UG136: Silicon Labs *Bluetooth* ® C Application Developer's Guide

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 SDK version 2.11.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

Table of Contents

1. Introduction

This document is a C developer's guide for the Silicon Labs Bluetooth stack.

The document 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 and 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 2.11.x.

Currently supported compilers and IDE versions are:

- **IDE:** Simplicity Studio 4.1.9 or newer
- **Compiler:** IAR v8.30.1 and GCC 7.2.1

Critical Changes from Version 2.10.x:

When moving a project from Bluetooth stack version 2.10.x to 2.11.x and higher, the following changes in the Bluetooth stack must be addressed in the project.

Command `cmd_endpoint_close` is removed

Use `command le_connection_close` to close Bluetooth connections.

Command cmd_gatt_server_set_database is removed

Use the GATT capability feature for dynamic configuration of services and characteristics in the local GATT database. See `cmd_gatt_server_set_capabilities` command for the details.

Command `cmd_hardware_enable_dcdc` is removed

DCDC control is not supported by Bluetooth API. Use EMDRV or EMLIB for necessary functions.

Command `cmd_le_gap_bt5_set_adv_parameters` is deprecated

Replacements are:

- `cmd_le_gap_set_advertise_timing` command for setting the advertising intervals,
- `cmd_le_gap_set_advertise_channel_map` command for setting the channel map, and
- `cmd_le_gap_set_advertise_report_scan_request` command for enabling and disabling scan request notifications.

Command `cmd_le_gap_bt5_set_mode` is deprecated

Replacements are:

- `cmd_le_gap_start_advertising` command to start the advertising, and
- `cmd_le_gap_stop_advertising` command to stop the advertising.
- `cmd_le_gap_set_advertise_timing` command can be used for setting the maximum events and
- command `cmd_le_gap_set_advertise_configuration` can be used to for setting address types.

Command `cmd_le_gap_discover` is deprecated

Replacement is `cmd_le_gap_start_discovery` command, which allows scanning on LE 1M PHY or LE Coded PHY.

Command `cmd_le_gap_open` is deprecated

Replacement is `cmd_le_gap_connect` command, which allows opening a connection with a specified PHY.

Command `cmd_le_gap_set_adv_data` is deprecated

Use the `cmd_le_gap_bt5_set_adv_data` command to set the advertising data and scan response data.

Command `cmd_le_gap_set_adv_parameters` is deprecated

Replacements are:

- `cmd_le_gap_set_advertise_timing` command for setting the advertising intervals, and
- `cmd_le_gap_set_advertise_channel_map` command for setting the channel map.

Command `cmd_le_gap_set_adv_timeout` is deprecated

Replacements is `cmd_le_gap_set_advertise_timing`, which should be used for this functionality.

Command `cmd_le_gap_set_mode` is deprecated

Replacements are:

- `cmd_le_gap_start_advertising` command for enabling the advertising, and
- `cmd_le_gap_stop_advertising` command for disabling the advertising

Command `cmd_le_gap_set_scan_parameters` is deprecated

Replacements are:

- `cmd_le_gap_set_discovery_timing` command for setting timing parameters, and
- `cmd_le_gap_set_discovery_type` command for the scan type.

Command `cmd_system_set_bt_address` is deprecated

Replacement is `cmd_system_set_identity_address` command.

Function `gecko_init()` and `gecko_stack_init()` are changed

Functions `gecko_init()` and `gecko_stack_init()` return `errorcode_t`. This error code must be verified to confirm that the Bluetooth stack has been succesfully initialized.

1.2 Prerequisites

This documents 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 *UG104.13: Bluetooth Technology Fundamentals*.

For instructions on getting started using example applications in Silicon Labs Simplicity Studio development environment, see *QSG139: Bluetooth Development with Simplicity Studio*.

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.

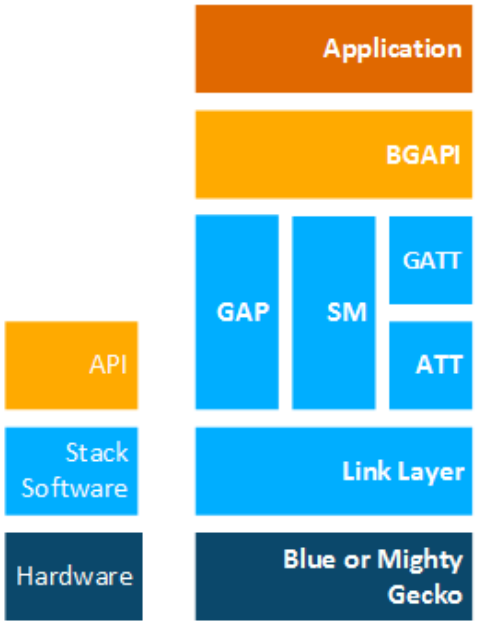


Figure 2.1. Bluetooth Stack Architecture Block Diagram

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.

2.1 Application Build Flow

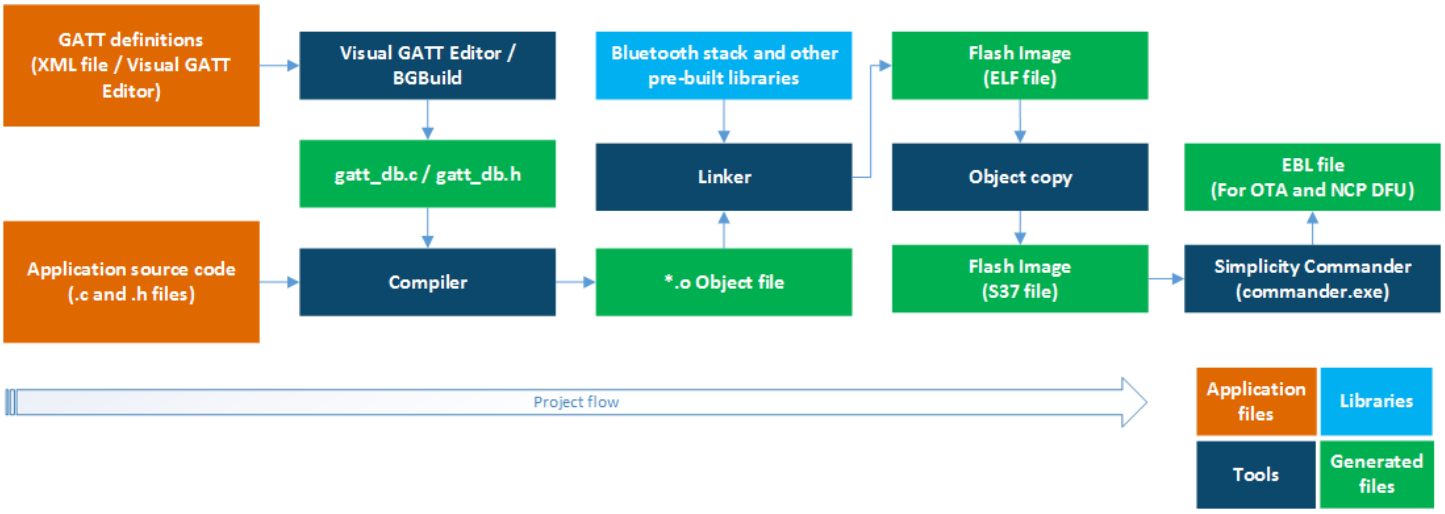


Figure 2.2. Bluetooth Project Build Flow

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 *QSG139: Bluetooth Application Development in Simplicity Studio*.

