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 2.6.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 Prerequisites

This documents assumes the Silicon Labs' Bluetooth SDK 2.6.0 or newer 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: Application Development Fundamentals: Bluetooth Technology*.

For installation and tool introductions see *QSG108: Getting Started with Silicon Labs Bluetooth Software*. For instructions on getting started using example applications in Silicon Labs Simplicity Studio development environment, see *QSG139: Bluetooth Development with Simplicity Studio*.

Finally, Silicon Labs offers plugins for Bluetooth Developer Studio for quick code GATT database generation. See *QSG126: Bluetooth® Developer Studio Quick Start Guide* for a quick start guide to the plugins.

Currently supported compilers and IDE versions are shown in the following table.

SDK	IDE	Compiler
v2.6.x	Simplicity Studio 4.1.4 or newer	IAR v7.80.2 and GCC 4.9.3

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.

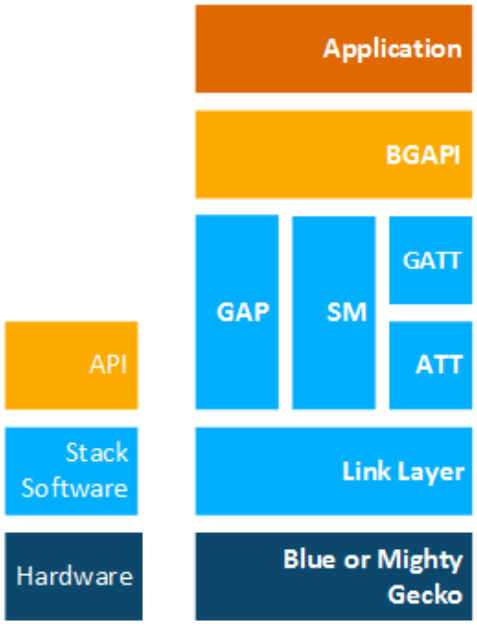


Figure 2.1. Bluetooth Stack Architecture Block Diagram

The Bluetooth stack contains following blocks.

- **Bootloader—**Three bootloaders are currently provided. Two legacy bootloaders are included in the stack:.
  - Legacy OTA (Over-the-Air) firmware updates
  - Legacy UART DFU (Device Firmware Update)

  The third, 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.
- **OTA supervisor—**An application that starts after the bootloader. It checks if the user application is valid and, if it is, the supervisor starts the application. If the application image is not valid the supervisor starts the configured bootloader to try to receive a valid application image. This requires using the legacy OTA bootloader or the Gecko Bootloader.

2.1 Application Build Flow

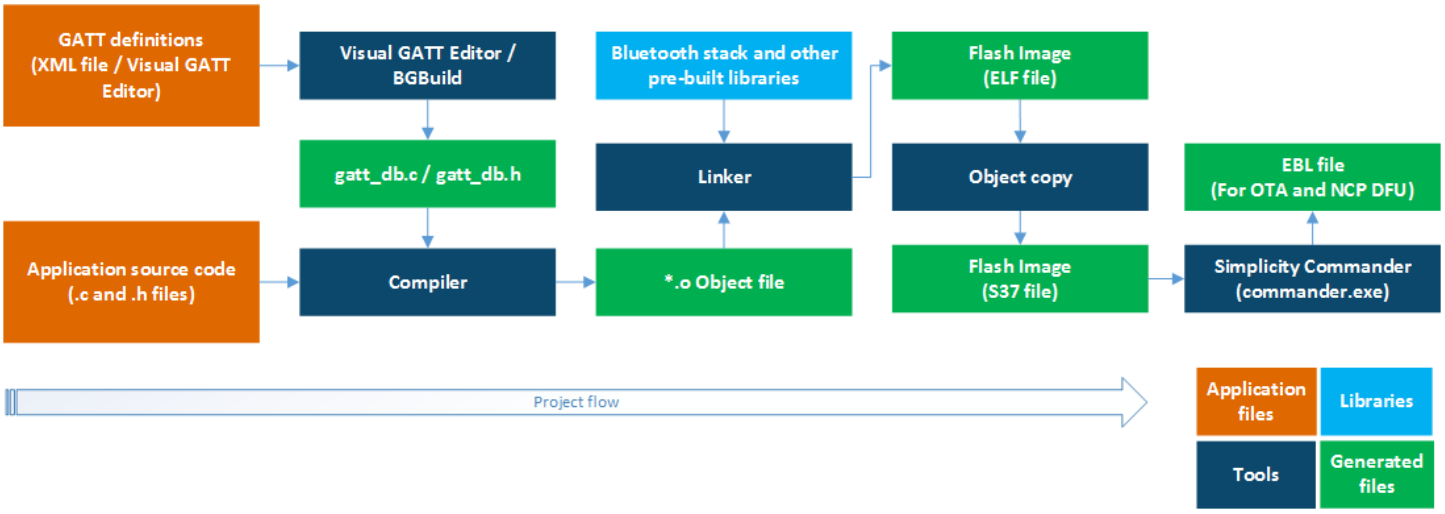


Figure 2.2. Bluetooth Project Build Flow

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 Protocol

This folder includes the Bluetooth stack-related files.

