Connect

- Developing with Connect
- Release Notes
  - Connect
- Getting Started
  - Overview
  - Proprietary Flex SDK v3.x Quick Start Guide
    - Overview
    - Proprietary Flex SDK Product Overview
    - Getting Started with Simplicity Studio 5 and the Proprietary Flex SDK
    - Working With Example Applications
    - Next Steps
- Fundamentals
  - Overview
  - Connect Fundamentals
    - Introduction
    - Overview
    - Connect Modes of Operation
    - Stack Structure
    - IEEE 802.15.4 Support RAIL vs Connect
    - Next Steps
  - Using Silicon Labs Connect v3.x with IEEE 802.15.4 (PDF)
  - Wireless Network App Development Fundamentals
    - Introduction
    - Overview
    - Embedded Networking
    - Radio Fundamentals
    - Networking Basic Concepts
    - Wireless Networking
    - Networking Devices
- Connect Developer's Guide
  - Overview
  - Developing and Debugging
    - Overview
    - Architecture of the Silicon Labs Connect Stack v3.x (PDF)
    - Customizing Applications with Silicon Labs Connect v3.x (PDF)
    - Network Co-Processor Applications with Silicon Labs Connect v3.x (PDF)
    - Using Real Time Operating Systems with Silicon Labs Connect v3.x (PDF)
    - Energy Saving with Silicon Labs Connect v3.x (PDF)

Copyright © 2025 Silicon Laboratories. All rights reserved.

- Building Low Power Networks with the Silicon Labs Connect Stack v3.x
  - Overview
  - Introducing Low Power Sensor Networks
  - Create the Baseline Network
  - Reduce Power Consumption
- EFR32 Radio Configurator Guide for Simplicity Studio 5 (PDF)
- PHY Limitations and Timing Optimization

Bootloading
- Overview
- Bootloader Fundamentals
  - Introduction
  - Overview
  - About the Gecko Bootloader
  - Memory Space for Bootloading
  - Design Decisions
- Bootloading and OTA with Silicon Labs Connect v3.x (PDF)
- Using the Gecko Bootloader with Silicon Labs Connect
  - Introduction
  - Overview
  - Using the Gecko Standalone Bootloaders
  - Using the Gecko Application Bootloader
- Gecko Bootloader User's Guide for Series 3 and Higher
  - Introduction
  - Gecko Bootloader File Format v4
  - Gecko Bootloader Operation Application Upgrade
  - Gecko Bootloader Operation Bootloader Upgrade
  - Gecko Bootloader Operation Secure Engine Upgrade
  - Getting Started with the Gecko Bootloader
  - Configuring the Gecko Bootloader
  - Simplicity Commander and the Gecko Bootloader
  - Gecko Bootloader Security Features
  - Application Interface
- Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher
  - Introduction
  - Overview
  - Gecko Bootloader File Format
  - Gecko Bootloader Operation Application Upgrade
  - Gecko Bootloader Operation Bootloader Upgrade
  - Gecko Bootloader Operation Secure Engine Upgrade
  - Getting Started with the Gecko Bootloader
  - Configuring the Gecko Bootloader
  - Simplicity Commander and the Gecko Bootloader
  - Gecko Bootloader Security Features
  - Gecko Bootloader and Trustzone

Copyright © 2025 Silicon Laboratories. All rights reserved.

- Gecko Bootloader and Delta DFU Upgrades
- Application Interface
- Series 2 Secure Boot with RTSL
  - Introduction
  - Series 2 Device Security Features
  - Secure Boot Process
  - Examples
  - Debugging On Secure Boot Enabled Device
  - Failure Analysis
- Transitioning to the Updated Gecko Bootloader in GSDK 4.0 and Higher
  - Overview
  - Introduction
  - About Projects
  - Components
  - Pin Tool
  - Linker File
  - Additional Macros
  - Postbuild Steps
  - Main Bootloader in Main Flash
  - Callbacks
  - Storage Slots
  - App Properties

Multiprotocol

- Overview
- Multiprotocol Fundamentals
  - Introduction
  - Types of Multiprotocol Implementations
  - Protocol Considerations
  - Radio Scheduler
- Dynamic Multiprotocol
  - Introduction
  - The Radio Scheduler
  - Radio Scheduler Examples
  - Implementing Multiprotocol with an 802.15.4-Based Stack
  - Implementing Multiprotocol with RAIL
  - Implementing Multiprotocol with Bluetooth

Non-Volatile Data Storage

- Overview
- Non-Volatile Data Storage Fundamentals
  - Overview
  - Introduction
  - Implementations of Non-Volatile Data Storage
  - Dynamic Data Storage Implementations
  - Comparing Non-Volatile Data Storage Implementations

Copyright © 2025 Silicon Laboratories. All rights reserved.

- Using Third Generation Non-Volatile Memory (NVM3) Data Storage
  - Introduction
  - Overview
  - NVM3 Default Instance
  - NVM3 in the Simplicity Studio 5 Project Configurator
  - Using NVM3 with Silicon Labs Connect
  - Using NVM3 with Silicon Labs OpenThread Applications
  - Using NVM3 with Zigbee EmberZNet v.7x And Higher
  - Using NVM3 with Silicon Labs Bluetooth Applications
  - Using NVM3 In Appbuilder Based Applications
  - Using NVM3 with Z Wave Applications
  - NVM3 API Options
  - Simplicity Commander and NVM3
  - Securing NVM3 Data on External Flash Devices
  - Best Practices and Optimizations For NVM3
- Bringing Up Custom Devices for the EFR32MG and EFR32FG Families
  - Overview
  - Introduction
  - EFR32 Wireless System
  - Setting Manufacturing Tokens
  - Testing
- Using Tokens for Non-Volatile Data Storage
  - Overview
  - Introduction
  - About Tokens
  - Creating and Accessing Tokens
  - Token Manager Component

Security

- Overview
- Security Key Migration
- IoT Endpoint Security Fundamentals
  - Introduction
  - Overview
  - No Universal Passwords
  - Secured Interfaces
  - Proven Cryptography
  - Security by Default
  - Signed Software Updates
  - Automatically Applied Updates
  - Vulnerability Reporting Program
  - Security Expiration Date
  - Next Steps
  - Series 2 Device Security Features
- Series 2 Secure Debug

Copyright © 2025 Silicon Laboratories. All rights reserved.

- Introduction
- Series 2 Device Security Features
- Introduction To Secure Debug
- Secure Engine Subsystem
- Debug Lock
- Debug Unlock
- Examples
- Precautions
- Failure Analysis

Production Programming of Series 2 Devices (PDF)

Anti-Tamper Protection Configuration and Use

- Overview
- Series 2 Device Security Features
- Introduction
- Secure Engine Manager
- Tamper Responses
- Tamper Sources
- Anti-Tamper Configuration
- Usage Example
- Tamper Disable
- Examples

Authenticating Silicon Labs Devices using Device Certificates

- Overview
- Series 2 Device Security Features
- Introduction
- Secure Identification on HSE-SVH Devices
- Device Certificate Options
- Entity Attestation Token (EAT)
- Remote Authentication Process
- Secure Engine Manager
- Examples

Secure Key Storage

- Overview
- Series 2 Device Security Features
- Introduction
- HSE Secure Key Storage
- TrustZone Secure Key Storage
- Secure Key Storage Implementations
- Examples

Programming Series 2 Devices Using the DCI and SWD (PDF)

Integrating Crypto Functionality with PSA Crypto vs. Mbed TLS (PDF)

Connect API Reference Guide

Connect Stack API Reference

Connect Stack Version

Copyright © 2025 Silicon Laboratories. All rights reserved.

- Connect Data Types
  - EmberNetworkParameters
  - EmberIncomingMessage
  - EmberOutgoingMessage
  - EmberMacAddress
  - EmberMacFrame
  - EmberIncomingMacMessage
  - EmberOutgoingMacMessage
  - EmberKeyData
  - sl_connect_radio_priority
  - EventActions
  - Event_s
  - EventQueue_s
  - EmberEventControl
  - EmberEventData_S
  - EmberTaskControl
- Stack Information
  - Stack Counters
- Network Management
  - Frequency Hopping
  - Parent Support
- Radio Stream
- Configuration
- Status Codes
- Stack Tokens
  - tokTypeStackKey
  - tokTypeStackNodeData
  - tokTypeStackChildTableEntry
- Event Scheduling
- Memory Buffer

Messaging

Connect Application Framework API Reference

- Application Framework Common
- Command Interpreter Plugin
- Debug Print Plugin
- Mailbox Client Plugin
- Mailbox Server Plugin
- Mailbox Common
- Ota Unicast Bootloader Client Plugin
- Ota Unicast Bootloader Server Plugin
- Ota Unicast Bootloader Common
- Ota Broadcast Bootloader Client Plugin
- Ota Broadcast Bootloader Server Plugin
- Ota Broadcast Bootloader Common

Copyright © 2025 Silicon Laboratories. All rights reserved.

- Poll Plugin
- WSTK Sensors Plugin
- Connect RAIL DMP Component
- Hardware Abstraction Layer (HAL) API Reference
  - Hardware Abstraction Layer (HAL)
  - Common Microcontroller Functions
    - RTCCRamData
  - Token Access
    - Tokens
  - Sample APIs for Peripheral Access
    - Serial UART Communication
    - Button Control
    - Buzzer Control
    - LED Control
    - Flash Memory Control
  - System Timer Control
  - Symbol Timer Control
  - HAL Configuration
    - Sample Breakout Board Configuration
    - IAR PLATFORM_HEADER Configuration
    - Common PLATFORM_HEADER Configuration
    - NVIC Configuration
    - Reset Cause Type Definitions
  - HAL Utilities
    - Crash and Watchdog Diagnostics
    - Cyclic Redundancy Code (CRC)
    - Random Number Generation
    - Network to Host Byte Order Conversion
- Migration Guide
- Deprecated List
- Connect

Training

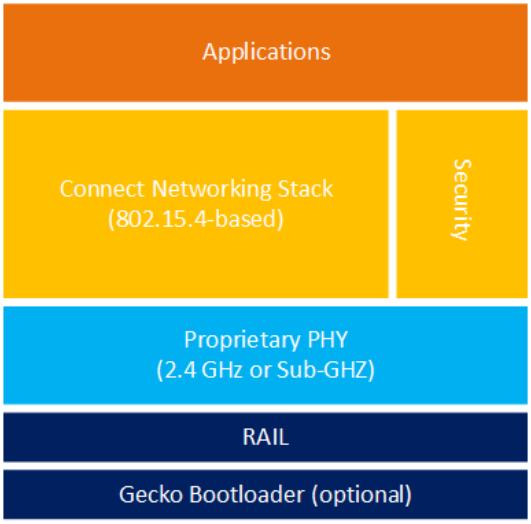
Copyright © 2025 Silicon Laboratories. All rights reserved.

Developing with Connect

Developing Proprietary Connect Applications

Silicon Labs Connect is an IEEE 802.15.4 MAC-based wireless networking stack for a variety of proprietary applications optimized for low-power devices. This full-featured, easily customizable networking stack is designed for compliance with regulatory specifications across worldwide geographic regions and supports both Sub-GHz and 2.4 GHz frequency bands.

The Silicon Labs Connect stack supports many combinations of radio modulation, frequency and data rates. The stack provides support for end nodes, coordinators, and range extenders. It includes all wireless MAC (Medium Access Control) layer functions such as scanning and joining, setting up a point-to-point or star network, and managing device types such as sleepy end devices, routers, and coordinators. With all this functionality already implemented in the stack, users can focus on their end application development and not worry about the lower-level radio and network details.



The Connect stack is part of the Silicon Labs Flex SDK (Software Development Kit), installed through Simplicity Studio. Connect runs on top of RAIL (Radio Abstraction Interface Layer), also included with the Flex SDK. RAIL provides an intuitive, easily-customizable radio interface layer that is designed to support proprietary or standards-based wireless protocols.

The content on these pages is intended for those who want to experiment with or are already developing an application using the Silicon Labs Connect Stack.

For details about this release: Links to the Flex SDK release notes are available on the silabs.com Gecko SDK page.

For Silicon Labs' Connect product information: See the product pages on silabs.com.

For background about the Connect stack and other wireless networking topics: The Fundamentals section is a good place to start.

To get started with development: See the Getting Started section to get started working with example applications.

If you are already in development: See the Developer's Guide for details or go directly to the API Reference.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Connect

Connect Version 4.1.1 (July 21, 2025) - Release Notes

Simplicity SDK Version 2025.6.1

The Connect SDK is a complete software development suite for proprietary wireless applications that was previously part of the Proprietary SDK. Starting with the Connect SDK 4.0.0 release, Proprietary SDK is split into RAIL SDK and Connect SDK.

Connect SDK uses Silicon Labs Connect, an IEEE 802.15.4-based networking stack designed for customizable broad-based proprietary wireless networking solutions that require low power consumption and operate in either the sub-GHz or 2.4 GHz frequency bands. The solution is targeted towards simple network topologies.

Connect SDK is supplied with extensive documentation and sample applications. All examples are provided in source code within the Connect SDK sample applications.

Release Summary

Key Features | API Changes | Bug Fixes | Chip Enablement

Key Features

Added in 4.1.1

None.

Added in 4.1.0

- Added Connect RAIL DMP component to support custom proprietary communication next to the Connect network. This component is able to handle additional stacks, for example a setup with Connect-BLE-RAIL DMP.
- Connect Stack and SDK was updated to use new RAIL 2.19.0 APIs.

API Changes

None.

Bug Fixes

Fixed in 4.1.1

- Fixed an issue where `sendCallbackCommand` accepted NULL pointer
- Fixed missing radio configuration file for FGM23 OPNs
- Fixed an issue where CCA/CSMA error wasn't handled

Fixed in 4.1.0

- `emberAfStackStatusCallback()` was not called if application is RTOS.
- RAIL Assert codes was missing from Connect Assert.

Chip Enablement

Copyright © 2025 Silicon Laboratories. All rights reserved.

Added in 4.1.1

None.