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* and *.bgproj* 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* or *.bgproj* according to the *UG118: Blue Gecko Bluetooth® Profile Toolkit Developer's Guide* and then use the BGBuild executable 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, provides the optional OTA (Over-the-Air) functionality.
- **libbluetooth.a:** Bluetooth stack library.
- **libmbedtls.a:** mbed TLS cryptographic library for Bluetooth Stack.
- **libcoex.a:** Wi-Fi and Bluetooth coexistence arbitration features for Bluetooth stack. Using this requires hardware support.
- **libpsstore.a:** PS Store functionality for Bluetooth stack. This is not available on EFR32BG2x devices. Instead, the NVM3 must be used. For more information how to use NVM3, see *AN1135: Using Third Generation Non-Volatile Memory (NVM3) Data Storage*.
- **librail.a:** RAIL (Radio Abstraction Interface Layer) library for the Bluetooth stack.

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 environment. RAIL libaries are provided in the Gecko SDK. For more information refer to *UG103.13: RAIL Fundamentals* and other RAIL documentation.

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 need to be included in the project. EMLIB and EMDRV are part of the Gecko SDK. 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\.

Header Files

bg_version.h

This file contains the Bluetooth stack version.

API Header Files

These files defines the Bluetooth stack API. There are three different files for different use cases. Only one of the files must be included. *native_gecko.h* is used with bare-metal Bluetooth application. *ncp_gecko.h* is used when building SoC application for NCP support. *rtos_gecko.h* is used with Micrium RTOS.

These files serve two purposes: first they contain the actual Bluetooth stack API and the commands and events for the stack, and second they provide a configuration, event, and sleep management API to the Bluetooth stack.

The configuration, event, and sleep management API is described below.

```
errorcode_t gecko_init(const gecko_configuration_t*config)
```

This function takes a single argument - a pointer to a gecko_configuration_t struct. Its purpose is to configure and initialize the Bluetooth stack with the parameters provided in the struct. The configuration options and how to use `gecko_init()` are discussed in more detail in section 4.2 Bluetooth Configuration with gecko_stack_init(). `gecko_init()` must be called by the application to initialize the Bluetooth stack.

This function is provided for convenience. It initializes all functionality in Bluetooth stack. To have more granularity in configuration, use `gecko_stack_init()` as described next.

`errorcode_t gecko_stack_init(const gecko_configuration_t*config)`

This function takes a single argument - a pointer to a `gecko_configuration_t` struct. Its purpose is to configure and initialize the Bluetooth stack with the parameters provided in the struct. Once the function `gecko_stack_init()` is called each stack used component must be initialized separately. This separation allows memory optimization, by not including those stack components that are not needed.

The following APIs can be used to initialize stack components separately:

- `gecko_bgapi_class_dfu_init();`
- `gecko_bgapi_class_system_init();`
- `gecko_bgapi_class_le_gap_init();`
- `gecko_bgapi_class_le_connection_init();`
- `gecko_bgapi_class_gatt_init();`
- `gecko_bgapi_class_gatt_server_init();`
- `gecko_bgapi_class_endpoint_init();`
- `gecko_bgapi_class_hardware_init();`
- `gecko_bgapi_class_flash_init();`
- `gecko_bgapi_class_test_init();`
- `gecko_bgapi_class_sm_init();`

`struct gecko_cmd_packet* gecko_wait_event(void)`

This is a blocking function that waits for events coming from the Bluetooth stack and blocks until an event is received. Once an event has been received, a pointer to a `gecko_cmd_packet` struct is returned.

If EM sleep modes have been enabled in the Bluetooth stack configuration, the device will automatically enter EM1 or EM2 mode when no events are being received from the Bluetooth stack. Using `gecko_wait_event()`is the easiest way to make sure the device is in the lowest power sleep mode whenever possible.

The Bluetooth stack's event handling is discussed in detail in section 5. Bluetooth Stack Event Handling.

`struct gecko_cmd_packet* gecko_peek_event(void)`

This is a non-blocking function to request Bluetooth events from the Bluetooth stack. When an event is requested and the event queue is not empty, a pointer to the `gecko_cmd_packet` struct is returned. If there are no events in the event queue, NULL is returned.

When using this non-blocking event listener, the EM sleep modes have to be managed by the application code as they are not managed automatically by the Bluetooth stack. The sleep mode management is done with `gecko_can_sleep_ms()` and `gecko_sleep_for_ms()` functions, which are discussed later.

The stack's event handling is discussed in detail in section 5. Bluetooth Stack Event Handling.

`int gecko_event_pending(void)`

This function checks to see if any Bluetooth stack events are pending in the event queue. If a pending Bluetooth event is found, the function returns a non-zero value to indicate that the event should be processed by either `gecko_peek_event()` or `gecko_wait_event()`. If no event is found, zero is returned.

`uint32 gecko_can_sleep_ms(void)`

This function is used to determine how long the Bluetooth stack can sleep. The return value is the number of milliseconds the stack can sleep until the next Bluetooth operation must occur. If sleeping is not possible, zero is returned. This function is only to be used with non-blocking `gecko_peek_event()` event handling.

`uint32 gecko_sleep_for_ms(uint32 max)`

This function is used to put the stack into EM sleep for a maximum number of milliseconds, set in the function's single parameter. The return value is the number of milliseconds actually slept. It is possible that the stack will awaken due to an external event. This function is only to be used with non-blocking `gecko_peek_event()` event handling.

native_gecko.h

This file is used in applications without RTOS. It provides IPC (interprocess communications) to the Bluetooth stack using direct function calls.

ncp_gecko.h

This file is to be used in applications providing the NCP functionality for the host. It provides IPC to Bluetooth stack using NCP headers as function calls.

host_gecko.h and gecko_bglib.h

These files are used when developing applications for an external host. `host_gecko.h` has the API definitions and `gecko_bglib.h` contains the adaptation layer between the host application and the BGAPI serial protocol.

rtos_gecko.h

`rtos_gecko.h` is used when the application is built for Micrium OS. The Bluetooth stack is a separate task for Micrium OS and uses its power, sleep, and memory management. `rtos_gecko.h` provides a wrapper for the IPC for using the Bluetooth stack from any task in Micrium OS. This file contains the Bluetooth stack API and the commands and events for the stack, and a configuration API for the Bluetooth stack.

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. With the Silicon Labs Bluetooth stack, the GATT definitions are either directly edited in the Visual GATT Editor GUI in Simplicity Studio or are written in XML and passed to the BGBuild executable as a pre-build task. 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 defines the GATT database structure and content, and is auto-generated by BGBuild.exe or by the Visual GATT Editor. 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.

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 thr 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 (see the Gecko Bootloader API reference in <Simplicity Studio Gecko SDK>\platform\bootloader\documentation\Gecko_Bootloader_API_Reference\index.html). 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 updated by changing the defines:

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

// Capabilities of this application (uint32_t)
#define BG_APP_PROPERTIES_CAPABILITIES

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

When using the OTA process with Bluetooth AppLoader, the application properties struct needs to reside immediately after the application vector table. This is enabled automatically when using linker files provided by the Bluetooth stack.

3.4 RTOS Support

The Bluetooth stack can also run on Micrium RTOS. In this case *native_gecko.h* is replaced by *rtos_gecko.h* and the following files are added to the project: *rtos_bluetooth.c* and *rtos_bluetooth.h*.

rtos_bluetooth.c and rtos_bluetooth.h

`rtos_bluetooth.c` and `rtos_bluetooth.h` provide the Micrium OS tasks for the IPC (Inter-Process Communication) with the Bluetooth stack and other Micrium OS tasks. The `rtos_gecko.h` header file, described below, also must be included when using Micrium OS. It provides the API IPC encapsulation for using the Bluetooth stack from any Micrium OS task.