Bluetooth Stack and Bootloader Libraries

The Bluetooth stack libraries are:

- **binbootloader.o**: Bootloader image (pre-compiled legacy OTA bootloader). This file is needed for legacy bootloaders, which are only supported in EFR32xG1 devices. Starting from v2.3.0 SDK the unified Gecko Bootloader is available, which supports all the EFR32 devices. See *UG266: Silicon Labs Gecko Bootloader User Guide* for more details.
- **binstack.o:** Binary image of the Bluetooth stack, supervisor, and libraries used by the Bluetooth stack.
- **stack.a:** Exported symbols of Bluetooth stack, EMLIB and EMDRV that are shared with application.
- **bgapi.a:** Implements BGAPI-commands for non-core features. Beginning with SDK v2.3.0 *bgapi.a* must be included in all projects. For older SDKs it only needs to be included in NCP applications.
- **libcoex.a:** Implements Wi-Fi and Bluetooth coexistence arbitration features. Using this requires hardware support.

att.h and att_def.h

AAT (Application Address Table) is a support structure (meta data) for the firmware update file format EBL, defining the size and location of the application within the upgrade image. It is the first item in the application image and has pointers to the application vector table. The stack supervisor will initialize the application vector table and call the stack location from the vector table, before calling the application.

While ATT does not define the size and location of the application within GBL upgrade images, it is still required. Therefore, AAT is a mandatory part of all projects, and applications must include the `aat.h` header file.

bg_version.h

The stack version.

native_gecko.h

This file serves two purposes: first it contains the actual Bluetooth stack API and the commands and events for the stack, and second it provides a configuration, event, and sleep management API to the Bluetooth stack.

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

```
void 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 the next section. `gecko_init()` must be called by the application to initialize the Bluetooth stack.

Beginning with SDK v2.3.0 this function is deprecated but functional to maintain backward compatibility.

`void 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.

3.1.1 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.

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.

The configuration API is:

```
void 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 component used 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();`

3.2 Platform

This folder includes the device-related files.

EMLIB is a low-level peripheral support library that provides a unified API for all EFM32, EZR32, and EFR32 MCUs and SoCs from Silicon Laboratories. EMDRV is a set of function-specific high-performance drivers for EFR32 on-chip peripherals. Drivers are typically DMA-based and use all available low-energy features. For most drivers, the API offers both synchronous and asynchronous functions.

The developer can choose which peripheral drivers to include in the project. The required files of the EMDRV source code can be included in the project, bearing in mind not to use resources reserved for the stack and to assign the radio interrupts the highest priority. Section 6. Interrupts and section 7. Wireless Gecko Resources discuss interrupts and resource constraints in more detail.

For more details on EMLIB and EMDRV, refer to the Gecko SDK API documentation and various app notes.

3.3 Hardware

This folder includes board- and WSTK-related files. Only the most important ones are discussed here.

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.

3.4 Application

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
```

If using the OTA supervisor, the `app.capabilities` struct member of application properties must contain the Bluetooth stack version number:

```
#define BG_APP_PROPERTIES_CAPABILITIES (BG_VERSION_MAJOR << 24) | (BG_VERSION_MINOR << 16) | (BG_VERSION_PATCH
<< 8)
```

The application properties struct is mandatory if using the Gecko Bootloader and can reside anywhere in the flash.

gatt_db.c and gatt_db.h

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 directly edited in the Visual GATT Editor GUI in Simplicity Studio or 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*.

The gatt_db.c defines the GATT database structure and content, 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.

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.

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 Bluetooth Clocks

The clock settings are initialized in the `initMcu_clocks()` function. Clock settings include initializations of oscillators (HFXO, LFXO, PLFRCO, 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, PLFRCO, or LFRCO, depending on the accuracy requirements.

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 or PLFRCO cannot be used, 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 4.2.3 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, Chapter 12.

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

4.1.2 DC-DC Configuration

The DCDC 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-niose 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 DCDC, 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. To use these modules, the LNA needs to be correctly configured and enabled. The LNA is configured in *hal-config-board.h* using with the prefix `BSP_LNA_`.

LNA is initialized in `module_initLna()` within the `initBoard()` function if the board supports LNA.

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 `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.

4.1.5 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 Stack 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:

```
/* Gecko configuration parameters (see gecko_configuration.h) */
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_endpoint_init()`	allows the creation and deletion of endpoints as well as configuration of data routing via the endpoint APIs.
`gecko_bgapi_class_hardware_init()`	enables access and configuration of the system hardware and peripherals via the hardware APIs.
`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_USE_LAST_CTUNE`	1=Overrides CTUNE to last saved value in PS Store.
`GECKO_CONFIG_FLAG_SUPERVISOR`	0=Normal application. OTA settings need to be saved to flash. 1= Application is Supervisor for OTA and uses saved ota settings.
`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

This configures cryptography library used by the stack. Mbedtls provides Class 1 (sl_*.c) and Class 2 (slcl_*.c) plugins to use for hardware acceleration for crypto operations. The two plugin classes are not compatible with each other. The Bluetooth stack uses Class 2 plugins, so if the crypto functions are used in the application they must also use the Class 2 plugin.