Added in 4.1.0

EFRxG29

Key Features

New Features | Enhancements | Removed Features | Deprecated Features

New Features

Added in 4.1.1

None.

Added in 4.1.0

- Added Connect RAIL DMP component to support custom proprietary communication next to Connect network, component is able to handle additional stack like a setup with Connect-BLE-RAIL DMP.
- Connect Stack and SDK was updated to use new RAIL 2.19.0 APIs.

Enhancements

Added in 4.1.1

None.

Added in 4.1.0

Updated sample applications to use `sl_main` instead of `sl_system`. `main.c` is now provided by platform with improved optimization.

Removed Features

None.

Deprecated Features

None.

API Changes

New APIs

Added in 4.1.1

None.

Added in 4.1.0

New API Signature	Deprecated API replaced by this (if any)
void rail_app_task_init(void)	
void rail_app_task(void *p_arg)	
void handle_os_pending(void)	
void post_os_flag(void)	

Copyright © 2025 Silicon Laboratories. All rights reserved.

New API Signature	Deprecated API replaced by this (if any)
sl_rail_time_t calculate_transaction_time(sl_rail_handle_t rail_handle)	
EmberStatus emberSetRadioRxFifo(uint16_t rxFifoSize)	

Modified APIs

None.

Removed APIs

None.

Deprecated APIs

None.

Bug Fixes

Fixed in 4.1.1

ID	Issue Description	GitHub / Salesforce Reference (if any)	Affected Software Variants, Hardware, Modes, Host Interfaces
1478113	`sendCallbackCommand` accepted NULL pointer.	None	• Connect Stack • All • SoC, NCP mode
1474777	Missing radio configuration file for FGM23 OPNs.	None	• Connect Stack • All • SoC, NCP mode
1465062	CCA/CSMA TX Scheduling error wasn't handled in the Connect Stack.	None	• Connect Stack • All • SoC, NCP mode

Fixed in 4.1.0

ID	Issue Description	GitHub / Salesforce Reference (if any)	Affected Software Variants, Hardware, Modes, Host Interfaces
1456611	If RTOS is enabled, emberAfStackStatusCallback() is not called.	None	• Connect Stack • All • SoC, NCP mode
1402669	RAIL LIB assert codes are not coming out from the Connect Stack.	None	• Connect Stack • All • SoC, NCP mode

Chip Enablement

Added in 4.1.1

None.

Added in 4.1.0

Copyright © 2025 Silicon Laboratories. All rights reserved.

Chip Family	OPNs / Boards / OPN Combinations	Supported Software Variants (if applicable)	Supported Modes	Supported Host Interfaces
EFR32xG29	• OPN: EFR32MG29B, EFR32BG29B • Boards: BRD1021A,BRD4412A,BRD4413A,BRD4414A,BRD4420A • External Hosts: hosts	All sample apps	SoC	UART,EUSART,SPI

Application Example Changes

New Examples | Modified Examples | Removed Examples | Deprecated Examples

New Examples

None.

Modified Examples

Modified in 4.1.1

None.

Modified in 4.1.0

Example Name	Changes	Supported Software Variants if applicable	Supported Modes	Supported OPNs / Boards / OPN Combinations	Supported Host Interfaces
Connect - NCP sample application	• Updated to use `sl_main` instead of `sl_system` • Added `app.c` instead of `main.c` for custom code	Standard	NCP	• OPN: All • Boards: All supported boards	SPI, EUART, UART

Removed Examples

None.

Deprecated Examples

None.

Known Issues and Limitations

Copyright © 2025 Silicon Laboratories. All rights reserved.

ID	Issue or Limitation Description	GitHub / Salesforce Reference (if any)	Workaround (if any)	Affected Software Variants, Hardware, Modes, Host Interfaces
652925	EFR32XG21 is not supported for “Flex (Connect) - SoC Light Example DMP” and “Flex (Connect) - SoC Switch Example”	None.	None.	• EFR32XG21 • BRD4176A,BRD4179B,BRD4180A,BRD4180B,BRD4181A,BRD4181B,BRD4181C,E • SoC
1139850	DMP instabilities with EFR32XG27	None.	None.	• EFR32XG27 • BRD2602A,BRD4109A,BRD4110A,BRD4110B,BRD4111A,BRD4111B,BRD4113A,B • SoC
501561	In the Legacy HAL component, the PA configuration is hardcoded regardless of the user or boards settings.	None.	Until this is changed to properly pull from the configuration header, the file ember-phy.c in the user's project will need to be modified by hand to reflect the desired PA mode, voltage, and ramp time.	• All • All • SoC
None.	When running the RAIL Multiprotocol Library (used, for example, when running DMP Connect+BLE), IR Calibration is not performed because of a known issue in the RAIL Multiprotocol Library. As result, there is an RX sensitivity loss in the order of 3 or 4 dBm.	None.		• All • All • SoC
None.	Updating the Baremetal sample applications to RTOS causes build errors.	None.	Follow the Platforms guide sl_main Baremetal to RTOS Application Transition	• Connect SDK Baremetal sample applications • All supported boards • SoC, NCP mode • UART,SPI interface

Copyright © 2025 Silicon Laboratories. All rights reserved.

Impact of Release Changes

Impact Statements | Migration Guide

Impact Statements

None.

Migration Guide

Click here for the migration guide for deprecated, removed, and modified items.

Using This Release

What's in the Release? | Compatible Software | Installation and Use | Help and Feedback

What's in the Release?

Connect SDK 4.1.0 showcasing the usage of the Connect stack 4.1.0 with the lates Simplicity SDK.

Compatible Software

None.

Installation and Use

To upgrade your existing software with this release, see instructions **here**.

To run your first demo, see our Getting Started Guides.

To kick start your development, see our Developer's Guide.

To learn more about the software in this release, dive into our online documentation.

For information about Secure Vault Integration, see here.

To review Security and Software Advisory notifications and manage your notification preferences:

1. Go to https://community.silabs.com/.
2. Log in with your account credentials.
3. Click your profile icon in the upper-right corner of the page.
4. Select **Notifications** from the dropdown menu.
5. In the Notifications section, go to the **My Product Notifications** tab to review historical Security and Software Advisory notifications
6. To manage your preferences, use the **Manage Notifications** tab to customize which product updates and advisories you receive.

Help and Feedback

- Contact Silicon Labs Support.
- To use our **Ask AI** tool to get answers, see the search field at the top of this page.

> **Note: Ask AI** is experimental.

- Get help from our developer community.

SDK Release and Maintenance Policy

Copyright © 2025 Silicon Laboratories. All rights reserved.

See our SDK Release and Maintenance Policy.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Overview

Getting Started with Silicon Labs Connect Development

To get started with Silicon Labs Connect development, download the Simplicity Studio Development environment as described in the Simplicity Studio 5 User's Guide.

This will also prompt you to install the Gecko SDK (GSDK). The GSDK combines Silicon Labs wireless software development kits (SDKs) such as the Flex SDK and Gecko Platform into a single, integrated package. The GSDK is your primary tool for developing in the Silicon Labs IoT Software ecosystem. All of Silicon Labs' stacks are written in-house to provide a seamless experience from silicon to tools, allowing you to unlock powerful features with ease, including:

- Abstraction of complex requirements like multiprotocol and pre-certification
- Industry-leading ability to support a large number of nodes
- Ultra-low power consumption
- Strong network reliability

Silicon Labs also helps future-proof your devices with over-the-air software and security updates, helping to minimize maintenance cost and improve your end user product experience!

The Flex SDK includes both the Connect library and examples and the RAIL (Radio Abstraction Interface Layer) Library and examples. Once you have downloaded Simplicity Studio and the GSDK, detailed instructions for using the Connect examples and configuration tools are provided in the **Proprietary Flex SDK Quick-Start Guide**.

Note: The recommended method to get started with the GSDK is to first install Simplicity Studio 5, which will set up your development environment and walk you through the GSDK installation. Alternatively, GSDK and other required tools may be installed manually from the GitHub GSDK site.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Proprietary Flex SDK v3.x Quick Start Guide

Proprietary Flex SDK v3.x Quick-Start Guide

NOTE: This section replaces *QSG168: Proprietary Flex SDK v3.x Quick Start Guide*. Further updates to this quick start guide note will be provided here.

This quick start guide provides basic information on configuring, building, and installing applications for the EFR32 using the Silicon Labs Flex Software Development Kit (SDK) v3.x with Simplicity Studio® 5. The Flex SDK provides two paths to application development. The first begins with Silicon Labs RAIL (Radio Abstraction Interface Layer), which is an intuitive, versatile radio interface layer that is designed to support the implementation of arbitrary wireless protocols.

The second uses the Silicon Labs Connect protocol stack, which provides a fully-featured, easily-customizable wireless networking solution optimized for devices that require low power consumption and are used in a simple network topology.

This guide is designed for developers who are new to the Silicon Labs Flex SDK, the Simplicity Studio development environment, and Silicon Labs development hardware. It provides instructions to get started using both the Connect and RAIL examples provided with the Flex SDK used on the EFR32.

Proprietary is supported on all EFR32FG devices. For others, check the device's data sheet under Ordering Information > Protocol Stack to see if Proprietary is supported. In Proprietary SDK version 2.7.n, Connect is not supported on EFR32xG22.

Key Points

- Product overview
- Setting up the development environment
- Discovering the SDK
- Working with example applications.
- Using the Pin Tool

Copyright © 2025 Silicon Laboratories. All rights reserved.

Proprietary Flex SDK Product Overview

Proprietary Flex SDK Product Overview

The Silicon Labs Proprietary Flex SDK supports developers who wish to take advantage of configurable protocol functionality provided in Silicon Labs Connect and the underlying RAIL library, as well as those who wish to start application development on top of RAIL to develop custom lower-level radio control, and network protocols.

This section covers:

- Background information on RAIL, Connect, and the example applications included with the Flex SDK
- Background information on the Gecko Bootloader
- Prerequisites for application development using the Flex SDK
- Support
- Documentation

About Connect and RAIL

The Flex SDK provides two paths to application development. The first begins with Silicon Labs RAIL (Radio Abstraction Interface Layer), which is an intuitive, versatile radio interface layer that is designed to support the implementation of arbitrary wireless protocols. The second uses Silicon Labs Connect protocol stack, which provides a fully-featured, easily-customizable wireless networking solution optimized for devices that require low power consumption and are used in a simple network topology. Connect example functionality is provided through easily-configurable components that can be turned on or off as desired.

Whether you begin development with Connect or RAIL is determined by the example application you select as a starting point in Simplicity Studio. Silicon Labs recommends that you start from a Connect example if you want to include the following functions without further development:

- MAC layer functionality including frequency hopping and security
- Network formation and, for star networks, routing support
- Application-level functionality, including diagnostics, I/O, mailbox, and sleepy end device management
- Bootloading, including serial and Broadcast or Unicast OTA (over-the-air)

The following sections provide additional detail on Connect and RAIL, including brief descriptions of the example applications. When you create a project based on an example, the Project Details on the Simplicity Studio IDE Overview tab provides additional detail about the example and interfacing with it.

Silicon Labs Connect

Silicon Labs Connect functionality for the EFR32 is implemented on top of the RAIL library. Silicon Labs Connect supports many combinations of radio modulation, frequency and data rates. The stack includes all MAC layer functions such as scanning and joining, setting up of a point-to-point or star network, device types such as sleepy end nodes, routers or coordinators, radio configuration, frequency hopping and LBT (Listen Before Talk) protocols required for regulatory compliance in each geographical region, and PHY configuration for each of these regions. With all this functionality already implemented in the stack, developers can focus on their application development and not worry about the lower-level radio and network details.

The Flex SDK includes a number of Connect example applications including the following.

Connect SoC - Empty: A minimal Connect project structure, used as a starting point for custom applications.

Connect SoC - Direct mode Device: Demonstrates direct communication between nodes in range. The network parameters are commissioned by the application.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Connect SoC - MAC Mode Device: Demonstrates direct MAC mode communication between nodes in range.

Connect SoC - Sensor and **Connect SoC - Sink**: The sensor and sink applications demonstrate how to set up a star network topology in which communication occurs in both directions between the sink and the sensor(s) nodes.

Connect – SoC ECDH Key Exchange: Illustrates how to share the network key between multiple devices in a secure way (using Elliptic-curve Diffie-Hellman (ECDH) key agreement protocol).

Connect – SoC Switch: In combination with the **Connect Bluetooth DMP – SoC Light** example, demonstrates a simple wireless communication between two or more boards using the Connect SDK.

Connect Bluetooth DMP – SoC Empty: An RTOS-based project that provides a skeleton for Connect, with the only function being a Bluetooth LE Task with a basic CLI interface.

Connect Bluetooth DMP – SoC Light: In combination with the **Connect – SoC Switch** example, demonstrates a simple wireless communication between two or more boards using the Connect SDK.

Connect – NCP: A Co-Processor Communication secondary implementation of the Connect API. It can be used with the CPC daemon and the Connect host library and example applications. All can be found on github. See the readme of the application or *UG435.08 Network Co-Processor Applications with Silicon Labs Connect v3.x* for further info.

Silicon Labs RAIL

Silicon Labs RAIL provides an intuitive, versatile radio interface layer designed to support proprietary or standards-based wireless protocols. RAIL is delivered as a library that you can link to your applications. A description of library functions is provided in the development environment. The RAIL API is documented in an online API reference available through Simplicity Studio and online at https://docs.silabs.com/.

RAIL:

- Implements commonly-used radio functionality, so that it does not have to be written in the application or stack.
- Eliminates the need for developers to become expert in RF register details of complex wireless SoCs.
- Simplifies code migration to new wireless ICs and the development of new stacks by providing a common radio interface layer.
- Allows lab evaluation in addition to application code development.

The RAIL library supports APIs for:

- General Radio Operation
- Channel definition and selection
- Output power configuration
- Transmit
- Clear Channel Assessment before Transmit
- Scheduled Transmit
- Energy Detection
- Receive
- Packet Filtering
- Calibration
- CW (Carrier Wave) Transmission
- Modulated Transmission
- RFSense configuration as wake source

The Flex SDK includes example RAIL application code to demonstrate the capabilities of the device and the RAIL library. These examples are provided as source code to offer a starting point for application development. The following examples, among others, are included in the current release.

