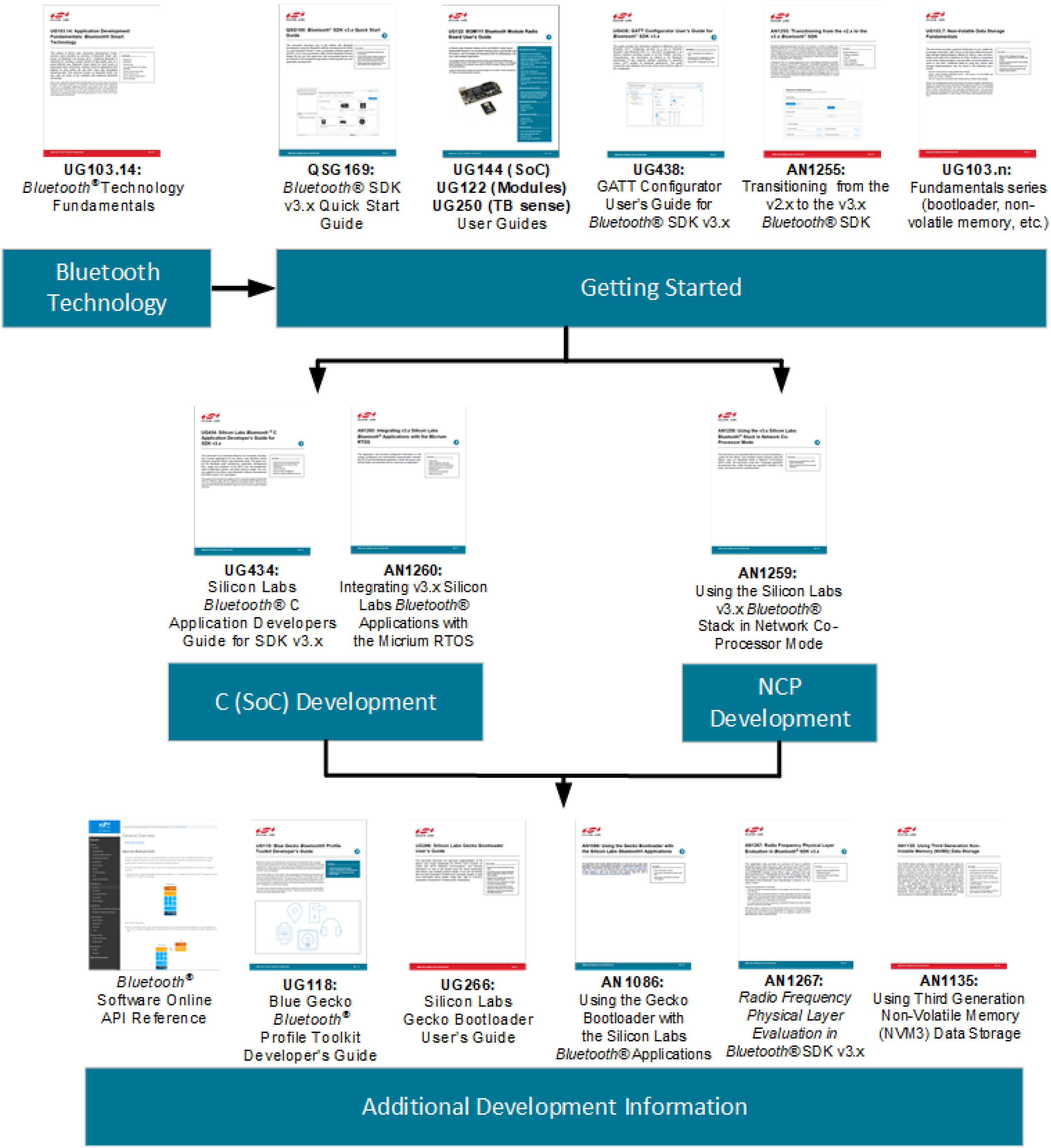
`.data` is the RAM section for initialized variables.

`.bss` is the RAM section for uninitialized variables.

`.heap` is the RAM section for heap.

Refer to GCC documentation for a description of the remaining sections.

9. Documentation



IoT Portfolio
www.silabs.com/products

Quality
www.silabs.com/quality

Support & Community
www.silabs.com/community

Disclaimer
Silicon Labs intends to provide customers with the latest, accurate, and in-depth documentation of all peripherals and modules available for system and software implementers using or intending to use the Silicon Labs products. Characterization data, available modules and peripherals, memory sizes and memory addresses refer to each specific device, and "Typical" parameters provided can and do vary in different applications. Application examples described herein are for illustrative purposes only. Silicon Labs reserves the right to make changes without further notice to the product information, specifications, and descriptions herein, and does not give warranties as to the accuracy or completeness of the included information. Without prior notification, Silicon Labs may update product firmware during the manufacturing process for security or reliability reasons. Such changes will not alter the specifications or the performance of the product. Silicon Labs shall have no liability for the consequences of use of the information supplied in this document. This document does not imply or expressly grant any license to design or fabricate any integrated circuits. The products are not designed or authorized to be used within any FDA Class III devices, applications for which FDA premarket approval is required or Life Support Systems without the specific written consent of Silicon Labs. A "Life Support System" is any product or system intended to support or sustain life and/or health, which, if it fails, can be reasonably expected to result in significant personal injury or death. Silicon Labs products are not designed or authorized for military applications. Silicon Labs products shall under no circumstances be used in weapons of mass destruction including (but not limited to) nuclear, biological or chemical weapons, or missiles capable of delivering such weapons. Silicon Labs disclaims all express and implied warranties and shall not be responsible or liable for any injuries or damages related to use of a Silicon Labs product in such unauthorized applications. **Note: This content may contain offensive terminology that is now obsolete. Silicon Labs is replacing these terms with inclusive language wherever possible. For more information, visit www.silabs.com/about-us/inclusive-lexicon-project**

Trademark Information
Silicon Laboratories Inc.®, Silicon Laboratories®, Silicon Labs®, SiLabs® and the Silicon Labs logo®, Bluegiga®, Bluegiga Logo®, EFM®, EFM32®, EFR, Ember®, Energy Micro, Energy Micro logo and combinations thereof, "the world's most energy friendly microcontrollers", Redpine Signals®, WiSeConnect , n-Link, ThreadArch®, EZLink®, EZRadio®, EZRadioPRO®, Gecko®, Gecko OS, Gecko OS Studio, Precision32®, Simplicity Studio®, Telegesis, the Telegesis Logo®, USBXpress® , Zentri, the Zentri logo and Zentri DMS, Z-Wave®, and others are trademarks or registered trademarks of Silicon Labs. ARM, CORTEX, Cortex-M3 and THUMB are trademarks or registered trademarks of ARM Holdings. Keil is a registered trademark of ARM Limited. Wi-Fi is a registered trademark of the Wi-Fi Alliance. All other products or brand names mentioned herein are trademarks of their respective holders.

SILICON LABS

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

www.silabs.com