Before using crypto operations, the hardware acceleration for the crypto must be initialized. This is done by calling `mbedtls_device_init()` and `mbdedtls_device_set_instance()` for the crypto device used. The initialization of the device context should be done only once. The Bluetooth stack will do this automatically for the crypto device it uses unless `GECKO_MBEDTLS_FLAGS_NO_MBEDTLS_DEVICE_INIT` flag has been set in configuration option **.mbedtls.flags**, which disables automatic initialization of crypto device. This flag needs to be set if the application handles the crypto device initialization.

Configuration option **.mbedtls.dev_number** defines what crypto device is used by Bluetooth stack if more than one is available; the default is 0.

```
.mbedtls.flags = 0,            // GECKO_MBEDTLS_FLAGS_NO_MBEDTLS_DEVICE_INIT disable automatic
.mbedtls.dev_number = 0,             // initialization of crypto device
```

4.2.3 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
```

4.2.4 Bluetooth Connections

The maximum number of simultaneous Bluetooth connections allowed by the stack is limited by the amount of memory that is allocated for the connection management. The memory is allocated during initialization in `gecko_init()`, and determined by the definition `#define MAX_CONNECTIONS`. Each connection increases the RAM usage by roughly 800 bytes.

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

```
#define MAX_CONNECTIONS 1
```

4.2.5 OTA Configuration

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

The OTA has a few configuration options configured through the OTA flags.

Note: When using the Gecko Bootloader, do not use OTA flags for security. Instead, configure security through the Gecko Bootloader.

Otherwise, **OTA flags** can be used to control the OTA Control Point and OTA Data characteristics security properties used in the OTA service. This controls the level of Bluetooth security required for performing the OTA firmware update. It is strongly recommended that security is used in the OTA flags, to prevent unauthorized devices from uploading new firmware images to the device.

```
  .ota.flags            = 0,     // 0x0 No restrictions
                                             // 0x200 Authenticated write
                                             // 0x100 Encrypted write
                                             // 0x400 Bonded write
```

The Wireless Gecko's device name, when it is in Supervisor's OTA mode, 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 *AN1045: Bluetooth Over-the-Air Device Firmware Update for EFR32xG1 and BGM11x Series Products* and *UG266: Silicon Labs Gecko Booloader User's Guide*.

4.2.6 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 two following lines must be added to the Bluetooth configuration structure.

```
 .pa.config_enable = 1,
 .pa.input = GECKO_RADIO_PA_INPUT_VBAT,
```

4.2.7 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;  // Maximum number of soft timers, up to 16,  Default: 4
```

4.2.8 Bluetooth 5 Advertisement Sets

The maximum number of advertisement sets must be defined. These sets can be used to start multiple advertisers using new `bt5_set_mode` commands. Each context allocates ~60 bytes of RAM.

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

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.