RAIL – SoC RAILtest

RAILtest is a general test tool for the RAIL library. RAILtest is developed by the core engineering team working on the RAIL library. As each RAIL library feature is implemented, a RAILtest serial command is added to allow scripted testing and ad hoc experimentation. Many of the RAILtest serial commands can be used for lab evaluation.

RAILtest includes commands to:

Copyright © 2025 Silicon Laboratories. All rights reserved.

Transmit and receive packets.
- Schedule transmits at a specific time in the RAIL timebase.
- Configure RAIL address filtering to receive only specific packets.
- Enable CCA mechanisms (CSMA/LBT) to validate that a channel is clear before transmit.
- Set a timer callback in the RAIL timebase to see how the RAIL timer API works.
- Change the transmit channel within the current configuration's band.
- Change the transmit power level.
- Enable RF energy sensing of specified duration across the 2.4 GHz and/or Sub-GHz bands, and sleep to wake on this event.
- Output a continuous unmodulated tone for debugging.
- Output a continuous modulated PN9 stream for debugging.
- Enter into direct mode where data can be sent and received using asynchronous GPIOs as input and output.

RAIL – SoC Range Test

The Range Test examples enable over-the-air range testing between two devices customized with user-defined parameters. Range Test is designed to be run on Silicon Labs hardware without the need for commands from a host computer. This capability allows for mobility during range testing activities.

RAIL – SoC Range Test BLE and IEEEE80215.4

The Range Test examples enable over-the-air range testing between two devices customized with user-defined parameters. Five predefined PHYs can be used: BLE: 125kbps, BLE: 500kbps, BLE: 1Mbps, BLE: 2Mbps, IEEE80215.4: 250kbps.

RAIL – SoC Switch: Demonstrates the simplest exchange of transmit and receive operation between a light and a switch.

RAIL – SoC Light: Demonstrates the simplest exchange of transmit and receive operation between a RAIL light and a RAIL switch. The light is capable of periodically and on change reporting its status back to the switch. This is not the dynamic multiprotocol light example application.

RAIL – SoC Mode Switch: Demonstrates an exchange of transmit and receive operations between two (or more) nodes expanded by the Wi-SUN Mode Switching features implemented in RAIL.

RAIL – SoC Simple TRX Multi-PHY: Demonstrates the use of multiple PHYs selectable by channels. By default, channel 0 is configured to 2.4 GHz, 250 kbps, and channel 1 is configured to 915 MHz, 500 kbps (both packets are receivable by a single-PHY application using the correct pre-configured PHY). For details see AN971: EFR32 Radio Configurator Guide.

RAIL – SoC Simple TRX: Demonstrates the simplest transmit and receive functions based on RAIL.

RAIL – SoC Simple TRX Standards: Demonstrates the simplest transmit and receive functions based on RAIL based on IEEE Std. 802.15.4.

RAIL – SoC Simple TRX with Auto-ACK: Demonstrates the simplest exchange transmit and ack operation between two nodes, based on RAIL.

RAIL – NCP Simple TRX with CPC Support: Same as *RAIL – SoC Simple TRX*, but implements a Co-Processor Communication endpoint, so the device can be controlled through CPCd from a Linux computer, using UART interface.

RAIL – NCP Simple TRX with CPC Support (SPI): Same as *RAIL – SoC Simple TRX with CPC Support*, but uses SPI interface to communicate with CPCd.

RAIL – SoC Long Preamble Duty Cycle: Demonstrates how Rx duty cycling can be implemented using a long preamble.

RAIL – SoC Burst Duty Cycle: Demonstrates how Rx duty cycle can be implemented using repeated transmissions

RAIL – SoC Energy Mode: Demonstrates the low power modes (EM0-Active, EM1-Sleep, EM2-Deep Sleep). of the EFR32.

RAIL – SoC Empty: A minimal RAIL project structure, used as a starting point for custom applications.

RAIL Bluetooth DMP – SoC Range Test: Range Test with Bluetooth connectivity. It runs on top of Micrium OS RTOS and multiprotocol RAIL.

Copyright © 2025 Silicon Laboratories. All rights reserved.

RAIL Bluetooth DMP – SoC Range Test BLE and IEEE802.15.4: Range Test BLE and IEEE802.15.4 with Bluetooth connectivity. It runs on top of Micrium OS RTOS and multiprotocol RAIL.

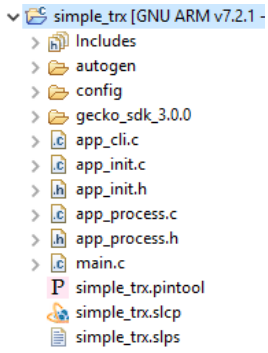
RAIL - SoC Wireless M-bus Meter: Implements a Wireless M-Bus meter application. For details, see *AN1119: Using RAIL for Wireless M-Bus Applications with EFR32*. Uses the multi-PHY configurator and is capable of limited multi-PHY features, like asymmetric bidirectional modes. For details see *AN1253: EFR32 Radio Configurator Guide for Simplicity Studio 5*.

RAIL - SoC Wireless M-bus Collector: Implements a Wireless M-Bus collector application. For details, see *AN1119: Using RAIL for Wireless M-Bus Applications with EFR32*. Uses the multi-PHY configurator. For details see *AN1253: EFR32 Radio Configurator Guide for Simplicity Studio 5*.

Project Structure

Projects always include the following parts:

- **autogen** folder: Only the autogen folder includes generated code. It includes the PHY configuration (*rail_config.c*), init code, the linker script, and other generated code used by components, like the command descriptors for the CLI interface.
- **config** folder: Component configuration headers are located in this folder. These can be edited with the Simplicity IDE Component Editor, but directly editing the header file is also possible. The Component Editor is available through the Project Configurator's **Configure** control, available only for configurable components.
- **gecko_sdk** folder (with version number): Contains source and binary files added by components.
- **files in the root folder**: Only the application specific files should be in the root folder, including source files, the project configurator (*.slcp*) file and the Pin Tool (*.pintool*) file.



All projects include *main.c*, which is not recommended to modify. Instead, add the initialization code to *app_init.c*, and implement the main loop in *app_process.c*. This way, the System components can initialize components, and call the "process" function of the components that require it. Additionally, enabling an RTOS will convert *app_process*'s main loop into an RTOS task.

These files are also available in Connect projects, and can be used for example for init and process tasks that are independent of the stack running or not. However, it is still recommended to use emberAfInitCallback() and emberAfTickCallback(), so the Connect stack could prioritize urgent stack operations above the application.

The Gecko Bootloader

A bootloader is a program stored in reserved flash memory that can initialize a device, update firmware images, and possibly perform some integrity checks. Silicon Labs networking devices use bootloaders that perform firmware updates in two different modes: standalone (also called standalone bootloaders) and application (also called application bootloaders). An application bootloader performs a firmware image update by reprogramming the flash with an update image stored in internal or external memory. For more information about bootloaders see *UG103.6: Bootloader Fundamentals*.

The Gecko Bootloader is a code library configurable through Simplicity Studio's IDE to generate bootloaders that can be used with a variety of Silicon Labs protocol stacks. The bootloaders work with specialized firmware update image formats. The Gecko Bootloader update images has the extension .gbl.

Examples provided for the EFR32 parts include Silicon Labs Gecko Bootloader examples. Examples are provided for all compatible Simplicity Studio SDKs. For more information on using the Gecko Bootloader see *UG266: Silicon Labs Gecko*

Copyright © 2025 Silicon Laboratories. All rights reserved.

Bootloader User's Guide in GSDK 3.2 and Lower and Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher.

> **Note**: When working with the Gecko Bootloader, you must use Simplicity Commander to enable some configuration options such as security features.

Gecko Platform

The Gecko Platform is a set of drivers and other lower layer features that interact directly with Silicon Labs chips and modules. Gecko Platform components include EMLIB, EMDRV, RAIL Library, NVM3, and mbedTLS. For more information about Gecko Platform, see release notes that can be found in Simplicity Studio's Documentation tab.

Prerequisites

Before following the procedures in this guide you must have:

- Purchased one of the Wireless Gecko (EFR32) Portfolio Wireless Kits.
- Installed the Flex SDK. The Flex SDK is provided as part of the Gecko SDK (GSDK), the suite of Silicon Labs SDKs. To quickly get started with the GSDK, install Simplicity Studio 5, which will set up your development environment and walk you through GSDK installation. Simplicity Studio 5 includes everything needed for IoT product development with Silicon Labs devices, including a resource and project launcher, software configuration tools, full IDE with GNU toolchain, and analysis tools. Installation instructions are provided in the online Simplicity Studio 5 User's Guide. Alternatively, Gecko SDK may be installed manually by downloading or cloning the latest from GitHub. See https://github.com/SiliconLabs/gecko_sdk for more information.

See the Flex SDK v3.x release notes for version restrictions and compatibility constraints for Connect and RAIL and these components.

Simplicity Commander is installed along with Simplicity Studio. An application with limited functionality can be accessed through Simplicity Studio's Tools menu. Most functions are accessible through a CLI invoked by opening a command prompt in the Simplicity Commander directory. See *UG162: Simplicity Commander Reference Guide* for more information.

- (optional) Installed IAR Embedded Workbench for ARM (IAR EWARM). See the Release Notes for the IAR version supported by this version of the Flex SDK. This can be used as a compiler in the Simplicity Studio development environment as an alternative to GCC (The GNU Compiler Collection), which is provided with Simplicity Studio. Download the supported version from the Silicon Labs Support Portal, as described below. Refer to the "QuickStart Installation Information" section of the IAR installer for additional information about the installation process and how to configure your license.

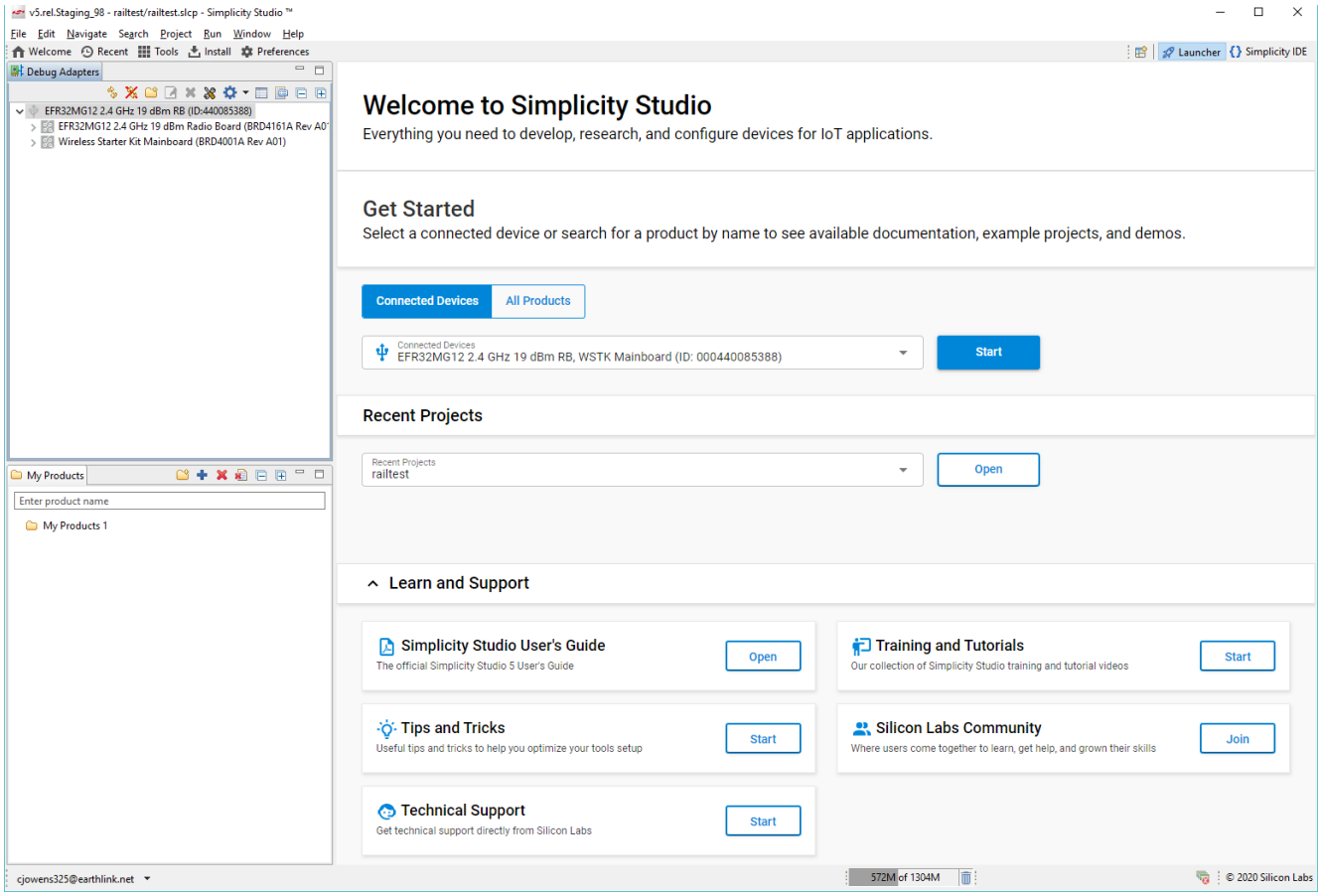
To get a 30-day evaluation license for IAR:

- Go to the Silicon Labs support portal at https://www.silabs.com/support.
- Scroll down to the bottom of the page, and click **Contact Support**
- If you are not already signed in, sign in.
- Click the Software Releases tab. In the View list select **Development Tools**. Click **Go**. In the results is a link to the IAR-EWARM version named in the release notes.
- Download the IAR package (takes approximately 1 hour).
- Install IAR.
- In the IAR License Wizard, click **Register with IAR Systems to get an evaluation license**.
- Complete the registration and IAR will provide a 30-day evaluation license.
- Once IAR-EWARM is installed, the next time Simplicity Studio starts it will automatically detect and configure the IDE to use IAR-EWARM.