Support for RTOS needs to be configured for the Bluetooth Stack in the `gecko_configuration_t` struct. The `config_flags` field needs to have `GECKO_CONFIG_FLAG_RTOS` set. This causes the Bluetooth stack to rely on the Micrium OS for sleeping, rather than sleeping directly. `scheduler_callback` and `stack_schedule_callback` must be configured to call proper functions. These callbacks are used to wake up the corresponding tasks.

The Bluetooth Stack configuration for use with Micrium OS is as follows:

```
.config_flags = GECKO_CONFIG_FLAG_RTOS,
.scheduler_callback = BluetoothLLCallback,
.stack_schedule_callback = BluetoothUpdate,
```

After calling `gecko_stack_init()` the `bluetooth_start_task()` can be called.

```
void bluetooth_start_task(OS_PRIO ll_priority, OS_PRIO stack_priority);
```

It takes task priorities as parameters. `ll_priority` is for Link Layer and `stack_priority` is for the Bluetooth stack. Link Layer Priority must be the highest priority in the system, as its timely execution is critical for the system's performance.

3.5 Multiprotocol Support

When the Bluetooth Stack is used in a multiprotocol environment, multiprotocol features in the Bluetooth stack must be enabled with following function:

```
gecko_init_multiprotocol(const void *config);
```

The `config` parameter is currently always set to NULL, it is reserved for future extensions.

Using Bluetooth in a multiprotocol environment also requires using the RAIL library with multiprotocol support.

3.6 Hardware Support

The following files are part of the Gecko SDK, they add support for hardware specific features.

hal-config.h

This header file contains the MCU peripheral initialization settings, such as those for clocks and power management and for peripherals such as UART, SPI, and so on. Note that this file contains only the non-board-specific settings of the peripherals, like the baud rate of the UART, and not the board-specific settings like input/output pins for UART.

init_mcu.c and init_mcu.h

These files include the device initialization function, which initializes internal settings of the MCU like clocks and power management.

init_board.c and init_board.h

These files include the board initialization function, which initializes external parts on the board. For example, it enables GPIOs, and initializes external flash on the radio board.

init_app.c and init_app.h

These files include the app initialization function, which initializes external parts on the WSTK according to the application. For example, it enables VCOM, sensors, and LCD display on the WSTK.

pti.c and pti.h

These files include the PTI initialization function, which enables the Packet Trace Interface.

hal-config-board.h

This header file contains the board initialization settings such as button and LED pins, UART and SPI pins, and so on. When the application is being developed for Silicon Labs' radio boards, these settings are set correctly in the examples provided in the SDK, but a developer who is creating applications for a custom hardware design needs to configure the settings accordingly.

bspconfig.h / bsphalconfig.h

The BSP (Board Support Package) header includes radio board-specific configurations, which are used as parameters for WSTK-specific functions like toggling IOs on the WSTK or driving the LCD display on the starter kit. If the Hardware Configurator tool is used, then the examples use `bsphalconfig.h`. Otherwise `bspconfig.h` is used.

mx25flash_spi.h

This header file includes functions to configure the SPI flash chip on some of the radio boards (for example BRD4100A) to low-power mode. This is useful when making sleep current measurements, for example, because if the SPI flash is not in low-power mode, the lowest EM2, EM3, or EM4 currents are not reached.

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 have to be properly configured. Once the hardware is initialized the stack also has to be initialized using the `gecko_init()` function.

4.1 Wireless Gecko MCU and Peripherals Configuration

initMcu()

The `initMCU()` function is used to initialize MCU core. This function starts the oscillators and configures energy modes of the device. Peripheral initializations that are independent of the board settings (for example, timer init) can be added to this function. The function must be called at the beginning of `main()`.

initBoard()

The `initBoard()` function is used to initialize board features, such as initializing the external flash. Peripheral initializations that depend on the board design (for example GPIO init or UART init) can be added to this function. The function must be called after `initMcu()`.

initApp()

The `initApp()` function is used to initialize application-specific features, such as enabling the SPI display on the WSTK. The function must be called after `initBoard()`.

4.1.1 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 following initialization function must be called:

```
void gecko_init_afh();
```

In a master-slave connection, both ends can use AFH independent of each other. The master may be non-adaptive, but the slave still may need to be adaptive. The standard allows using control transfer on a blocked channel. For compliance reasons if the slave 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.

4.1.2 Bluetooth Clocks

The clock settings are initialized in the `initMcu_clocks()` function. 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

To allow a device to stop HFCLK and enter sleep modes another clock is needed, which is the LFCLK clock.

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 has 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 or LFRCO, depending on the accuracy requirements.

The accuracy of the clock is defined in the Bluetooth stack configuration structure, see 4.2.5 Bluetooth Stack Configuration.

The default configuration is that the LFXO is connected to the Wireless Gecko and set as the low frequency clock (LFCLK). If the design doesn't have the LFXO connected then sleep has to be explicitly disabled from the application and LFRCO has to be set to be used as clock source. As mentioned in section 4.2.4 Sleep, if the LFCLK is not accurate enough, then the sleep modes have to be disabled for the Bluetooth stack to operate correctly.

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 with the `hfxoInit.ctuneSteadyState` and the `lfxoInit.ctune` settings in the `initMcu_clocks()` function.

```
// Initialize HFXO
CMU_HFXOInit_TypeDef hfxoInit = BSP_CLK_HFXO_INIT;
hfxoInit.ctuneStartup = BSP_CLK_HFXO_CTUNE;
hfxoInit.ctuneSteadyState = BSP_CLK_HFXO_CTUNE;
CMU_HFXOInit(&hfxoInit);
```

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 `cmd_flash_ps_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 for EFR32xG1x devices, or the last flash page for EFR32xG21 devices. Please 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.

Note: The Bluetooth stack only supports 38.4 MHz HFXO frequency; no other HFXO frequencies are supported.

4.1.3 DC-DC Configuration

On devices that have DC-DC, the configuration is set in the `initMCU()` function. 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 *hal-config-board.h*.

```
#define BSP_DCDC_INIT                                                   \
  {                                                                     \
    emuPowerConfig_DcdcToDvdd,   /* DCDC to DVDD */                     \
    emuDcdcMode_LowNoise,        /* Low-noise mode in EM0 */            \
    1800,                        /* Nominal output voltage for DVDD mode, 1.8V  */  \
    15,                          /* Nominal EM0/1 load current of less than 15mA */ \
    10,                          /* Nominal EM2/3/4 load current less than 10uA  */ \
    200,                         /* Maximum average current of 200mA
                                    (assume strong battery or other power source) */      \
    emuDcdcAnaPeripheralPower_DCDC,/* Select DCDC as analog power supply (lower power) */ \
    160,                         /* Maximum reverse current of 160mA */                   \
    emuDcdcLnCompCtrl_1u0F,      /* 1uF DCDC capacitor */                                 \
  }
```

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

4.1.4 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 *hal-config-board.h* using the prefix `BSP_FEM_`.

FEM is initialized in `initFem()` within the `initBoard()` function if the board supports FEM.

4.1.5 Periodic Advertising

Periodic advertising enables multiple listeners to be synchronized with a single advertising device. Thus it is a form of multicast.