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

}
```

5.3 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.

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	What happens 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.

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.
	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.
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	BLE 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.

* In SDK 2.3 the hardware configurator reserves the PRS 0, 1, 2, 3 and 11 channels unnecessarily. These PRS channels are not used by the stack but cannot be configured in the GUI.

7.1 Flash

The Bluetooth stack is executed from the flash memory, and currently (v.2.0.0 software and higher) takes about half of flash. The flash can be split into blocks for the bootloader, the Bluetooth stack, application, and the Persistent Store (PS Store), 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 stack block includes the actual Bluetooth firmware including link layer, GAP, SM, ATT and GATT layers plus a small supervisor application for OTA upgradability and the hardware specific libraries required by the Bluetooth stack.
- The application is located between the Bluetooth stack and PS Store.
- PS Store is a non-volatile data store, 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 PS Store is located at the last 4 kB of the flash.

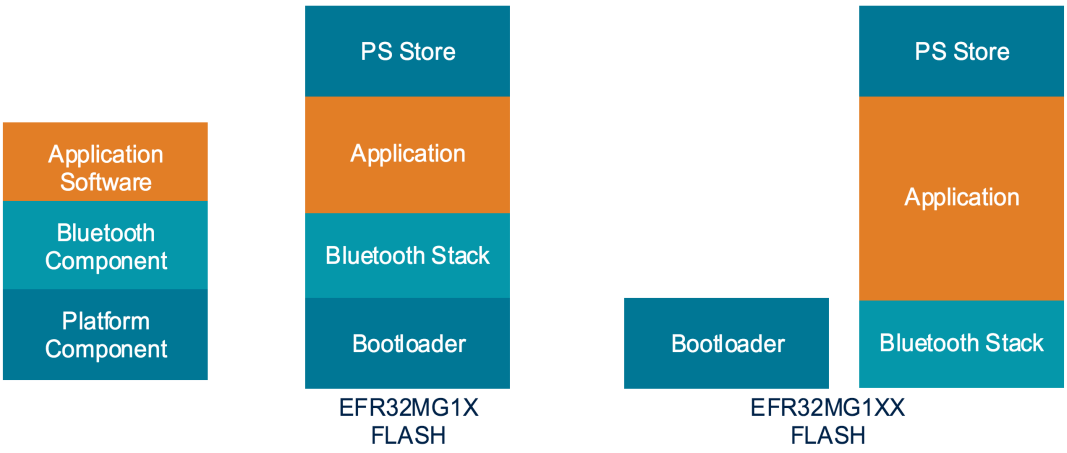


Figure 7.1. Flash Usage With and Without Separate Bootloader Flash

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

		EFR32MG1X	EFR32MG12X	EFR32MG13X	EFR32MG14X
Bootloader	Size	16	16	16	16
	Address	0-0x3FFF	0x0FE10000-0x0FE13FFF	0x0FE10000-0x0FE13FFF	0x0FE10000-0x0FE13FFF
Stack	Size	117	122	123	123
	Address	0x4000-0x20FFF	0-0x1E800	0-0x1EC00	0-0x1EC00
PS Store	Size	4	4	4	4
	Address	0x3F000-0x3FFFF	0xFF000-0xFFFFF	0x7F000-0x7FFFF	0x7F000-0x7FFFF

7.2 Linking

The Bluetooth stack is delivered as a pre-linked binary object. The application links to the Bluetooth stack as a separate binary image. For In-place OTA update, this method of linking means that the Bluetooth stack does not have any dependency on the application, which allows the Bluetooth stack to operate independently during the update. The Bluetooth stack provides support for OTA, but the application needs to start the update process.

For more information on the OTA updates and how to enable them, please refer to *AN1045: Bluetooth Over-the-Air Device Firmware Update for EFR32xG1 and BGM11x Series Products*, *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	2976 + Number of connections * 844 bytes
Bluetooth GATT database	Application-dependent (20 to 200 bytes)
C STACK	1.5 kB
C HEAP	0 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 (binstack.o) and the application programming interface (bgapi.o).

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 = 2976 + Number of connections * 844 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 around 20 to 200 bytes. See an example in section 8. Appendix: Calculating Flash and RAM Consumption.

7.3.4 C STACK

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

7.3.5 C HEAP

Application HEAP also must be reserved based on application requirements. The Bluetooth stack uses its own static memory pool for dynamic memory allocation and does not use the C HEAP at all.

7.4 RTCC

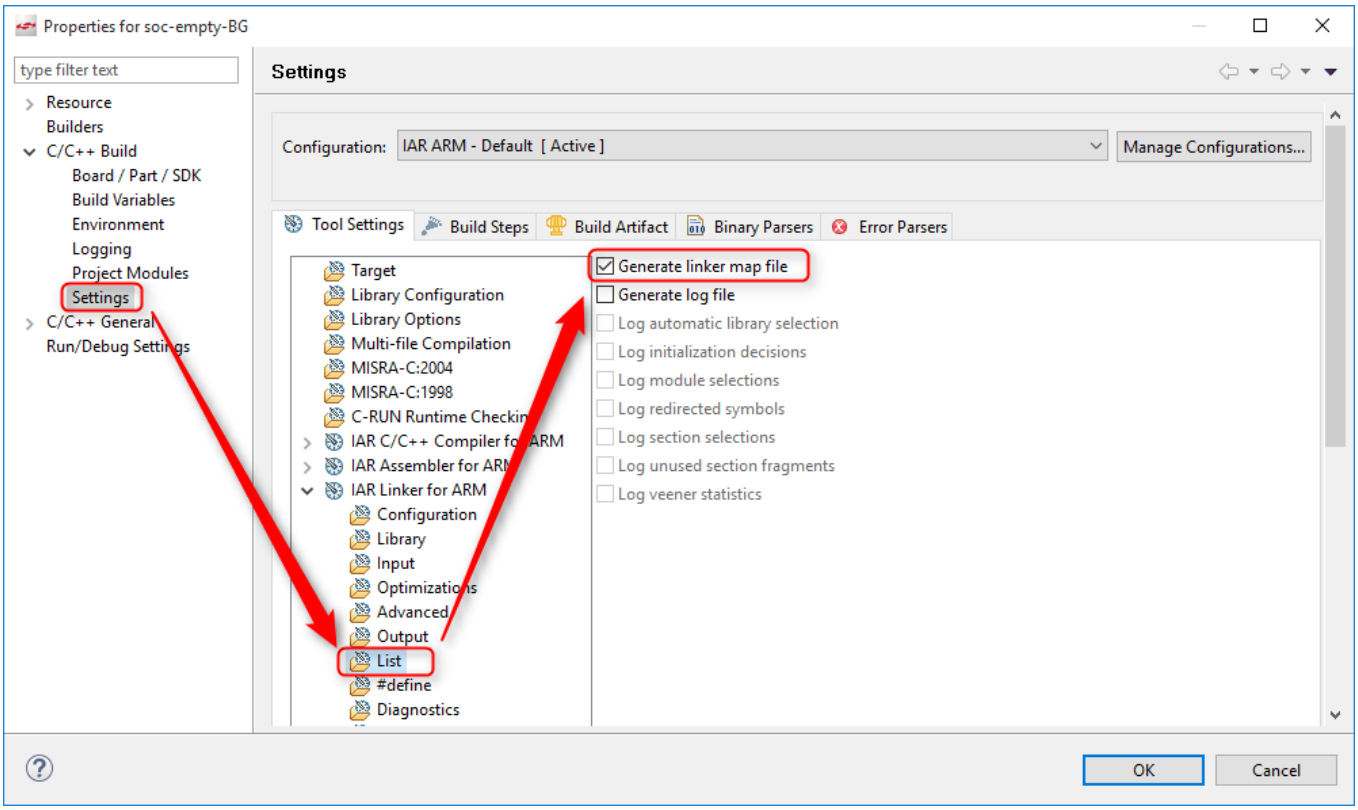
The hardware RTCC (Real Time Clock and Calendar) is set to run in counter mode by the Bluetooth stack and is reserved for the stack's use. The RTC value can, however, be read by the application, but it cannot be written by the application. The RTC value is reset every time the device boots up.

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. Appendix: Calculating Flash and RAM Consumption

To get the exact RAM and ROM consumption results a *.map* file should be generated after linking. In IAR this option must be switched on in the project settings page, as shown in the following figure.



After compilation, the map file can be found in the *IAR ARM – Default* folder in the workspace.

Look in the beginning of the file for the placement summary part. Here you can find the exact size and memory location of every object.

For example in SDK v2.3.0 the legacy OTA bootloader (binbootloader.o) is located at 0x00000000 address and consumes 0x31c (796) bytes from the flash.