Technical Support

You can access the Silicon Labs support portal at https://www.silabs.com/support through Simplicity Studio 5's Welcome view under Learn and Support. Use the support portal to contact Customer Support for any technical questions you might have during the development process.

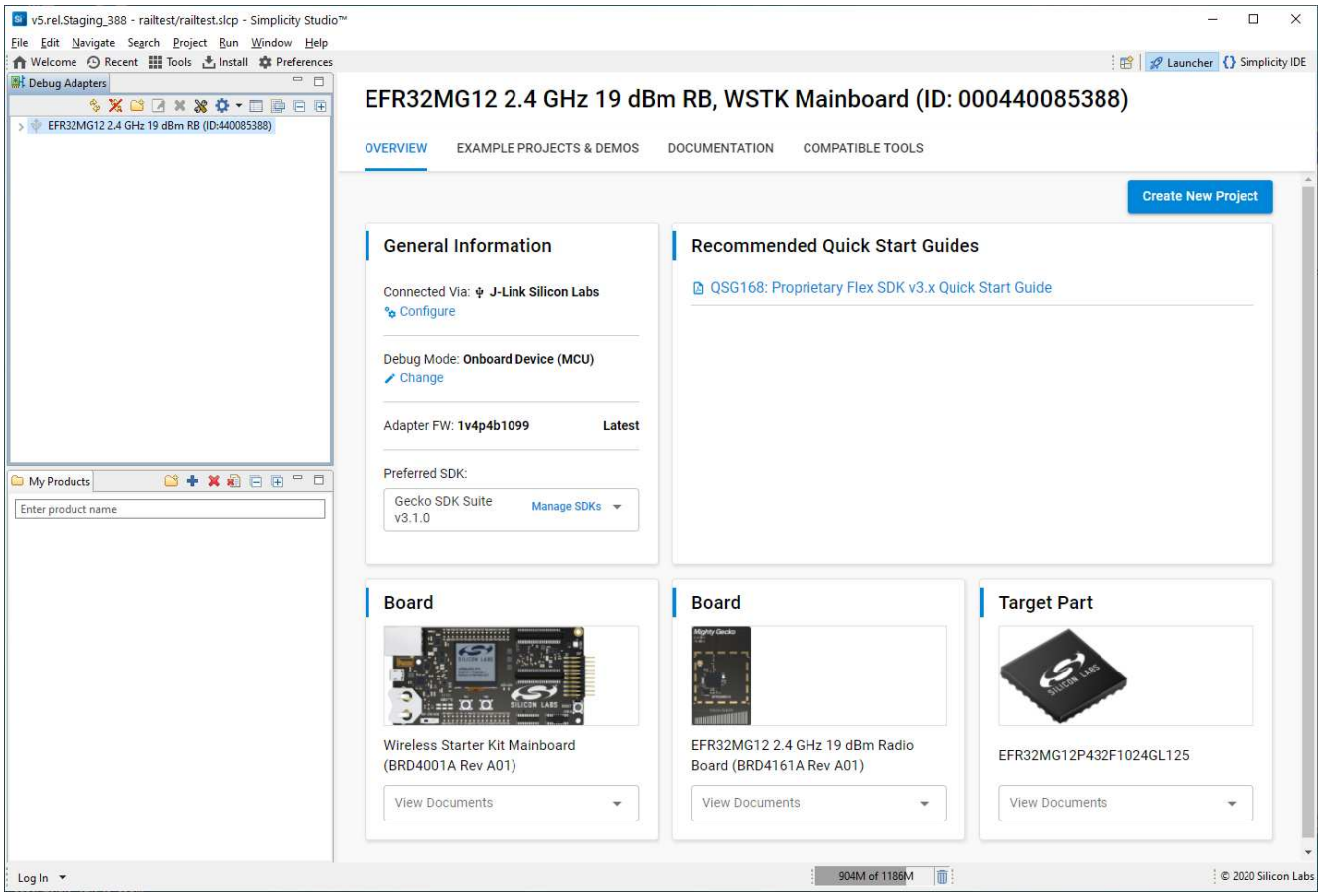
Copyright © 2025 Silicon Laboratories. All rights reserved.



Documentation

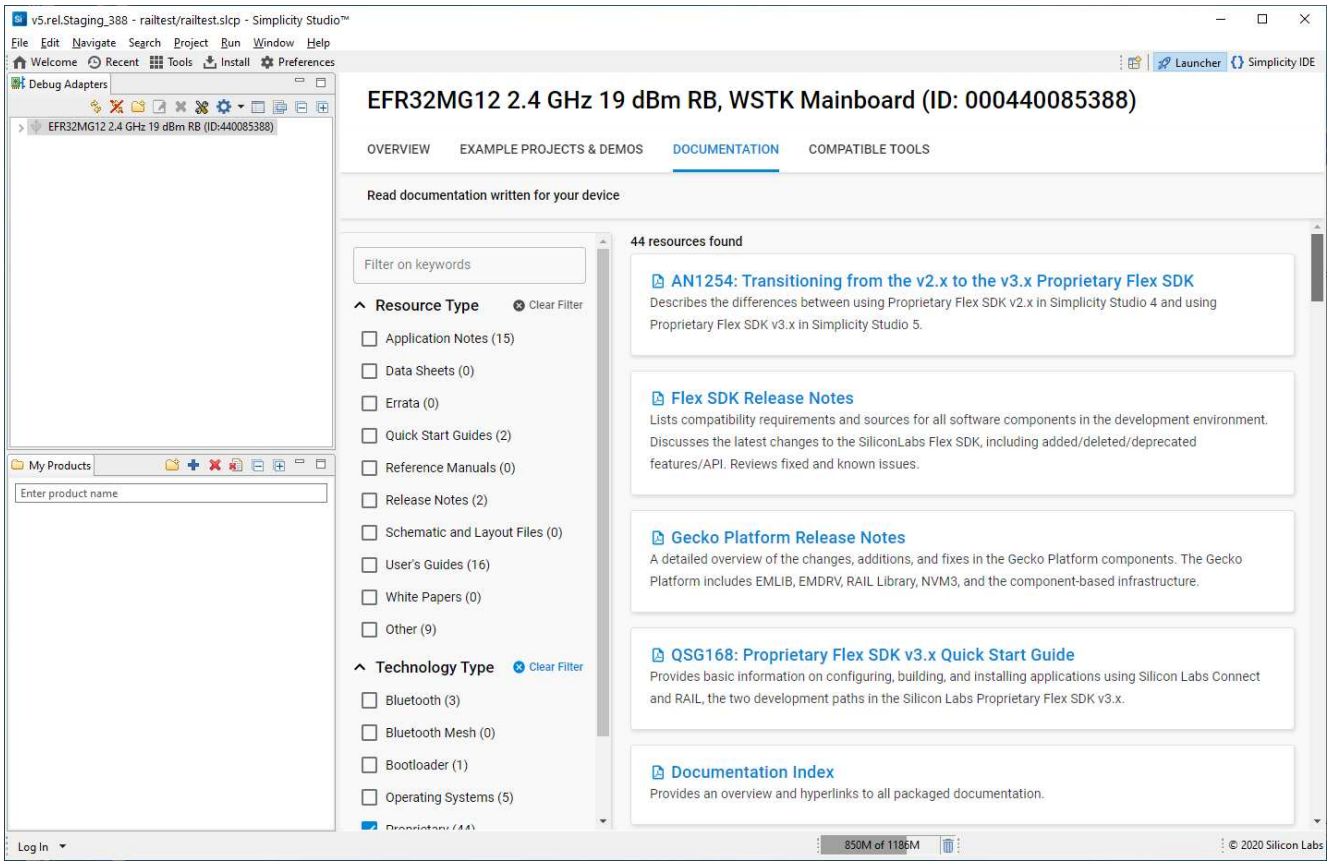
Hardware-specific documentation may be accessed through links on the part OVERVIEW tab in Simplicity Studio 5.

Copyright © 2025 Silicon Laboratories. All rights reserved.



SDK documentation and other references are available through the Documentation tab.

Copyright © 2025 Silicon Laboratories. All rights reserved.



Copyright © 2025 Silicon Laboratories. All rights reserved.

Getting Started with Simplicity Studio 5 and the Proprietary Flex SDK

Getting Started with Simplicity Studio 5 and the Proprietary Flex SDK

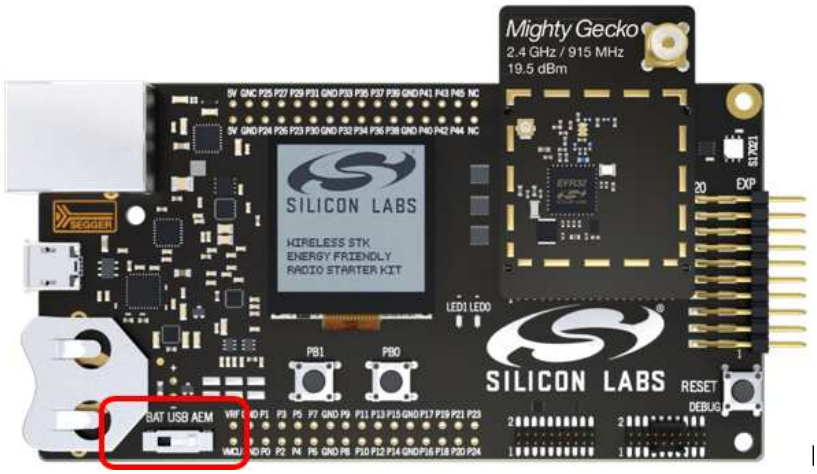
Simplicity Studio 5 (SSv5) is a free Eclipse-based Integrated Development Environment (IDE) and a collection of value-add tools provided by Silicon Labs. Developers can use SSv5 to develop, debug and analyze their Flex and other Silicon Labs SDK-based applications. Its main goal is to reduce development time so that you can focus on your application instead of researching the hardware reference manuals.

The online Simplicity Studio 5 User's Guide provides both a summary overview of SSv5 functions and how to use it with Flex SDK, as well as detailed references to the various SSv5 tools and perspectives.

This guide assumes that you have downloaded SSv5 and the Gecko SDK, which includes the Flex SDK, and are familiar with the features of the SSv5 Launcher perspective.

You should have your mainboard connected.

> **Note**: For best performance in Simplicity Studio 5, be sure that the power switch on your mainboard is in the Advanced Energy Monitoring or "AEM" position, as shown in the following figure.



Copyright © 2025 Silicon Laboratories. All rights reserved.

Working With Example Applications

Working with Example Applications

This section offers instructions on working with RAIL and Connect SoC examples.

Simplicity Studio offers a variety of ways to begin a project using an example application. The Simplicity Studio 5 User's Guide describes them all. This guide uses the File > New > Silicon Labs Project Wizard method, because it takes you through all three of the Project Creation Dialogs.

Regardless of how you begin, when working with example applications in Simplicity Studio, you will execute the following steps:

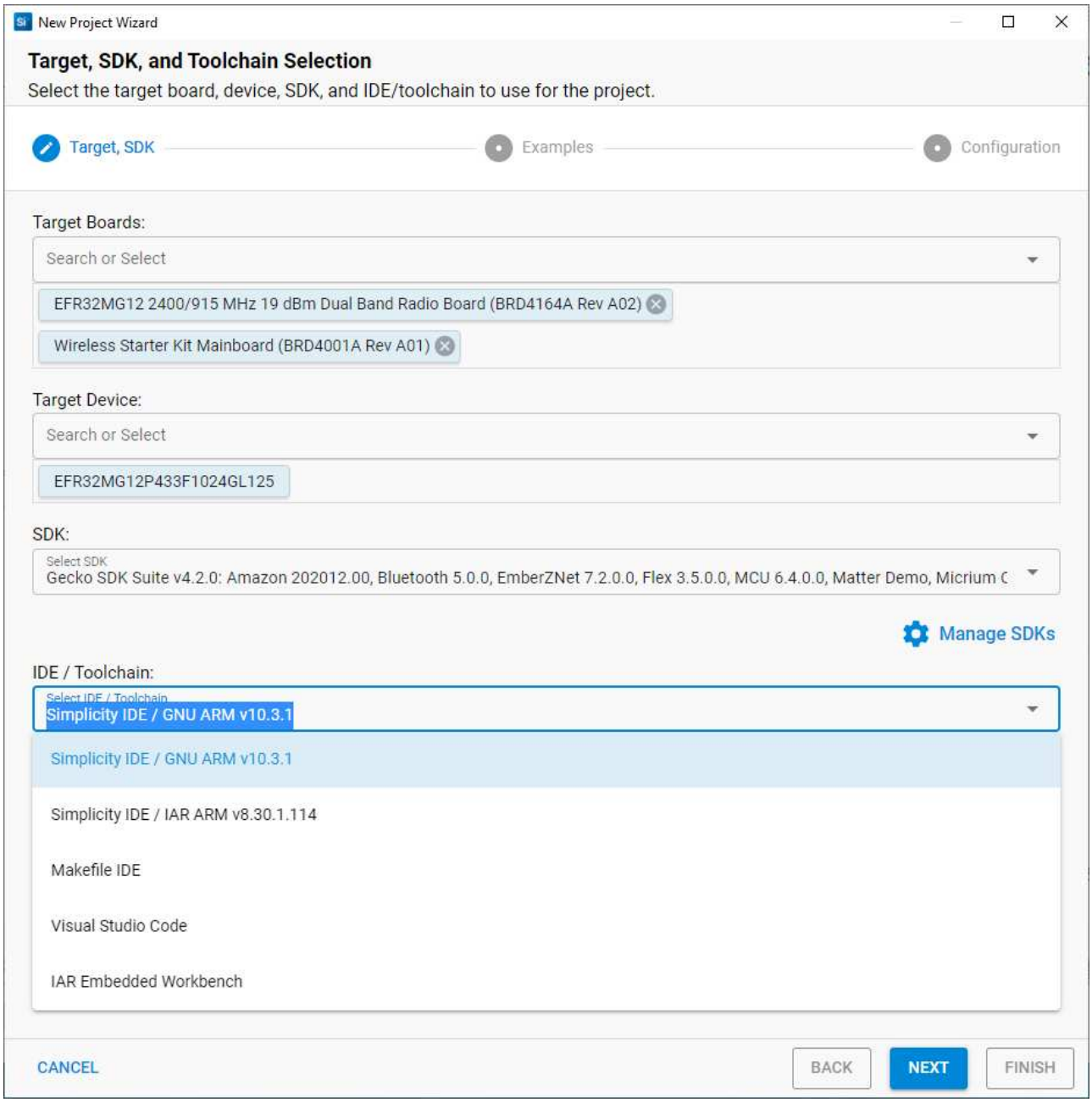
- Select an example application and create a project
- Modify and generate radio configuration.
- Modify code if necessary
- Compile and flash the application to the radio board.