Each listener needs to be synchronized to the advertising device before they start receiving data. Periodic advertising uses a scanner on the listening device to establish a synchronization to the advertising device. After synchronization the scanner can then be stopped. This makes it much more power-efficient than using the scanner full time for listening for broadcast advertisements.

The periodic advertising consists of two components: periodic advertiser role and periodic advertising synchronization on listening side.These two components are independent of each other and needs to be initialized separately.

Periodic Advertiser

`max_advertisers` in the Bluetooth configuration also configures the maximum number of periodic advertisers.

To enable Periodic Advertiser in the Bluetooth stack, the following initialization function must be called after the generic `gecko_init` function:

```
void gecko_init_periodic_advertising();
```

Periodic Advertising Synchronization

`max_periodic_sync` in the Bluetooth config is used to configure the maximum number of synchronizations the Bluetooth stack needs to support.

To enable Periodic Advertising Synchronization in the Bluetooth stack, the following initialization function must be called after the generic `gecko_init` function:

```
void gecko_bgapi_class_sync_init();
```

This command also initializes the BGAPI sync class, making it available to use.

4.1.6 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 `configEnablePti()` within the `initApp()` function. The baudrate can be set in *hal-config.h* using the HAL_PTI_BAUD_RATE definition, while pins can be configured in *hal-config-board.h* using the definitions with the BSP_PTI_ prefix.

Starting with Bluetooth 2.6.x PTI is configured with functions provided by RAIL.

4.1.7 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.

4.1.8 Whitelisting

Whitelisting 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 whitelist.

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

Bonded devices are added to the whitelist automatically. Alternatively they can be added manually with the BGAPI command `gecko_cmd_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 whitelisting feature will effectively block these devices during device discovery.

To enable whitelisting in the Bluetooth stack, the following initialization function must be called after the generic `gecko_init` function:

```
void gecko_init_whitelisting();
```

When the function is enabled, it can be enabled and disabled at runtime by the BGAPI command `gecko_cmd_le_gap_enable_whitelisting()`.

4.1.9 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. COEX is configured in *hal-config-board.h* using defines with prefixes `BSP_COEX_` and `HAL_COEX_`.

To enable COEX, call the following function after `gecko_stack_init()`.

```
gecko_initCoexHAL();
```

COEX implements the GPIO interface to the Wi-Fi IC. It depends on EMLIB `em_gpio.c` and EMDRV `gpiointerrupt.c` and requires both files to be included in the project.

4.2 Bluetooth Configuration with gecko_stack_init()

The `gecko_stack_init()` function is used to configure the Bluetooth stack, including sleep mode configuration, memory allocated for connections, OTA configuration, and so on. None of the Bluetooth stack functions can be used before the Bluetooth stack has been configured.

Bluetooth stack configuration example:

```
uint8_t bluetooth_stack_heap[DEFAULT_BLUETOOTH_HEAP(MAX_CONNECTIONS)];
static const gecko_configuration_t config = {
  .config_flags=0,
  .sleep.flags=SLEEP_FLAGS_DEEP_SLEEP_ENABLE,
  .bluetooth.max_connections=MAX_CONNECTIONS,
  .bluetooth.heap=bluetooth_stack_heap,
  .bluetooth.heap_size=sizeof(bluetooth_stack_heap),
  .bluetooth.sleep_clock_accuracy = 100, // ppm
  .gattdb=&bg_gattdb_data,
  .ota.flags=0,
  .ota.device_name_len=3,
  .ota.device_name_ptr="OTA",
  .max_timers=4
};
```

Configuration options in the `gecko_stack_init()` function are: sleep enable/disable, Bluetooth connection count, heap size, sleep clock accuracy, GATT database, OTA configuration, and PA configuration.

Once the function `gecko_stack_init()` is called, each stack component used has to be initialized separately. This separation allows memory optimization by not including unnecessary stack components.

The following APIs can be used to initialize stack components separately:

`gecko_bgapi_class_dfu_init()`	enables device firmware upgrade (dfu) APIs.
`gecko_bgapi_class_system_init()`	enables local device (system) APIs.
`gecko_bgapi_class_le_gap_init()`	enables Generic Access Profile (gap) APIs.
`gecko_bgapi_class_le_connection_init()`	allows managing connection establishment, parameter setting, and disconnection procedures via the connection APIs.
`gecko_bgapi_class_gatt_init()`	enables the ability to browse and manage attributes in a remote GATT server via the gatt APIs.
`gecko_bgapi_class_gatt_server_init()`	enables the ability to browse and manage attributes in a local GATT database gatt_server APIs.
`gecko_bgapi_class_hardware_init()`	enables access and configuration of the software timers.
`gecko_bgapi_class_flash_init()`	enables persistent store commands (flash) APIs that can be used to manage the user data in the flash memory.
`gecko_bgapi_class_test_init()`	enables the DTM test APIs.
`gecko_bgapi_class_sm_init()`	enables the security manager (sm) APIs.
`gecko_bgapi_class_util_init()`	enables utility function APIs like *atoi* and *itoa*.

4.2.1 CONFIG_FLAGS

Current flags:

`GECKO_CONFIG_FLAG_RTOS`	1 = Application uses RTOS. Stack does not configure clocks, vectors, TEMPDRV, or sleeps as they are provided by RTOS.

4.2.2 Mbedtls

The Mbedtls cryptography library used by the stack is configured using a configuration file that defines what algorithms are supported and if the implementation uses hardware acceleration or is done on software. The Mbedtls configuration file path is given using `#define MBEDTLS_CONFIG_FILE`. The default configuration file `mbedtls_config.h` is in the SDK and should be used as a template if the configuration needs to be changed.

4.2.3 Multiprotocol Priority Configuration

When the Bluetooth stack is used with other protocols in a multiprotocol environment, it may become necessarily 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:

```
gecko_bluetooth_ll_priorities  custom_priorities;
static const gecko_configuration_t config = {
  //
  .bluetooth.linklayer_priorities = &custom_priorities,
  //
};
```

The `gecko_bluetooth_ll_priorities` struct must be initialized to default state by the `GECKO_BLUETOOTH_PRIORITIES_DEFAULT` constant.

The `gecko_bluetooth_ll_priorities` structure contain following fields:

- `scan_min & scan_max` - The priority range for scan operation.
- `adv_min & adv_max` - 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.
- `threshold_coex` - The threshold level when to raise priority signal, only used if coex is enabled.
- `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
```

4.2.4 Sleep

Wireless Gecko's sleep mode EM2 (energy mode two) must be enabled in the `gecko_init()` function. The sleep flags are part of the `gecko_configuration_t` struct. The `SLEEP_FLAGS_DEEP_SLEEP_ENABLED` flag must be set to enable the sleep. Sleep modes are handled automatically by the stack in the case of blocking events, as described in section 5. Bluetooth Stack Event Handling.

Example of enabling sleep in the `gecko_configuration_t` struct (main.c):

```
.sleep.flags = SLEEP_FLAGS_DEEP_SLEEP_ENABLE    // EM sleeps enabled
```

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, then low power sleep modes cannot be entered. For applications where low power sleep modes are not needed, the LFXO can be left out, but then the sleep flag in the gecko configuration struct must be set like this:

```
  .sleep.flags = 0,                                  // Sleeps disabled
```

Disabling Sleep at Runtime

If sleep needs to be disabled at runtime, it can be done by calling the Sleep driver function `SLEEP_SleepBlockBegin(sleepEM2)`. To re-enable EM2 Deep Sleep mode use `SLEEP_SleepBlockEnd(sleepEM2)`. While EM2 is disabled (/blocked), the stack will switch between EM0 and EM1. For more information, refer to knowledge base article Using Energy Modes with Bluetooth Stack.

4.2.5 Bluetooth Stack Configuration

Stack Memory

The Bluetooth stack uses internal memory management to allocate memory for each connection and for internal data buffers. This memory needs to be allocated and passed to the Bluetooth stack by the application. Memory size depends on the number of connections. The C macro `DEFAULT_BLUETOOTH_HEAP()` calculates the default size in bytes of needed memory.

Example of allocating `bluetooth_stack_heap` array and passing it to the Bluetooth stack:

```
uint8_t bluetooth_stack_heap[DEFAULT_BLUETOOTH_HEAP(MAX_CONNECTIONS)];
static const gecko_configuration_t config = {
  //
  .bluetooth.heap = bluetooth_stack_heap,
  .bluetooth.heap_size = sizeof(bluetooth_stack_heap),
  //
};
```

Number of Connections

The absolute maximum number of simultaneous Bluetooth connections is 8. The amount of memory that is allocated for connection management further limits the number of connections. The memory is allocated during initialization in `gecko_init()`. C-define `MAX_CONNECTIONS` can be defined to set the number of connections. The same define is then also used to calculate the memory size of the Bluetooth stack as explained above. `MAX_CONNECTIONS` is then further passed to Bluetooth stack in the `.bluetooth.max_connections` field in the configuration struct.

Example of limiting the Bluetooth connections to one (1).

```
#define MAX_CONNECTIONS 1
```

For more information about connection RAM usage, refer to 7.3.2 Bluetooth Connection Pool.

Sleep Clock Accuracy

The Bluetooth stack uses `.sleep_clock_accuracy` to optimize wake-up time from sleep. The unit is in ppm (Parts Per Million). If this value is too large, then the Bluetooth stack wakes up from sleep too early to wait for the actual event, which causes excessive power usage. If this value is too small, then the Bluetooth stack wakes up too late and misses the connection event, which causes dropped connections.

If this is not defined or is set to 0, then the default value of 250 ppm is used.

Example of setting sleep clock accuracy.

```
.bluetooth.sleep_clock_accuracy = 100, // ppm
```

Advertisers

The maximum number of advertisement sets can be defined by this configuration option. These sets can be used to start multiple advertisers. This configuration option also configures the maximum number of periodic advertisements. Each advertisement context allocates ~60 bytes of RAM.

```
.bluetooth.max_advertisers = 5;  //!< Maximum number of advertisers to support, if 0 defaults to 1
```

Note: Maximum connectable advertisements are limited by `MAX_CONNECTIONS`.

Synchronous Advertisements

The maximum number of supported synchronous advertisements needs to be configured. Each context allocates ~40 bytes of RAM.

```
.bluetooth.max_periodic_sync = 5;  //!< Maximum number of synhronous advertisers to support. Default is 0,
none supported
```

4.2.6 OTA Configuration

Bluetooth Over-the-Air (OTA) firmware upgrades are supported, since part of the firmware upgrade is handled by the Bluetooth AppLoader application.

The OTA mode uses the `.ota.flags` configuration field. It currently has a single option, `GECKO_OTA_FLAGS_RANDOM_ADDRESS`, which sets OTA to use a static random adress, instead of a public address.

When the Wireless Gecko is in AppLoader's OTA mode, its device name and the device name length can be configured through the gecko configuration struct.

```
  .ota.device_name_len = 3,    // OTA name length
  .ota.device_name_ptr = "OTA",   // OTA Device Name
```

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 Booloader User's Guide* and *AN1086: Using the Gecko Bootloader with Silicon Labs Bluetooth Applications*.

4.2.7 PA

On EFR32 SoC-based designs, the PAVDD (Power Amplifier voltage regulator VDD input) can be supplied from the output of the DC/DC or straight from a 3.3 V power supply.

The Bluetooth stack configuration defaults to using DC/DC as the PAVDD input. If PAVDD is being supplied from an 3.3 V power supply then the `.pa.input` field needs to be defined.

The Bluetooth stack automatically selects the high power PA if available. Setting 1 in `pa_mode` configuration will configure the Bluetooth stack to always use low power PA.

```
 .pa.config_enable = 1,      // PA Configuration is enabled
 .pa.input = GECKO_RADIO_PA_INPUT_VBAT, // PAVDD is upplied from an 3.3 V power supply
 .pa.pa_mode=0                          // selects high power PA if available
```

4.2.8 Software Timers

Maximum available software timers can be configured. Each timer needs resources from the stack to be implemented. Increasing amount of soft timers may cause degraded performance in some use cases.

```
.max_timers = 4;  // Maximum number of soft timers, up to 16,  Default: 4
```

4.2.9 RF Path Gain

The application can define RF path gain values for RX and TX separately.

The Bluetooth stack takes TX RF path gain into account when adjusting transmitter power. Power radiated from the antenna then matches the application request. For example, if maximum power requested by the application is at +10 dBm and path loss is -1 dBm, then actual power at the pin is +11 dBm.

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 dBm
.rf.rx_gain = -18; // RF RX path gain in unit of 0.1 dBm
```

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.

5.1 Blocking Event Listener

`gecko_wait_event()` is a implementation of a blocking wait function, which waits for events to emerge to the event queue and returns them to the event handler. This is the recommended mode of operation with the Bluetooth stack, because it manages the sleep most efficiently and automatically, while keeping the device and connections in sync.

- The `gecko_wait_event()` function processes the internal message queue until an event is received.
- If there are no pending events or messages to process, the device goes to EM1 or EM2 sleep mode.
- The function returns a pointer to a `gecko_cmd_packet` structure holding the received event.

The code snippet below shows the simple main while loop from the iBeacon example using `gecko_wait_event()`, which sets up advertising after boot.

```
/* Main loop */
while (1) {
    struct gecko_cmd_packet* evt;

    /* Wait (blocking) for a Bluetooth stack event. */
    evt = gecko_wait_event();

    /* Run application and event handler. */
    switch (BGLIB_MSG_ID(evt->header))
    {
    /*  This boot event is generated when the system is turned on or reset. */
    case gecko_evt_system_boot_id:

      /* Initialize iBeacon ADV data */
      bcnSetupAdvBeaconing();
      break;

    /* Ignore other events  */
    default:
      break;
    }
```

5.2 Non-Blocking Event Listener

This mode of operation requires more manual adjustment, for example sleep management needs to be done by the application. In some use cases non-blocking operation is required.

- The `gecko_peek_event()` function processes the internal message queue until an event is received or all of the messages are processed.
- The function returns a pointer to a `gecko_cmd_packet` structure holding the received event, or NULL if there are no events in the queue.

5.2.1 Sleep and Non-Blocking Event Listener

When an application uses the non-blocking `gecko_peek_event()` function to create an event handler, the sleep implementation differs as well. The application must use `gecko_can_sleep_ms()` to query the stack for how long the device can sleep, and then use the `gecko_sleep_for_ms()` function to set it to sleep for that time. Interrupts must be disabled before calling `gecko_can_sleep_ms()` or `gecko_sleep_for_ms()` functions, and enabled once the functions have been executed.

Note: A recommendation is that no extra functionality is added to this critical section. It will add latency to interrupts and degrade performance.