```
 Section Kind Address Size Object
------- ---- ------- ---- ------
".text_bootloader": 0x4000
 bootloader 0x00000000 0x4000 <Block>
 .binbootloader const 0x00000000 0x31c binbootloader.o [1]
 bootloader const 0x0000031c 0x3ce4 <Block tail>
```

The BLE stack (binstack.o) is located at 0x00004000 and consumes 0x1c9e4 (117220) bytes.

```
 Section Kind Address Size Object
 ------- ---- ------- ---- ------
".text_stack": 0x1c9e4
 stack 0x00004000 0x1c9e4 <Block>
 .binstack const 0x00004000 0x1c9e4 binstack.o [1]
 - 0x000209e4 0x1c9e4
```

Beginning with SDK v.2.3 0 the bgapi is a separate library from the stack and linked in the application space. The bgapi was refactored to several bgapi classes. These classes can be used independently so the unused classes do not consume memory. The objects of the bgapi classes can be found in the .text_app section.

```
 Section Kind Address Size Object
 ------- ---- ------- ---- ------
".text_app": 0x4cae
 app 0x00021000 0x4cae <Block>
…
.rodata const 0x00021584 0x28 gecko_bgapi_dfu.c.obj [9]
.rodata const 0x000215ac 0x38 gecko_bgapi_endpoint.c.obj [9]
.rodata const 0x000215e4 0x30 gecko_bgapi_flash.c.obj [9]
.rodata const 0x00021614 0xa0 gecko_bgapi_gatt.c.obj [9]
.rodata const 0x000216b4 0x48 gecko_bgapi_gatt_server.c.obj [9]
.rodata const 0x000216fc 0x78 gecko_bgapi_hardware.c.obj [9]
.rodata const 0x00021774 0x28 gecko_bgapi_le_connection.c.obj [9]
.rodata const 0x0002179c 0x70 gecko_bgapi_gap.c.obj [9]
.rodata const 0x0002180c 0x80 gecko_bgapi_sm.c.obj [9]
.rodata const 0x0002188c 0x38 gecko_bgapi_system.c.obj [9]
.rodata const 0x000218c4 0x30 gecko_bgapi_test.c.obj [9]
.rodata const 0x000218f4 0x18 gecko_bgapi_util.c.obj [9]
.rodata const 0x0002190c 0xc bgapi_sm.c.obj [9]
.rodata const 0x00021918 0xc bgapi_sm.c.obj [9]
.rodata const 0x00021924 0x18 adc.c.obj [9]
.rodata const 0x0002193c 0x54 uart.c.obj [9]
.rodata const 0x00021990 0xc uart.c.obj [9]
.rodata const 0x0002199c 0x28 uart.c.obj [9]
…
.text ro code 0x00023e2c 0x58 gecko_bgapi_le_connection.c.obj [9]
.text ro code 0x00023e84 0x58 gecko_bgapi_gatt.c.obj [9]
.text ro code 0x00023edc 0x58 gecko_bgapi_gatt_server.c.obj [9]
.text ro code 0x00023f34 0x58 gecko_bgapi_endpoint.c.obj [9]
.text ro code 0x00023f8c 0x58 gecko_bgapi_hardware.c.obj [9]
.text ro code 0x00023fe4 0x58 gecko_bgapi_flash.c.obj [9]
.text ro code 0x0002403c 0x58 gecko_bgapi_test.c.obj [9]
.text ro code 0x00024094 0x58 gecko_bgapi_sm.c.obj [9]
.text ro code 0x000240ec 0x58 gecko_bgapi_util.c.obj [9]
.text ro code 0x00024144 0x50 bgapi_dfu.c.obj [9]
.text ro code 0x00024194 0xf6 bgapi_system.c.obj [9]
..
.text ro code 0x0002428c 0x248 bgapi_gap.c.obj [9]
.text ro code 0x000244d4 0xd4 bgapi_le_connection.c.obj [9]
.text ro code 0x000245a8 0x454 bgapi_gatt_lib.c.obj [9]
.text ro code 0x000249fc 0x192 bgapi_endpoint.c.obj [9]
..
.text ro code 0x00024b90 0x184 bgapi_hardware.c.obj [9]
.text ro code 0x00024d14 0x154 bgapi_ps.c.obj [9]
.text ro code 0x00024e68 0x200 bgapi_test_lib.c.obj [9]
.text ro code 0x00025068 0x2b8 bgapi_sm.c.obj [9]
.text ro code 0x00025320 0x14c bgapi_util.c.obj [9]

...etc.
```

CSTACK and HEAP usage can also be checked. In this example the CSTACK starts at 0x20003038 and the HEAP starts at 0x20003818 and both are 2 K.