These steps are described in the following sections. Note: Your SDK version may be later than the version shown in the procedure illustrations.

Creating a Project Based on an Example

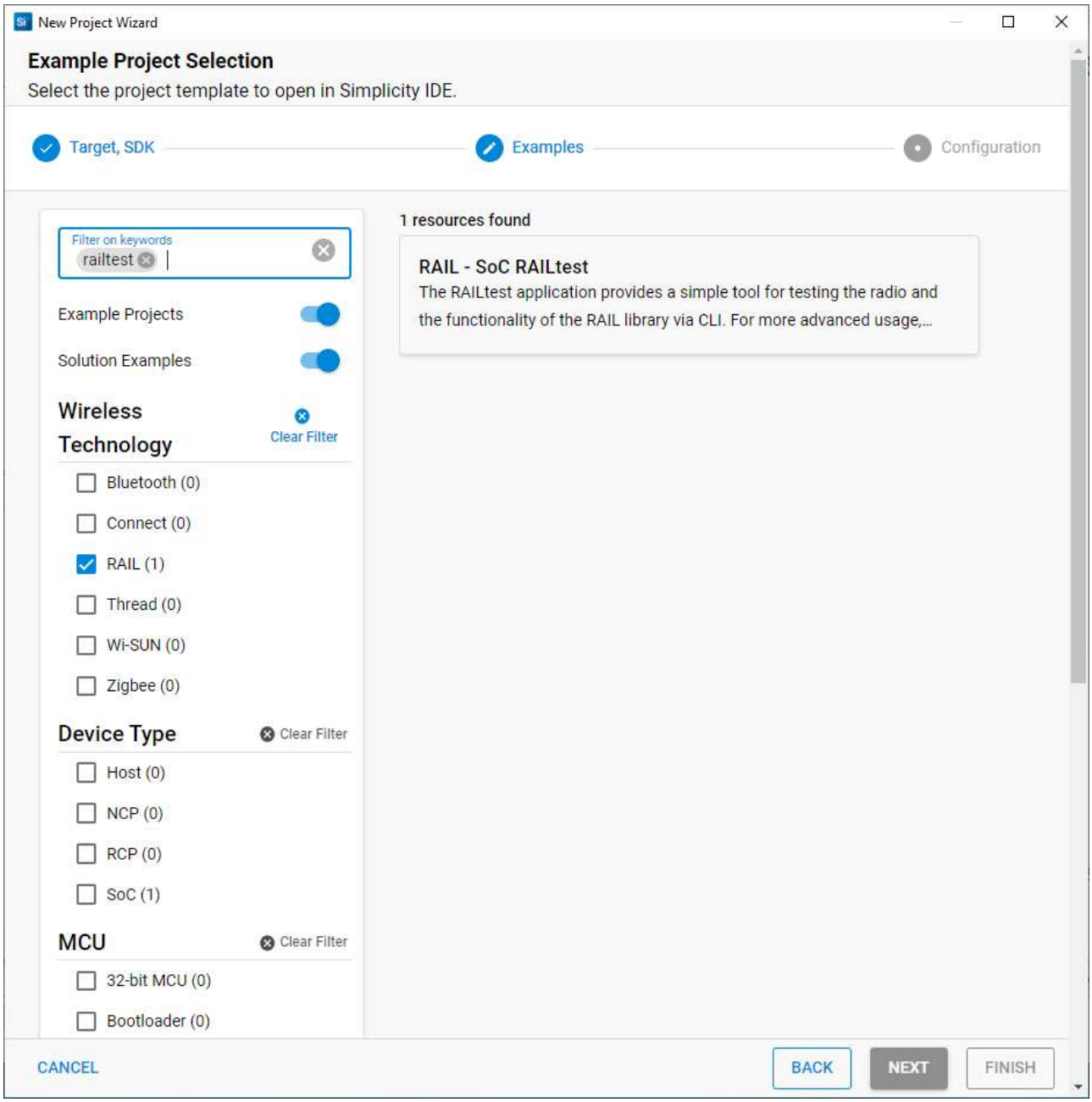
1. Open SSv5's File menu and select **New > Silicon Labs Project Wizard**. The Target, SDK, and Toolchain Selection dialog opens. If you want to change the toolchain from the default GCC to IAR, do so here. Click **NEXT**.

Copyright © 2025 Silicon Laboratories. All rights reserved.



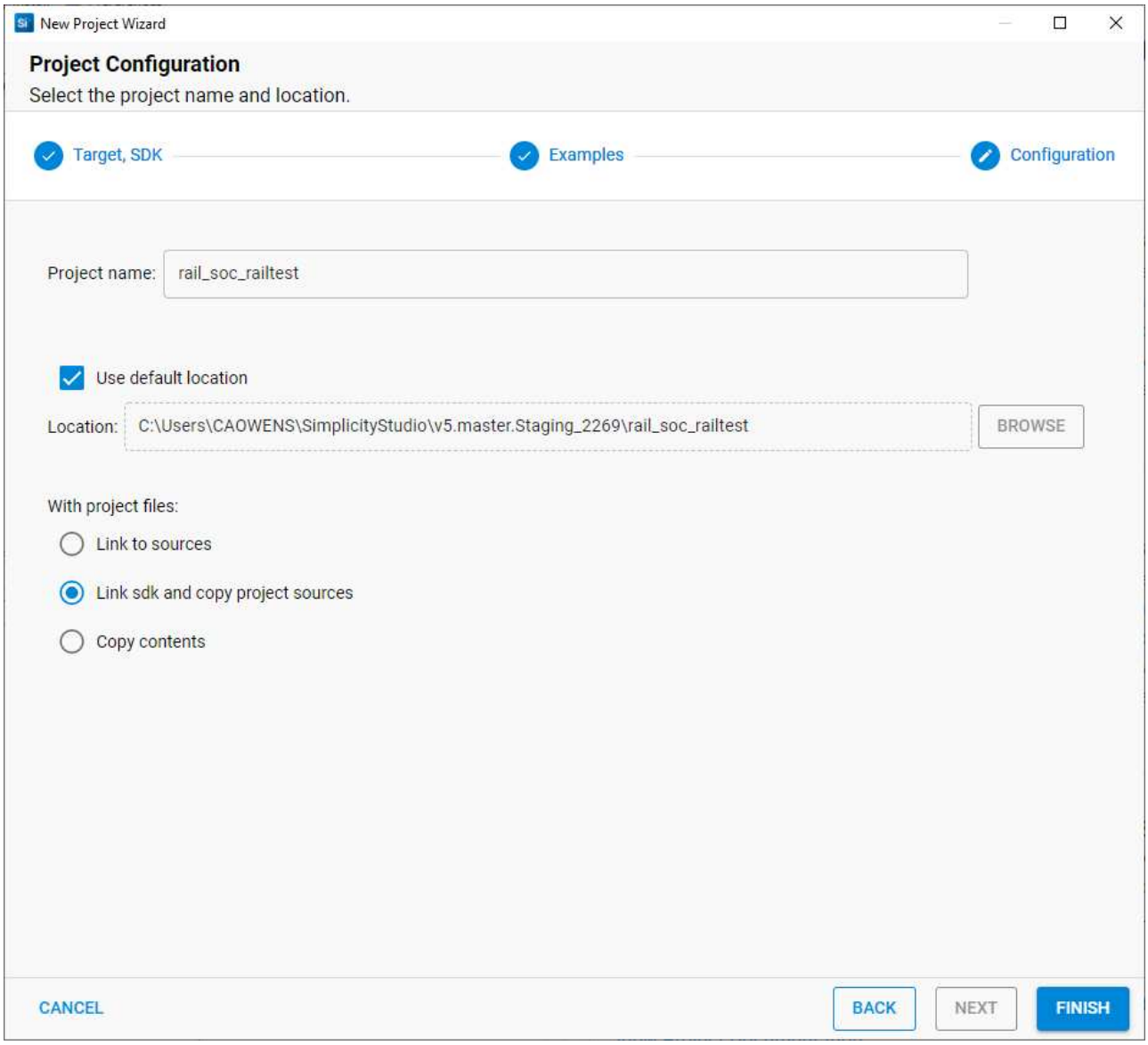
2. The Example Project Selection dialog opens. A variety of filters are available to help you find a specific example, such as 'railt' and Technology Type: Proprietary in the following figure. Select the example, in this case **RAIL – SoC RAILtest**. Select it and click **NEXT**.

Copyright © 2025 Silicon Laboratories. All rights reserved.



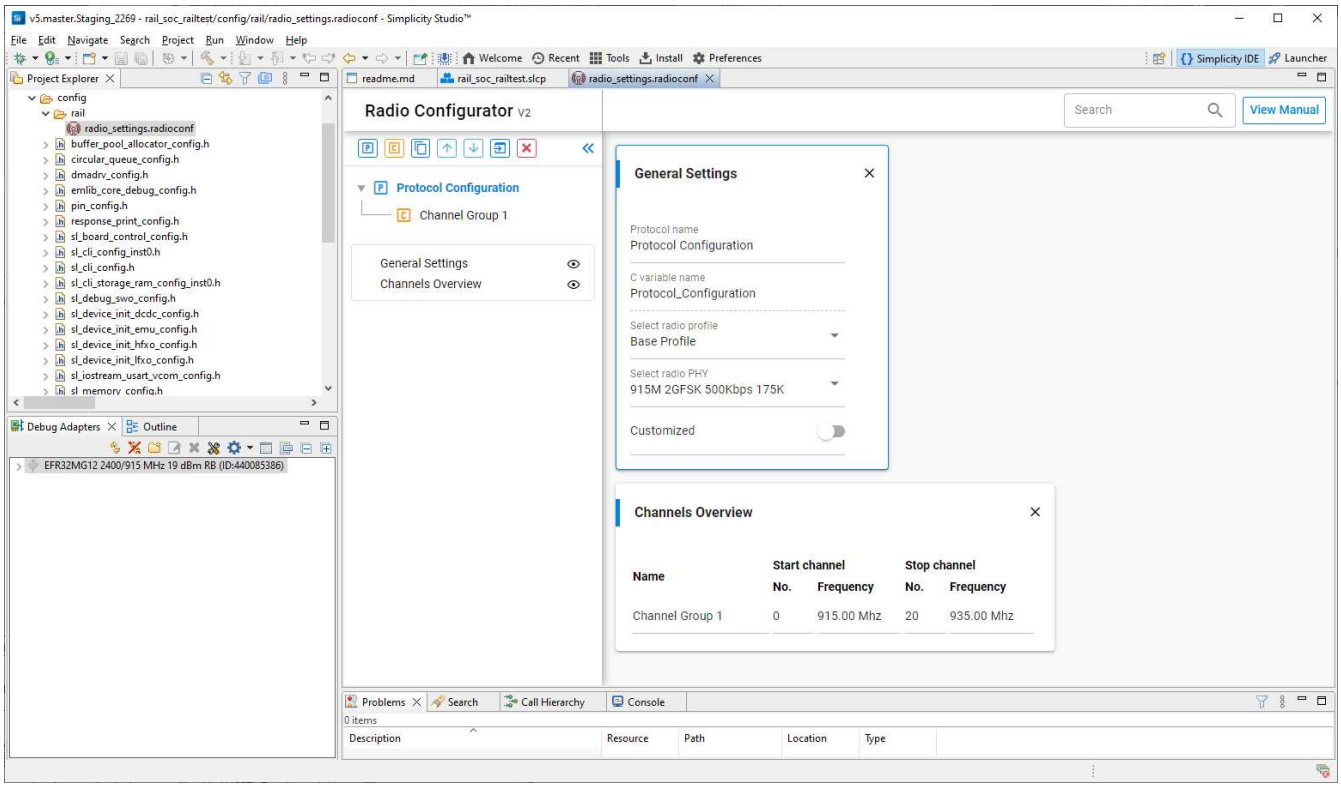
3. The Project Configuration dialog opens. Here you can rename your project, change the default project file location, and determine if you will link to or copy project files. Note that if you change any linked resource, it is changed for any other project that references it. Click **FINISH** and project generation will start.

Copyright © 2025 Silicon Laboratories. All rights reserved.