The example below shows how to implement sleep management when non-blocking event handling is used.

```
/* Main loop */
while (1) {
    struct gecko_cmd_packet* evt;
    CORE_DECLARE_IRQ_STATE;

    /* Poll (non-blocking) for a Bluetooth stack event. */
    evt = gecko_peek_event();

    /* Run application and event handler. */
    switch (BGLIB_MSG_ID(evt->header))
    {
    /* This boot event is generated when the system is turned on or reset. */
    case gecko_evt_system_boot_id:

      /* Initialize iBeacon ADV data */
      bcnSetupAdvBeaconing();
      break;

    /* Ignore other events */
    default:
      break;
    }
     CORE_ENTER_ATOMIC();                    // Disable interrupts

      /* Check how long the stack can sleep */
      uint32_t durationMs = gecko_can_sleep_ms();
      /* Go to sleep. Sleeping will be avoided if there isn't enough time to sleep */
      gecko_sleep_for_ms(durationMs);

    CORE_EXIT_ATOMIC();                     // Enable interrupts

}
```

5.2.2 Notification for Updating Event Listener

In some cases, there may be a need for running the Bluetooth event loop inside another event loop in the application. The Bluetooth stack has a callback mechanism for notifying the application about the demand for updating the Bluetooth stack event listener. This is enabled by defining a callback function in the Bluetooth configuration struct.

Note: This `stack_schedule_callback` is called from the interrupt context. It is important NOT to call `gecko_peek_event` or `gecko_wait_event` from this context. The application must set a flag or use another mechanism for enabling the application main loop to update the Bluetooth stack.

```
static const gecko_configuration_t config = {
  //
  .stack_schedule_callback = bluetooth_update
  //
};

void bluetooth_update()
{
  //set notification for application
}
```

5.3 Event Listener with Micrium OS

The application uses a different procedure to receive an event with Micrium OS. Instead of calling the function to receive events, the application needs to pend for a Micrium OS flag. Events can be only received from a single task.

Micrium OS flags in `bluetooth_event_flags` are used to notify different tasks about the state of the Bluetooth stack. The application only uses `BLUETOOTH_EVENT_FLAG_EVT_WAITING` and `BLUETOOTH_EVENT_FLAG_EVT_HANDLED`.

The application event handler needs to pend for `BLUETOOTH_EVENT_FLAG_EVT_WAITING`.

```
OSFlagPend(&bluetooth_event_flags, (OS_FLAGS)BLUETOOTH_EVENT_FLAG_EVT_WAITING        0,OS_OPT_PEND_BLOCKING +
OS_OPT_PEND_FLAG_CONSUME, NULL,&os_err);
```

The incoming event is then available in `bluetooth_evt`.

```
switch (BGLIB_MSG_ID(bluetooth_evt->header)) {
      ...
}
```

After the event is handled, it needs to be freed to allow the next event to be received. This is done by notifying the Bluetooth task by posting the flag `BLUETOOTH_EVENT_FLAG_EVT_HANDLED`.

```
OSFlagPost(&bluetooth_event_flags, (OS_FLAGS)BLUETOOTH_EVENT_FLAG_EVT_HANDLED, OS_OPT_POST_FLAG_SET, &os_err);
```

Note: When the application is pending, sleep and power management are automatically handled by Micrium OS rather than by the application.

5.3.1 Commands from Multiple Tasks

It is possible to send Bluetooth commands from multiple Micrium OS tasks. It requires that each task acquires exclusivity before sending the commands and releases it afterward.

The Bluetooth stack provides two functions for convenience. `BluetoothPend` acquires the Micrium OS mutex and `BluetoothPost` releases the mutex.

The following code block acquires the mutex for Bluetooth before the Bluetooth command and releases it afterward.

```
BluetoothPend(&err); //acquire mutex for Bluetooth stack
gecko_cmd_gatt_server_send_characteristic_notification(0xff, gattdb_temp_measurement, 5, temp_buffer);
BluetoothPost(&err);//release mutex
```

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 `gecko_external_signal()` function can be called from interrupt context.
- Interrupts must be disabled before calling `gecko_sleep_for_ms(...)` as shown in the previous code example.