```
 Section Kind Address Size Object
 ------- ---- ------- ---- ------

 CSTACK 0x20003038 0x800 <Block>
 CSTACK uninit 0x20003038 0x800 <Block tail>
 - 0x20003838 0x800

 HEAP 0x20003818 0x800 <Block>
 HEAP uninit 0x20003818 0x800 <Block tail>
 - 0x20004018 0x1000
```

To calculate the RAM usage of the GATT database, add the occupied memory space by gatt_db.o in .data and in .bss sections. The example below shows the RAM usage of the GATT database from the empty-soc-example. It uses 20 bytes of RAM.

```
Section Kind Address Size Object
------- ---- ------- ---- ------
"P3", part 1 of 3: 0x34
 P3-1 0x20003000 0x34 <Init block>
..
 .data inited 0x20003004 0x10 gatt_db.o [2]
…
"P3", part 3 of 3: 0x1c20
..
 .bss zero 0x20005424 0x4 gatt_db.o [2]
..
```

To see overall flash and RAM usage, for every memory section look for the first line of the module summary for the location and size.

```
 Section Kind Address Size Object
 ------- ---- ------- ---- ------
".text_app": 0x4cae
 app 0x00021000 0x4cae <Block>
```

Add these numbers for every section to calculate the overall FLASH and RAM usage.

The following sections consumes flash:

- .text_bootloader
- .text_stack
- .text_app

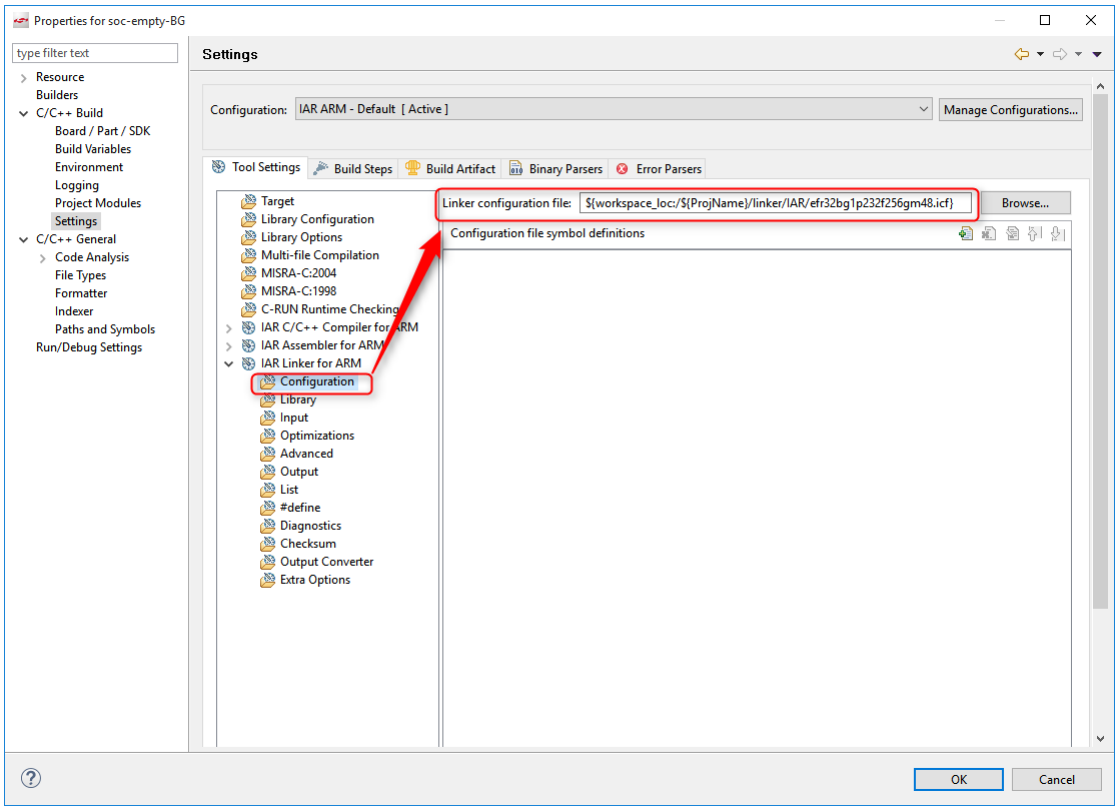
The following sections consume RAM:

- P3.data
- P3.CSTACK
- P3.HEAP
- P3.bss

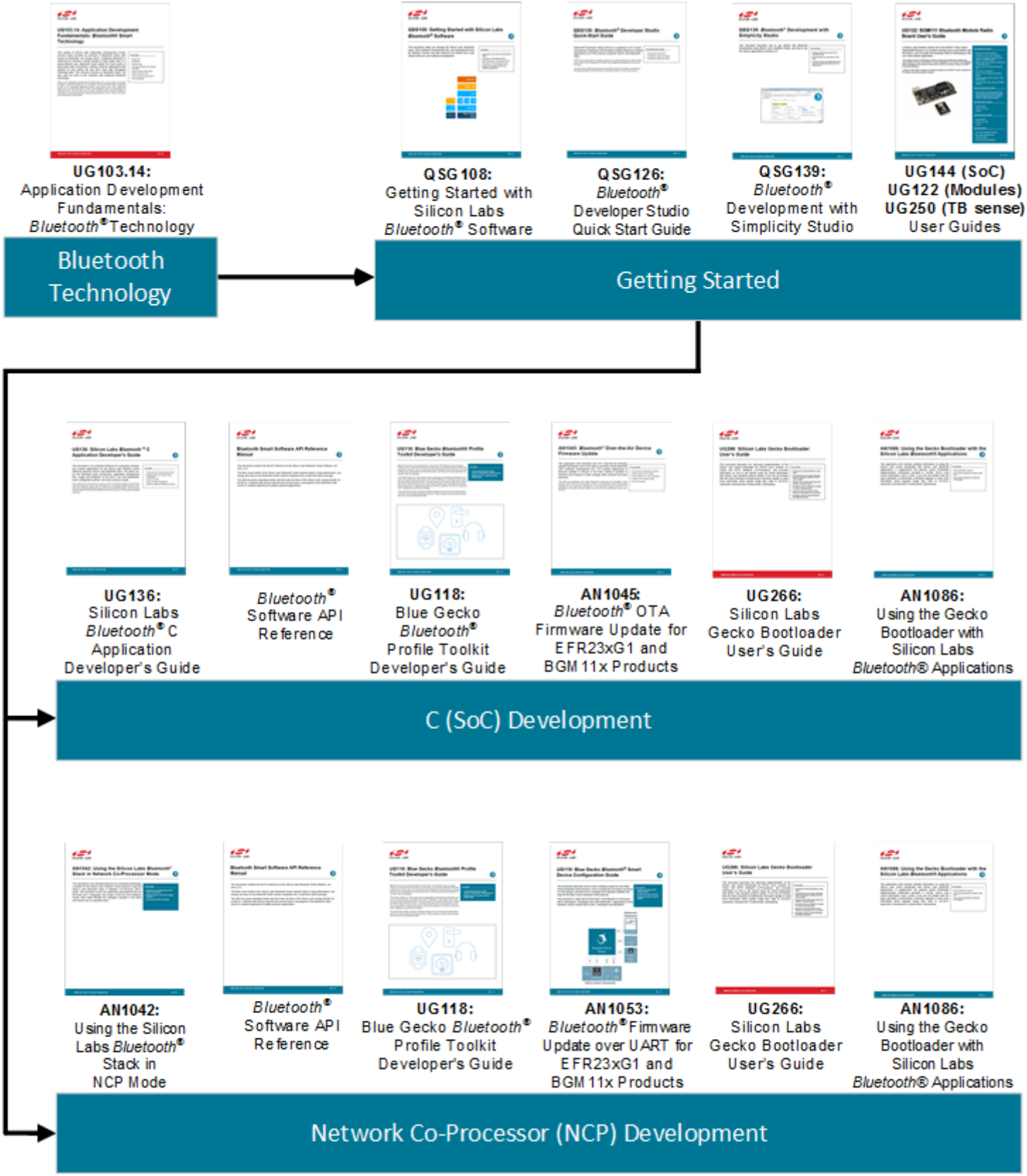