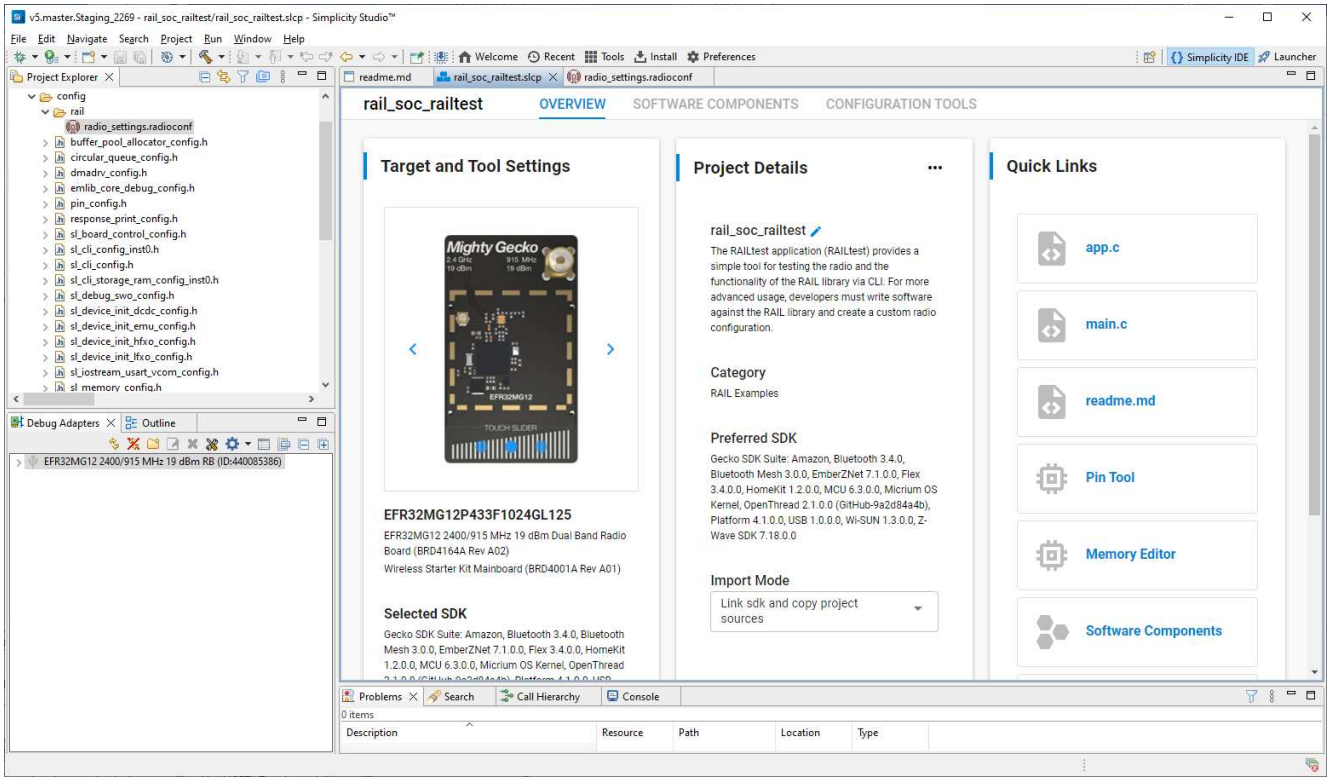
4. The new project opens in the Simplicity IDE Perspective with the Radio Configurator tab (*radio_settings.radioconf*) selected. The Project Configurator tab (*<projectname>.slcp*) and a description of the project (*readme.md*) are also available.

Copyright © 2025 Silicon Laboratories. All rights reserved.



Component Configuration

Click the <projectname>.slcp tab to open the Project Configurator. The default configuration settings will work on the connected development board so, when first experimenting with a development kit, keeping them unchanged and going straight to compiling and flashing is recommended.



Copyright © 2025 Silicon Laboratories. All rights reserved.

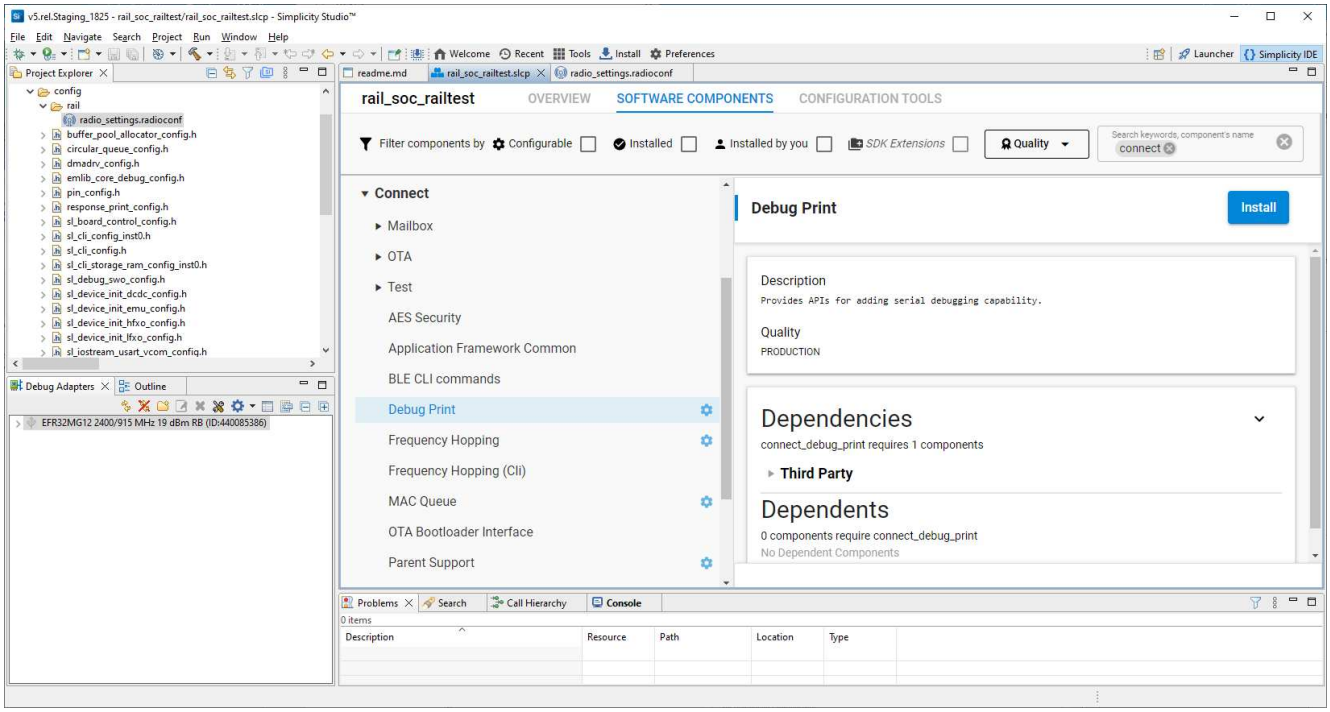
Flex SDK v3.x projects incorporate a Gecko Platform component-based architecture. Software features and functions can be installed and configured through the Component Library. The installation process will:

1. Copy the corresponding SDK files from the SDK folder into the project folder.
2. Copy all the dependencies of the given component into the project folder.
3. Add new include directories to the project settings.
4. Copy the configurations files into the */config* folder.
5. Modify the corresponding auto-generated files to integrate the component into the application.

Additionally, “init” type software components will implement the initialization code for a given component, utilizing their corresponding configuration file as input.

Some software components (like Parent support) will fully integrate into the application to perform a specific task without the need of any additional code, while other components provide an API to be used in the application.

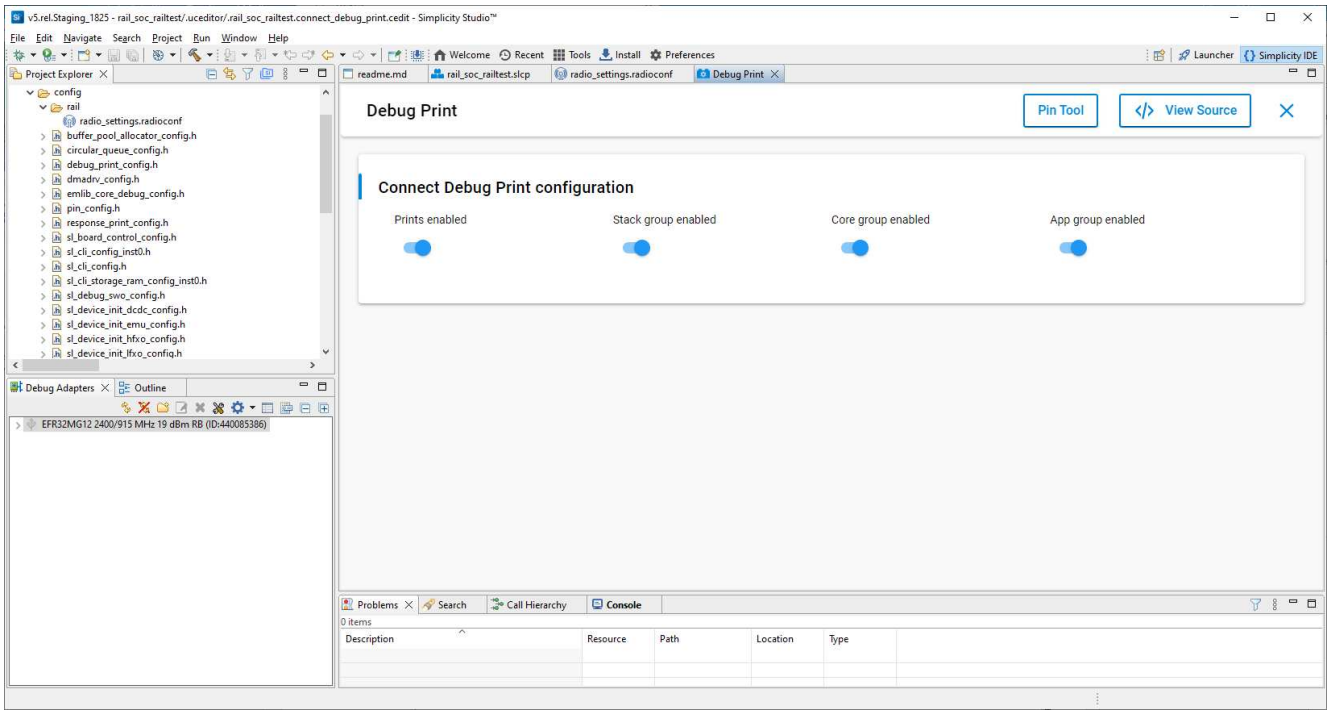
To see the component library, go to the SOFTWARE COMPONENTS tab.



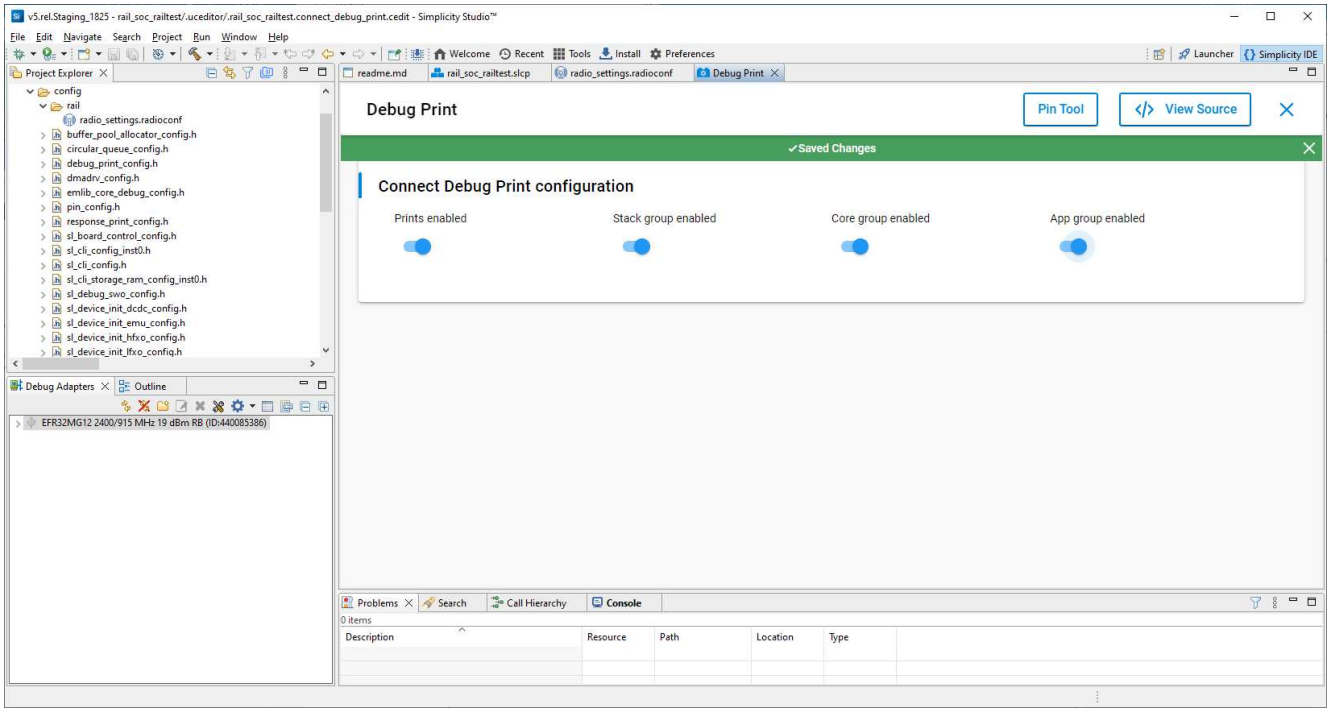
The project is configured by installing and uninstalling components, and configuring installed components. Installed components are checked. Configurable components have a gear symbol. Select a component to see information about it. A number of filters as well as a keyword search are available to help you explore the various component categories. Note that components for all installed SDKs are presented. See the online *Simplicity Studio 5 User’s Guide* for details about the functionality available through the Simplicity IDE perspective and the Project Configurator.

If the component is configurable, click **CONFIGURE** to open the Component Editor.

Copyright © 2025 Silicon Laboratories. All rights reserved.

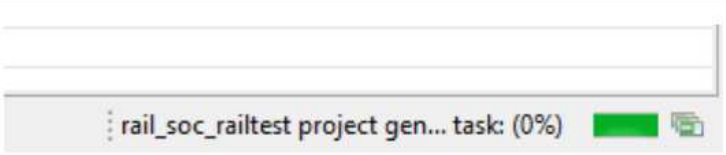


Any changes you make in the Component Editor are autosaved.