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 `system_external_signal` can be generated by calling the `void gecko_external_signal(uint32 signals)` function.
- The function `gecko_external_signal` can be called from the interrupt context.
- The `signals` parameter of the `gecko_external_signal` function is passed to the `system_external_signal` event.

```
/**
* Main
*/
void main()
{
  ...

      //Event loop
  while(1)
  {
    ...

      //External signal indication (comes from the interrupt handler)
    case gecko_evt_system_external_signal_id:
      // Handle GPIO IRQ and do something
      // External signal command's parameter can be accessed using
     // event->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
  gecko_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. Default interrupt priorities for radio is 1, for Link Layer the priority is 2, USART interrupts are 3, and other interrupts have default priority of 4.

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.

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*	< ~20 ms	If the link control procedure is not handled in time, Bluetooth link loss may happen. Slave-side channel map update and connection update timings are controlled by master.
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.

*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	Used for sleep timings. Both channels are always reserved. The application can only read the RTC value, but cannot write it or use RTCC. RTCC is only reserved in EFR32BG1 and EFR32BG12. For more information see 7.4 RTCC.
	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 application, but MSC must not be disabled.
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.

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, 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 it 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.
- 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.
- 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.
- User data page is used for storing manufacturing tokens. On EFR32BG2X devices it is located at end of main flash.

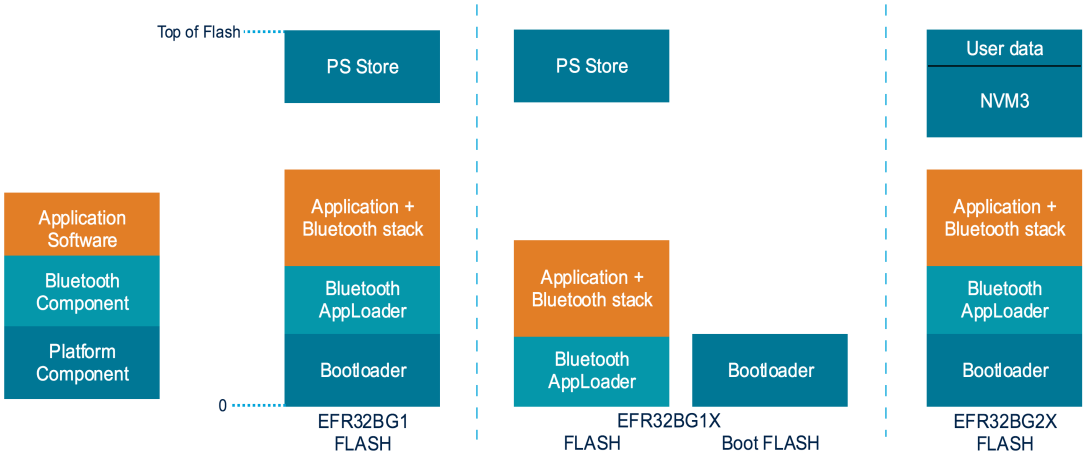


Figure 7.1. Flash Usage With and Without Separate Bootloader Flash

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

	Compiler	EFR32BG1	EFR32BG12	EFR32BG13	EFR32BG21
Bootloader		16	16	16	16
Bluetooth AppLoader		38	42	42	48
soc-empty*	GCC	120	128	129	136
	IAR	120	128	129	136
soc-thermometer*	GCC	124	130	131	136
	IAR	123	130	131	136
NVM		4	4	4	24

**soc-empty* and *soc-thermometer* are example applications provided in the Bluetooth SDK. They are compiled with high size optimizations. GCC uses the `-Os` flag, and IAR the `-Ohz` flag.

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

`gecko_init()` function automatically initializes each stack component. For more selective initialization, **`gecko_stack_init()`** must be used. Then each required stack component is invidually initialized. For more information, see section 4.2 Bluetooth Configuration with gecko_stack_init().

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 Bluetooth stack.

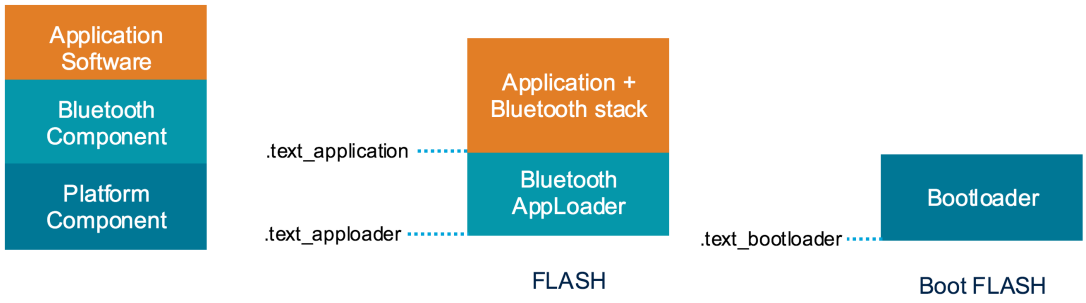


Figure 7.2. Sections Defined in the Linker File and Their Placement

The linker file defines two memory areas, one for main flash and one for bootloader flash. If no separate bootloader flash exists, the linker file reserves some memory from main flash for the bootloader. 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 connection pool
- Bluetooth GATT database
- C STACK
- C HEAP

The following table shows the details of RAM usage.

Component	Allocated RAM
Bluetooth stack	12 kB
Bluetooth connection pool	4824 + Number of connections * 480 bytes
Bluetooth GATT database	Application-dependent (20 to 200 bytes)
Call stack	2 kB
Heap memory	3 kB

7.3.1 Bluetooth Stack

The Bluetooth stack requires at least 12 kB RAM. It includes Bluetooth stack software with low-level radio drivers and the application programming interface.

7.3.2 Bluetooth Connection Pool

The Bluetooth stack uses its own static memory pool for dynamic memory allocation. The size of the allocated memory pool depends on the number of parallel connections. The number is set with the `.bluetooth.max_connections` parameter in the `gecko_init()` function.

*Bluetooth Connection Pool Size = 4824 + Number of connections * 436 bytes*

7.3.3 Bluetooth GATT Database

The Bluetooth GATT database uses 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.

7.3.4 Call Stack

The Bluetooth stack requires at least a 1.5 kB call stack to be reserved from RAM. Application developers must allocate RAM for the application call stack on top of the 1.5 kB required by the stack.

Location of the call stack size definition depends on the compiler and startup files. The default call stack size is 2 kB. This can be overridden by the following command line options:

Compiler	Command line option	Note
IAR	`--config_def __STACK_SIZE=<size>`	Call stack is defined in the linker file. This parameter needs to be passed to the linker.
GCC	`-D __STACK_SIZE=<size>`	Call stack is defined in startup code. This needs to be defined for the compiler.

7.3.5 Heap memory

Heap memory must be reserved based on application requirements.

Location of heap size definition depends on compiler and startup files. Minimum heap size is 3328(0xD00) bytes, which is also the default value. This can be overridden by the following command line options:

Compiler	Command line option	Note
IAR	`--config_def __HEAP_SIZE=<size>`	Call stack is defined in the linker file. This parameter needs to be passed to the linker.
GCC	`-D__HEAP_SIZE=<size>`	Call stack is defined in startup code. This needs to be defined for the compiler.

7.4 RTCC

This chapter applies only to EFR32BG1 and EFR32BG12. Other devices have a separate timer for Bluetooth, and RTCC is freely usable by the application.

The hardware RTCC (Real Time Clock and Calendar) is set to run in counter mode by the Bluetooth stack and is reserved for the stack's use. The RTC value can, however, be read by the application, but it cannot be written by the application. The RTC value is reset every time the device boots up.

If the application requires RTCC-like functionality, the following application code can be developed:

1. Build a mechanism to retrieve the current time from an external device such as a smart phone. Some smart phones implement Bluetooth Time Profile and it can be used to read a time and date value.
2. Convert the time to “seconds since epoch” (for example using `mktime` from stdlib).
3. Use the Bluetooth stack’s API `hardware_get_time()` to get seconds elapsed since reset.
4. Calculate the difference between “seconds since epoch” and seconds since reset and store it for example to a PS-key.
5. When you want to get the current calendar time, use `hardware_get_time` to get the current RTC value, add the value from the PS-key to it, and then use `localtime` from stdlib to get current calendar time.

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 contins 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 IAR\ ARM\ -\ Default/soc-thermometer-iar-mg1p.out
```

objdump then gives the following output:

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text_apploader rw 00008fc0  00004000  00004000  00000034  2**11
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .text_application us 0001e3d3  0000d000  0000d000  00008ff4  2**11
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 A1 rw         00000800  20000000  20000000  000273c8  2**3
                  ALLOC
  3 P3 rw         00000246  20000800  20000800  000273c8  2**2
                  ALLOC, CODE
  4 P3 ui         00000d00  20000a48  20000a48  000273c8  2**3
                  ALLOC
  5 P3 zi         00002b60  20001748  20001748  000273c8  2**8
```

`.text_apploader` contains the Bluetooth AppLoader.

`.text_application` contains the application code and read-only data. Size of the application in this example is 0x1e3d3 in hexadecimal, and 123859 bytes in decimal.

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 GNU\ ARM\ v4.9.3\ -\ Default/soc-thermometer-gcc-mg1p.axf
```

objdump then gives the following output:

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text_bootloader 00000000  00000000  00000000  000306d0  2**0
                  CONTENTS
  1 .text_apploader 00009000  00004000  00004000  00004000  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .text_application 0001e4c4  0000d000  0000d000  0000d000  2**8
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .text_application_ARM.exidx 00000008  0002b4c4  0002b4c4  0002b4c4  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .stack_dummy  00000400  20000000  20000000  000306d0  2**3
                  CONTENTS
  5 .text_application_data 000002d0  20000400  0002b4cc  00030400  2**2
                  CONTENTS, ALLOC, LOAD, CODE
  6 .bss          00002a88  20000700  0002b800  00030700  2**8
                  ALLOC
  7 .heap         00000c00  20003188  20003188  00030ad0  2**3
                  CONTENTS
```

`.text_bootloader` contains the bootloader. In this example it is loaded separately, and the section is empty.

`.text_apploader` contains the Bluetooth AppLoader.

`.text_application` contains the application code and read-only data. The size of the application in this example is 0x1e3c3 in hexadecimal and 124100 bytes in decimal.

`.text_application_ARM.exidx` is used for debugging

`.stack_dummy` is a placeholder section for the call stack.

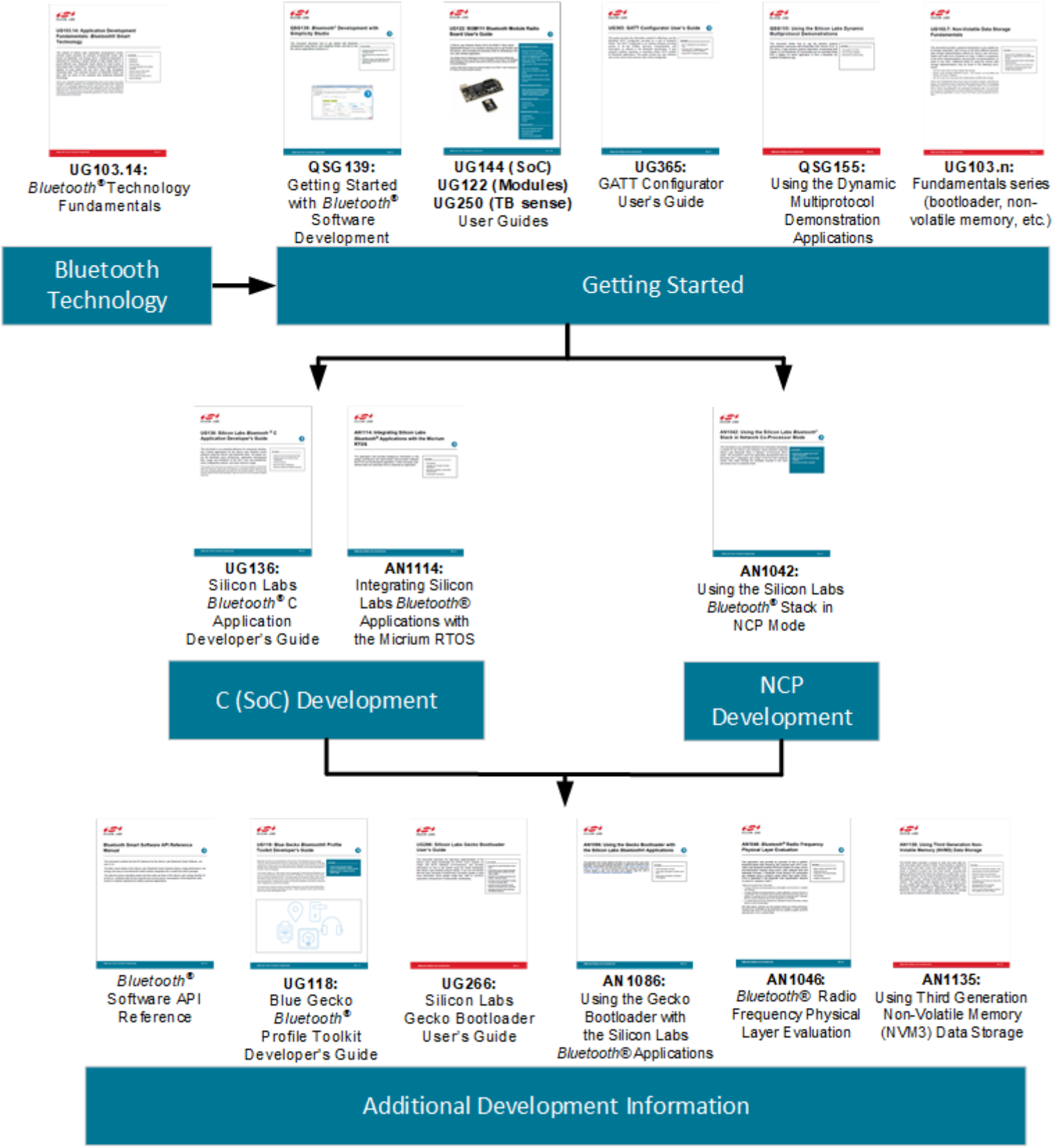
`.text_application_data` is the RAM section for initialized variables.

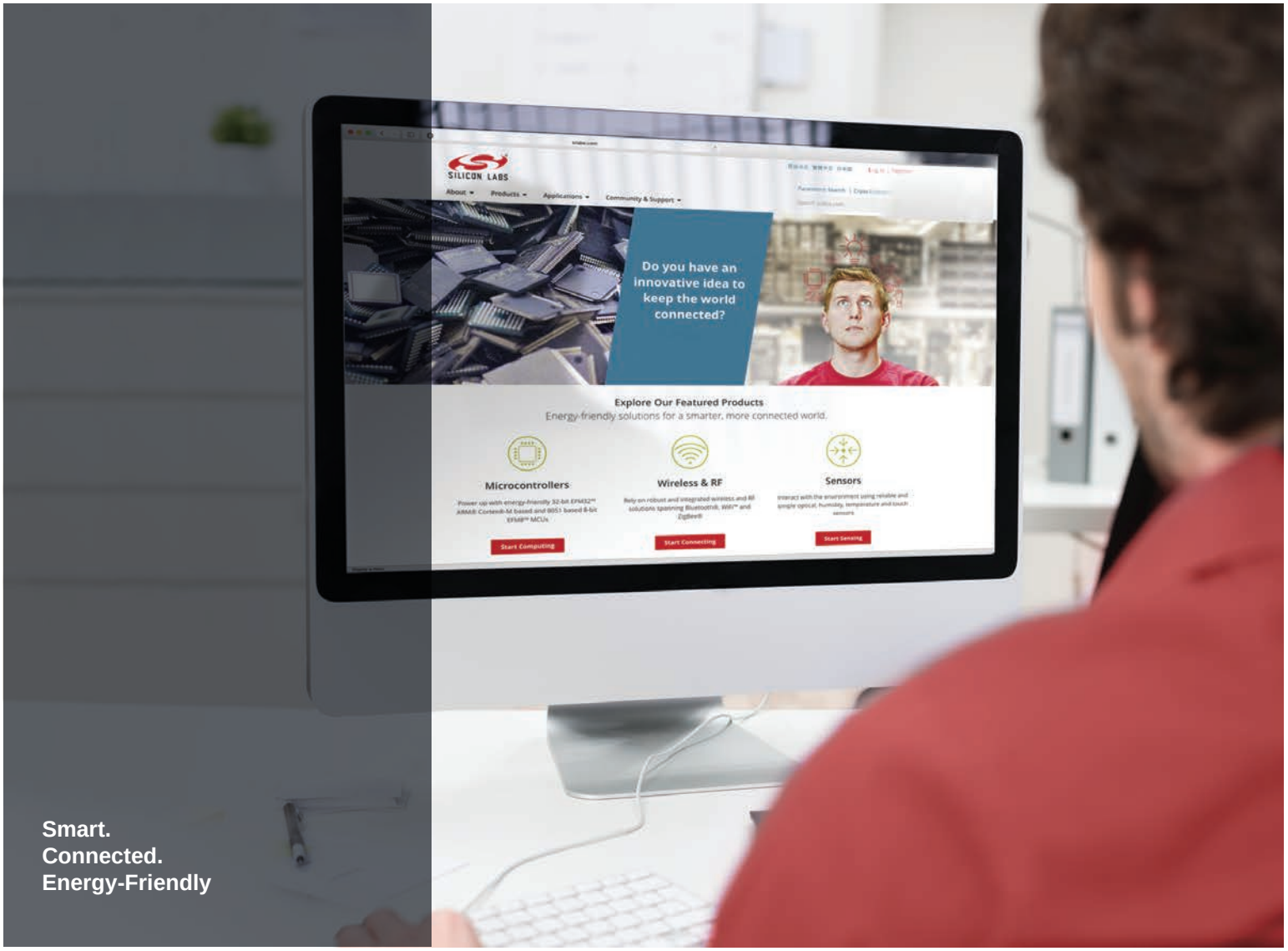
`.bss` is the RAM section for uninitialized variables.

`.heap` is the RAM section for heap.

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

9. Documentation





Products
www.silabs.com/products

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

SILICON LABS®

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

http://www.silabs.com