One additional 12 K RAM section reserved for the Bluetooth stack is not listed in the *.map* file but rather in the IAR linker script file.

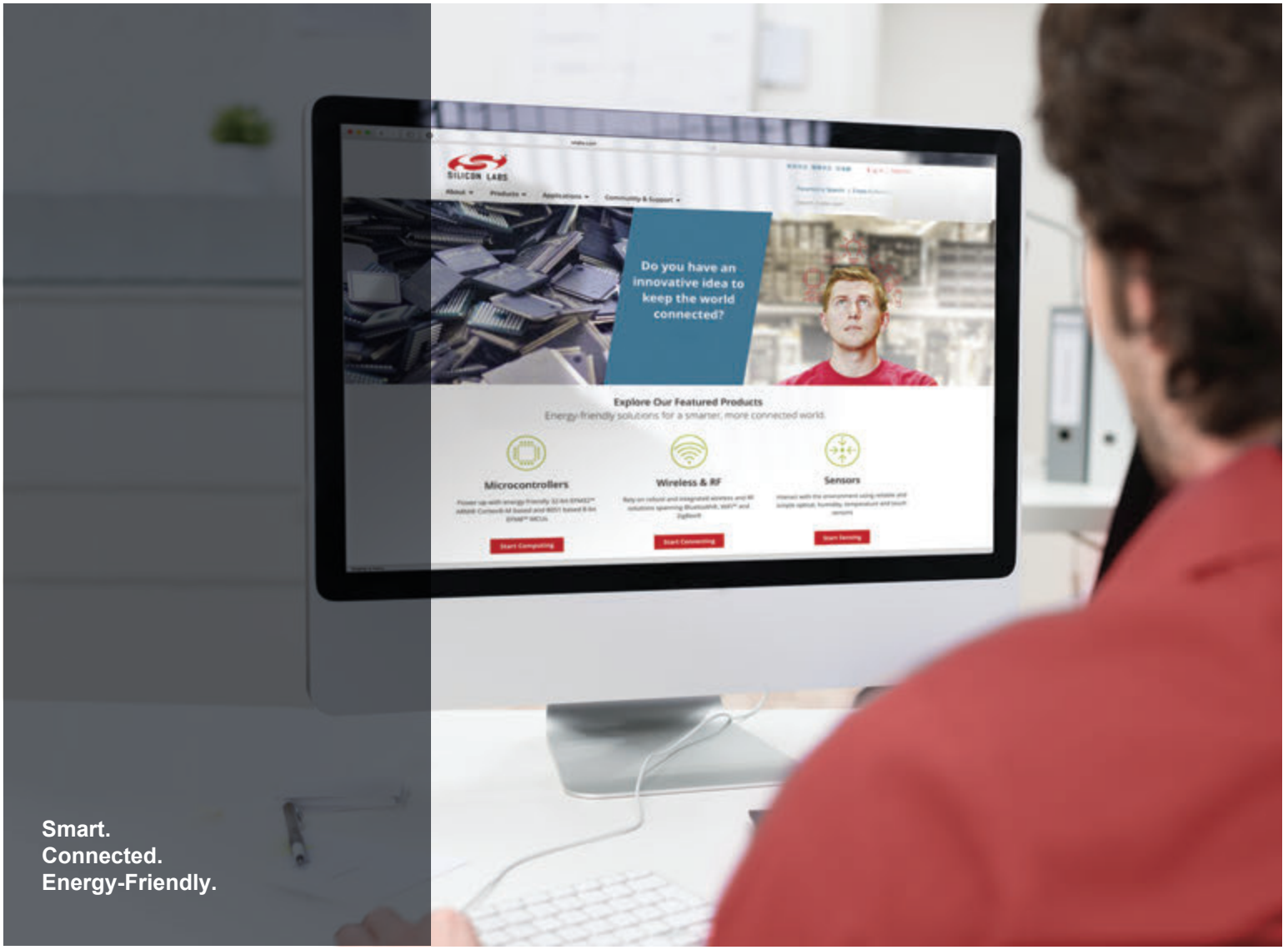
```
/*12KiB ram for stack*/
define symbol blob_ram_size = 0x3000;
define region RAM_region = mem:[from __ICFEDIT_region_RAM_start__+blob_ram_size to __ICFEDIT_region_RAM_end__];
```

This file can be found in the project workspace but is different for each target. See the project properties to locate the used *.icf* file.


Properties for soc-empty-BG
type filter text
Resource
Builders
C/C++ Build
Board / Part / SDK
Build Variables
Environment
Logging
Project Modules
Settings
C/C++ General
Code Analysis
File Types
Formatter
Indexer
Paths and Symbols
Run/Debug Settings
Settings
Configuration: IAR ARM - Default [Active]
Manage Configurations...
Tool Settings
Build Steps
Build Artifact
Binary Parsers
Error Parsers
Target
Library Configuration
Library Options
Multi-file Compilation
MISRA-C:2004
MISRA-C:1998
C-RUN Runtime Checking
IAR C/C++ Compiler for ARM
IAR Assembler for ARM
IAR Linker for ARM
Configuration
Library
Input
Optimizations
Advanced
Output
List
#define
Diagnostics
Checksum
Output Converter
Extra Options
Linker configuration file: ${workspace_loc:/${ProjName}/linker/IAR/efr32bg1p232f256gm48.icf}
Browse...
Configuration file symbol definitions
OK
Cancel

9. Documentation





Products
www.silabs.com/products

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

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 and limitation to product information, specifications, and descriptions herein, and does not give warranties as to the accuracy or completeness of the included information. Silicon Labs shall have no liability for the consequences of use of the information supplied herein. This document does not imply or express copyright licenses granted hereunder to design or fabricate any integrated circuits. The products are not designed or authorized to be used within any Life Support System 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.

Trademark Information
Silicon Laboratories Inc.® , Silicon Laboratories®, Silicon Labs®, SiLabs® and the Silicon Labs logo®, Bluegiga®, Bluegiga Logo®, Clockbuilder®, CMEMS®, DSPLL®, EFM®, EFM32®, EFR, Ember®, Energy Micro, Energy Micro logo and combinations thereof, "the world's most energy friendly microcontrollers", Ember®, EZLink®, EZRadio®, EZRadioPRO®, Gecko®, ISOmodem®, Micrium, Precision32®, ProSLIC®, Simplicity Studio®, SiPHY®, Telegesis, the Telegesis Logo®, USBXpress®, Zentri 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. 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

http://www.silabs.com