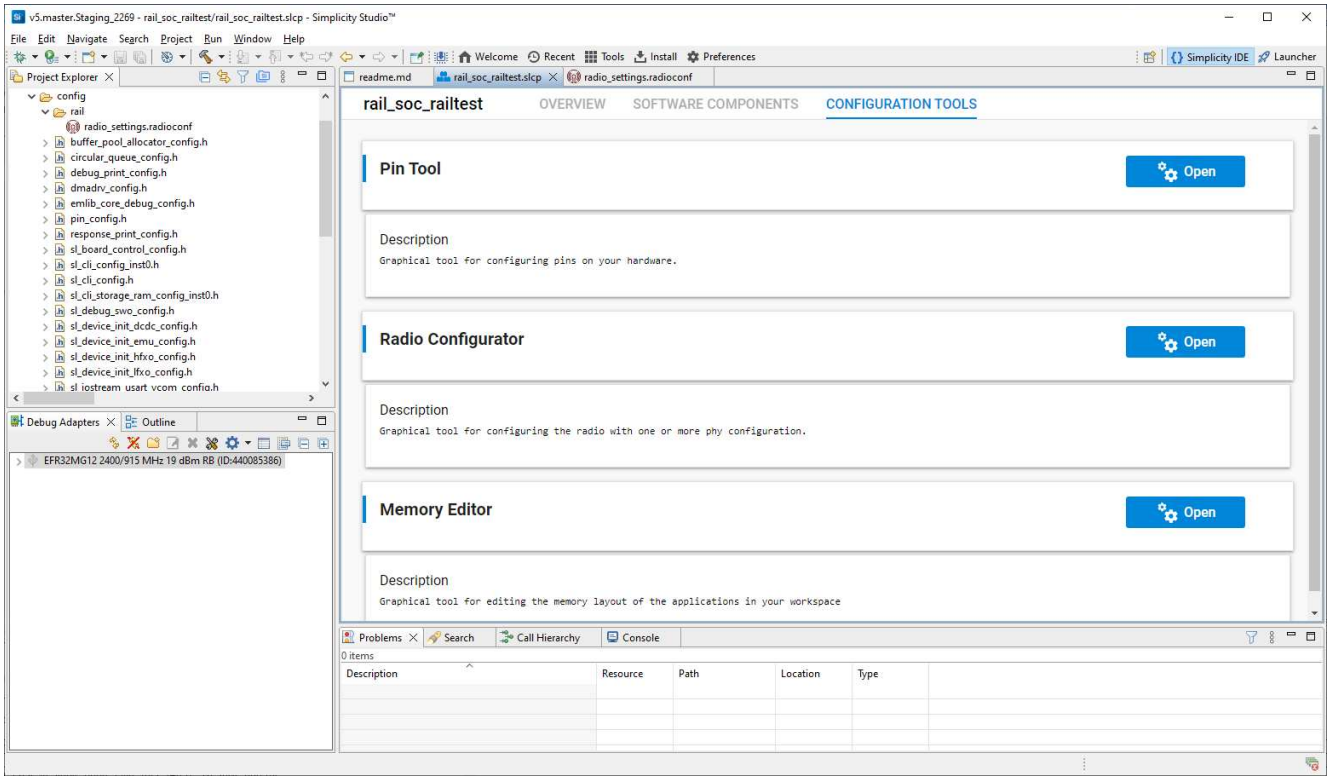
As you install and otherwise make changes in the Project Configurator, project files are autogenerated. Progress is shown in the lower right corner of the Simplicity IDE perspective.

Copyright © 2025 Silicon Laboratories. All rights reserved.



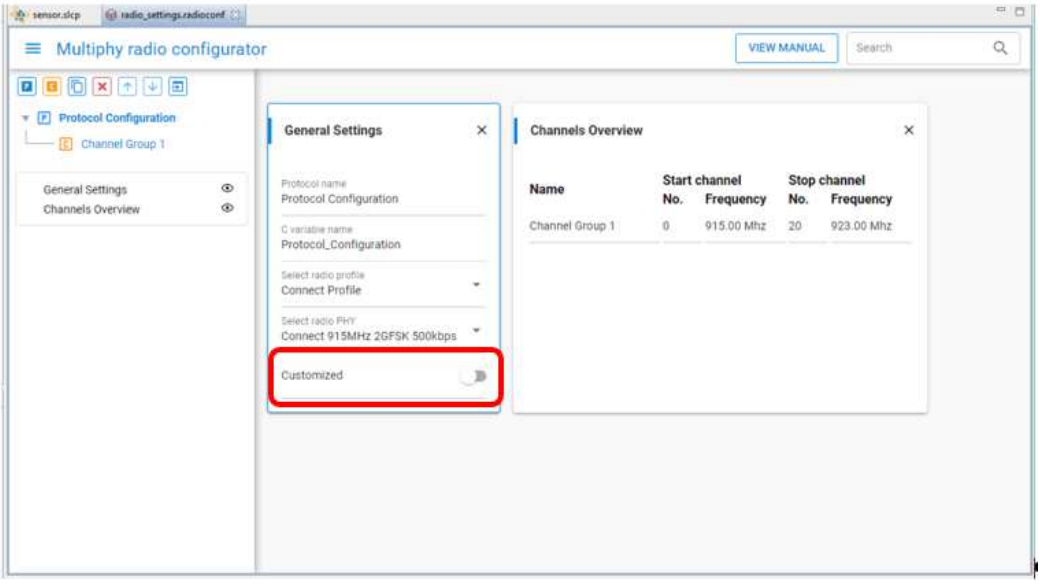
Radio Configuration

If the *radio_settings.radioconf* tab is not present, click the Project Configurator CONFIGURATION TOOLS tab. Click **Open** next to the Radio Configurator. It will open the *radio_settings.radioconf* file of the **/config/rail** folder in the Radio Configurator.



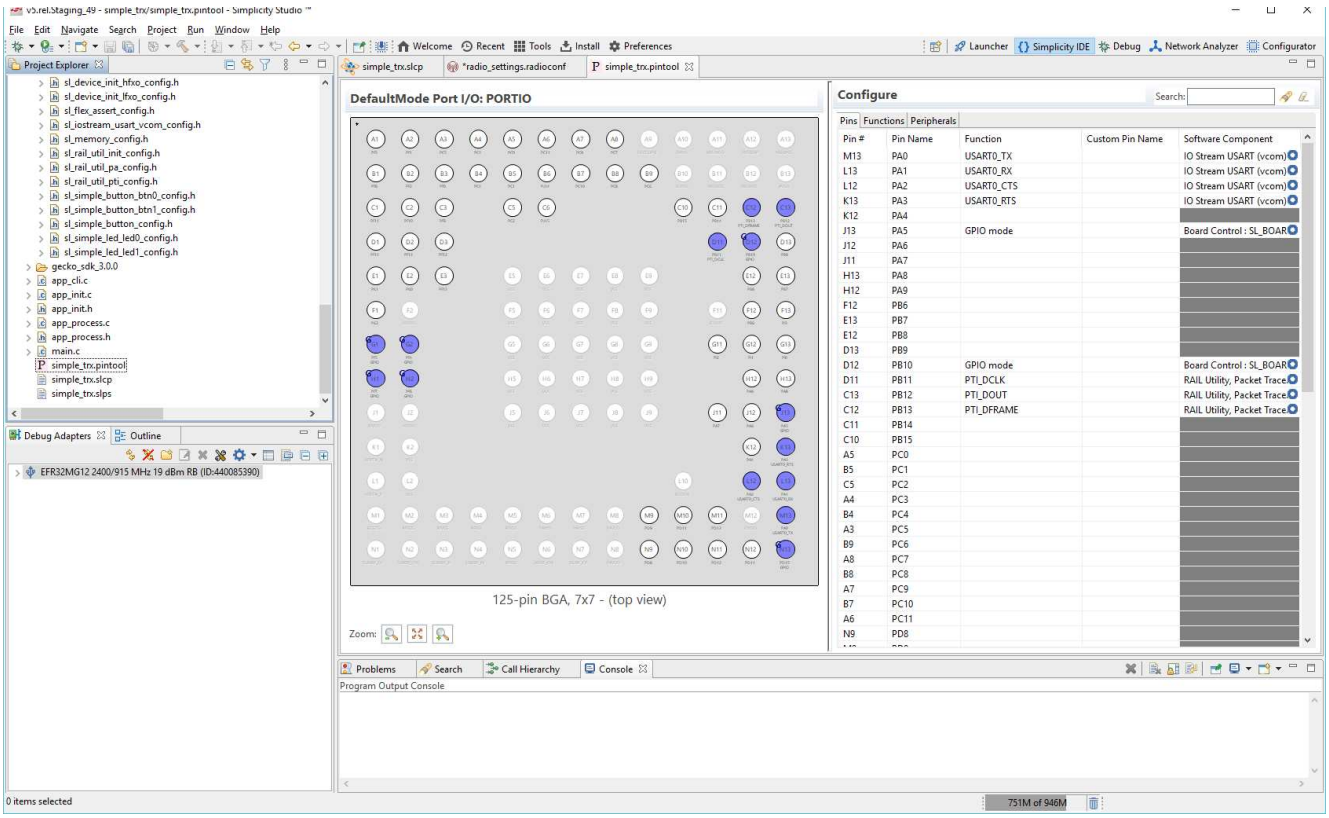
Select a radio profile, and a pre-defined radio PHY of the selected profile. You can then customize the PHY by turning on **Customized**. Changes made in the Radio Configurator are not autosaved. When saving the changes to *radio_settings.radioconf*, the radio configuration will be autogenerated and saved in the */autogen/rail_config.h* and */autogen/rail_config.c* files. See *AN1253: EFR32 Radio Configurator Guide for Simplicity Studio 5* for more information on using the Radio Configurator.

Copyright © 2025 Silicon Laboratories. All rights reserved.



Pin Tool

On the CONFIGURATION TOOLS tab, click **Open** next to Pin Tool. This will open the file *<project>.pintool* of the project in the Pin Tool, which allows you to easily configure new peripherals or change the properties of existing ones. You can also double-click the file *<project>.pintool* in the Project Explorer view to open it.



Double-click a Software Component to open the Component Editor and configure that function. Pin Tool does not autosave.

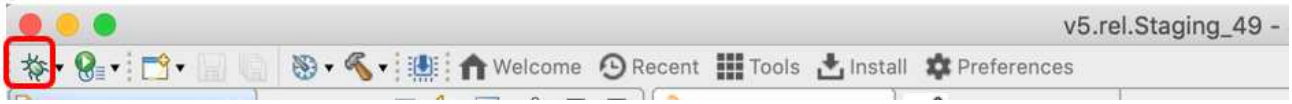
Compiling and Flashing the Application

Copyright © 2025 Silicon Laboratories. All rights reserved.

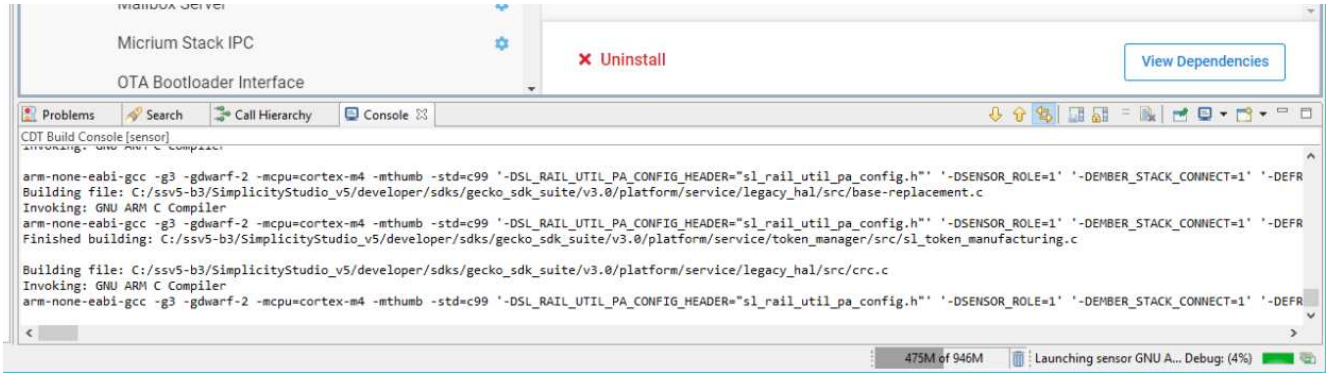
You can either compile and flash the application automatically, or manually compile it and then flash it.

Automatically Compile and Flash

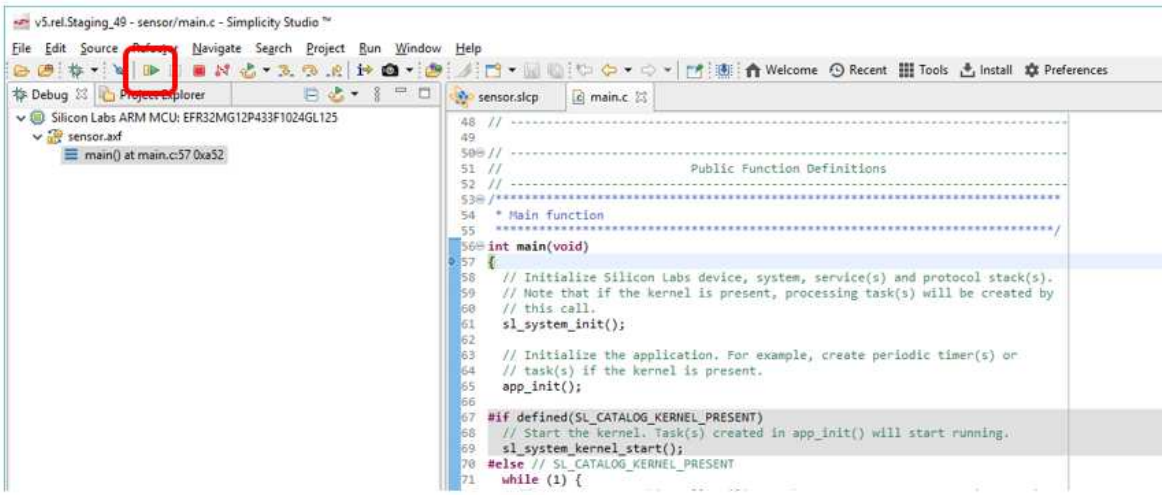
You can automatically compile and flash the application to your connected development hardware in the Simplicity IDE. Click the **Debug** control.



Progress is displayed in the console and a progress bar in the lower right.



When building and flashing are complete, a Debug perspective is displayed. Click the **Resume** control to start the application running on the device.



Next to the **Resume** control are **Suspend**, **Disconnect**, **Reconnect**, and **stepping** controls. Click **Disconnect** when you are ready to exit Debug mode.



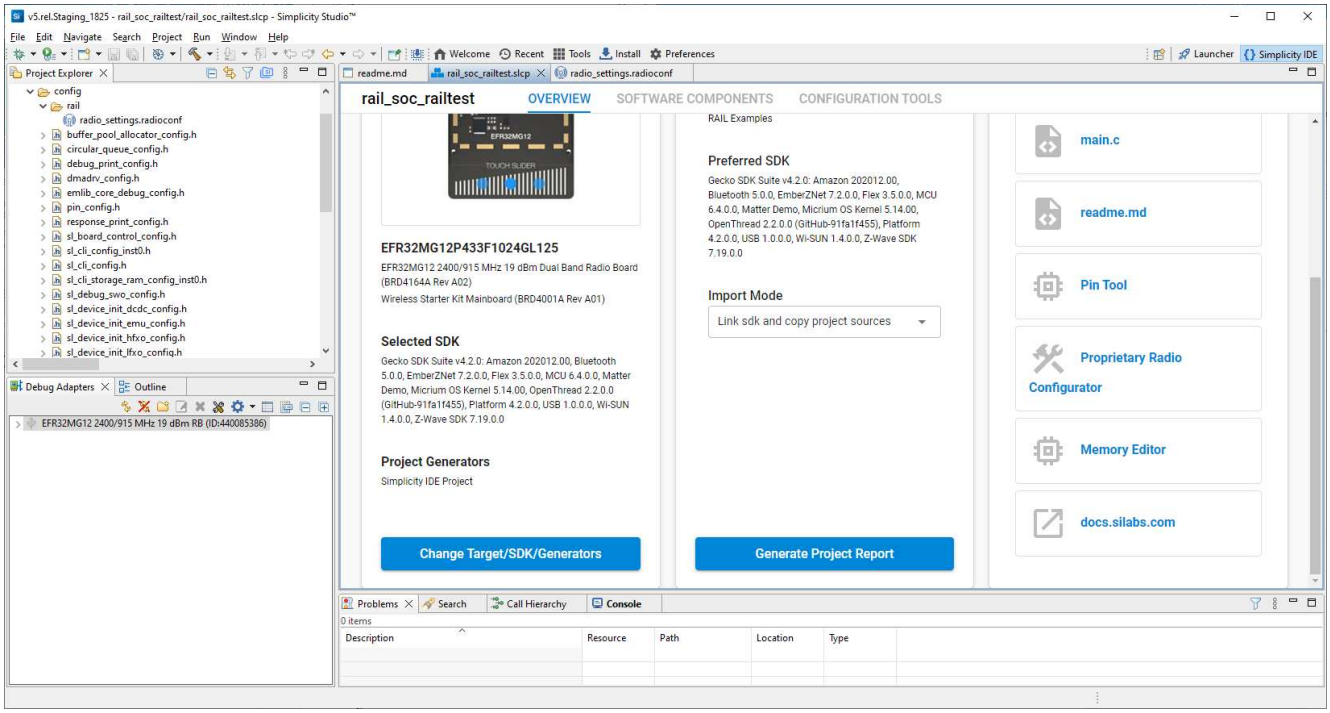
Copyright © 2025 Silicon Laboratories. All rights reserved.

Manually Compile and Flash

After you generate the project files, instead of clicking **Debug** in the Simplicity IDE, click the **Build** control (hammer icon) in the top tool bar.

The sample application will compile based on its build configuration. You can change the build configuration at any time by right-clicking the project directory in Project Explorer view and opening **Build Configurations > Set Active**.

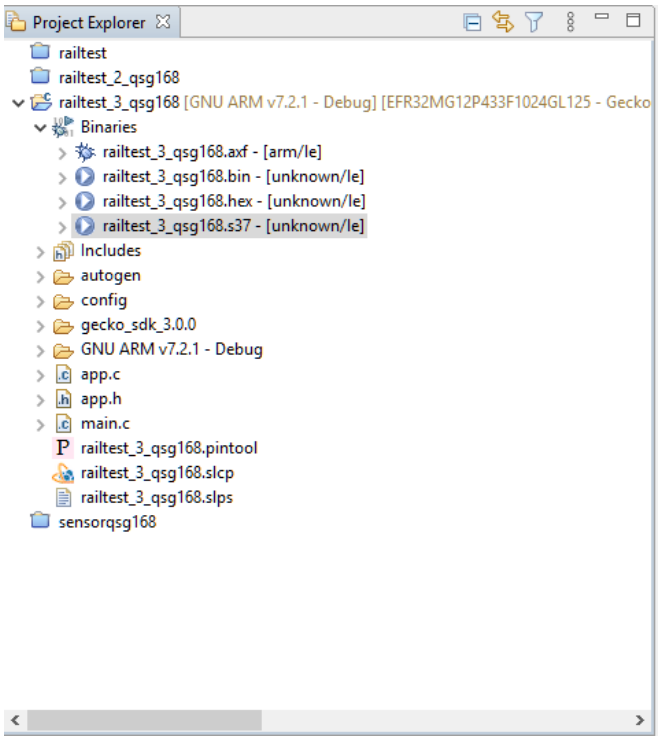
You can also build your application directly in IAR-EWARM by opening IAR-EWARM and opening the generated project file inside IAR. To generate the file for IAR EWARM, go to the OVERVIEW tab of the Project Configurator, scroll down, and click **Change Target/SDK/Generators** on the Target and Tool Settings card, drop down the list under CHANGE PROJECT GENERATORES, and include IAR EMBEDDED WORKBENCH PROJECT. The .eww file is created after you save the change.



1. Open IAR-EWARM.
2. Select **File > Open > Workspace** and navigate to the location you selected for your sample application.
3. Select the application *.eww* file and click **Open**.
4. Select **Project > Make** or press F7. If the application builds without errors, you are ready to install the image on a device.

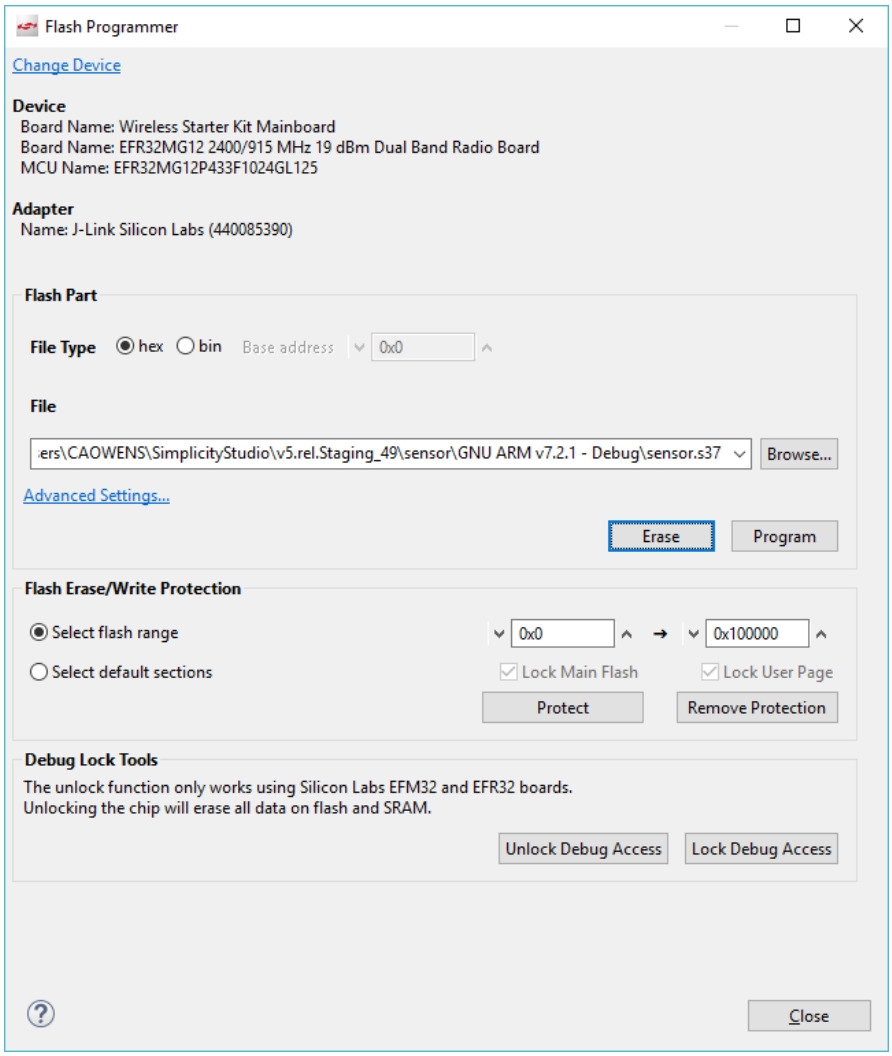
You can load the binary image through Project Explorer view. Locate the <project>.bin, .hex, or .s37 file in the **Binaries** subdirectory.

Copyright © 2025 Silicon Laboratories. All rights reserved.



Right-click the file and select **Flash Programmer**. The Flash Programmer opens with the file path populated. Click **Program.**

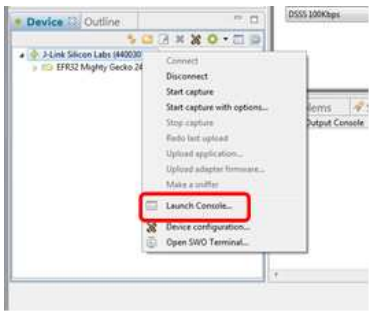
Copyright © 2025 Silicon Laboratories. All rights reserved.



Interacting with Examples

Depending on the example application, you may be able to interact with it through buttons, LEDs and the LCD of the mainboard and through your development environment's Console interface using a CLI (command line interpreter).

To launch the Console interface, in the Simplicity IDE perspective right-click on the debug adapter in the Debug Adapters view. Choose **Launch Console**. Alternatively, from the Tools icon in the Simplicity IDE toolbar, or on the COMPATIBLE TOOLS tab in the Launcher perspective, select **Device Console**. Select the Serial 1 tab and click Enter to get a prompt from the sample app CLI.



Copyright © 2025 Silicon Laboratories. All rights reserved.

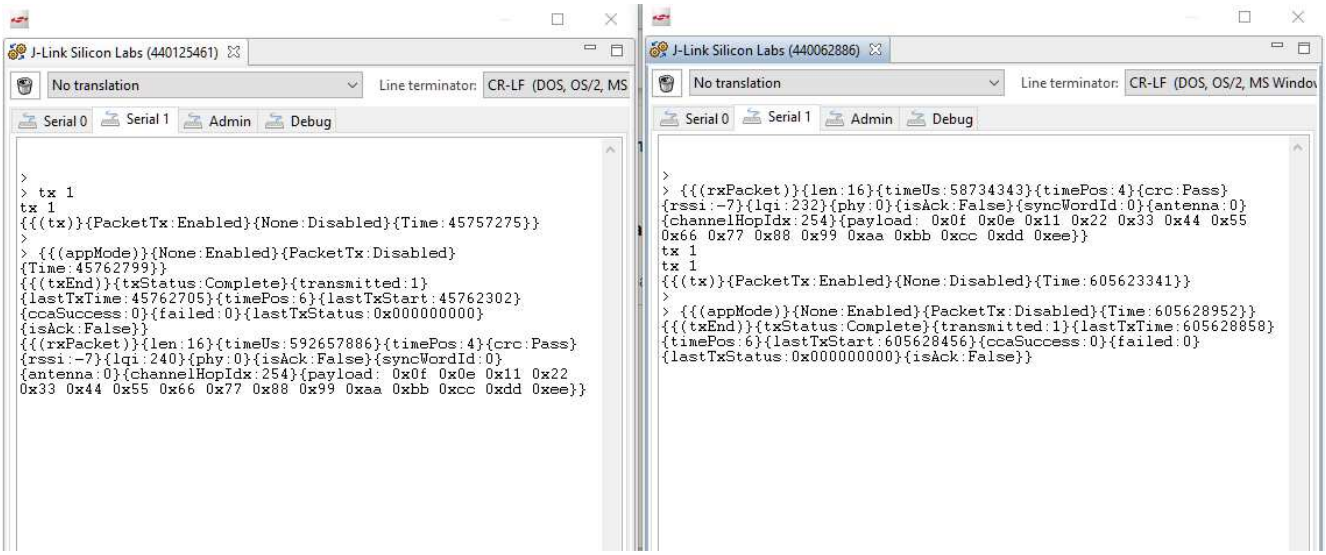
In the RAILtest application the console interface allows testing of any RAIL feature. Enter “help” to get a list of the available CLI commands.

For a simple radio functionality test flash the same RAILtest binary on two radio boards and open a console for each of them. Make sure an antenna is connected to each radio board if applicable. RAILtest starts in RX state on both boards.

Type tx 1 in the first console window. The first radio node will transmit one packet and then return to RX state. A received packet will be reported in the second console window by the second radio node.

Type tx 1 in the second console window. The second radio node will transmit one packet and then return to RX state. A received packet will be reported in the first console window by the first radio node.

The following figure shows the consoles at the end of the procedure.



Working with Custom Boards

When working with custom boards, two methods are available, depending on your needs:

- Create an example on the mainboard, then remove the mainboard on the Project Configurator OVERVIEW tab using the **Change Target/SDK** control. This way, the wiring of the kit is kept, but you can change it using the Pin Tool.
- Create an example with just the part, no mainboard, and set up the wiring for all required peripherals of the example (like buttons or pins) using the Pin Tool.

Note that a mainboard can also depend on components (typically peripheral init components) that are removed if you remove the mainboard from the project.

Copyright © 2025 Silicon Laboratories. All rights reserved.

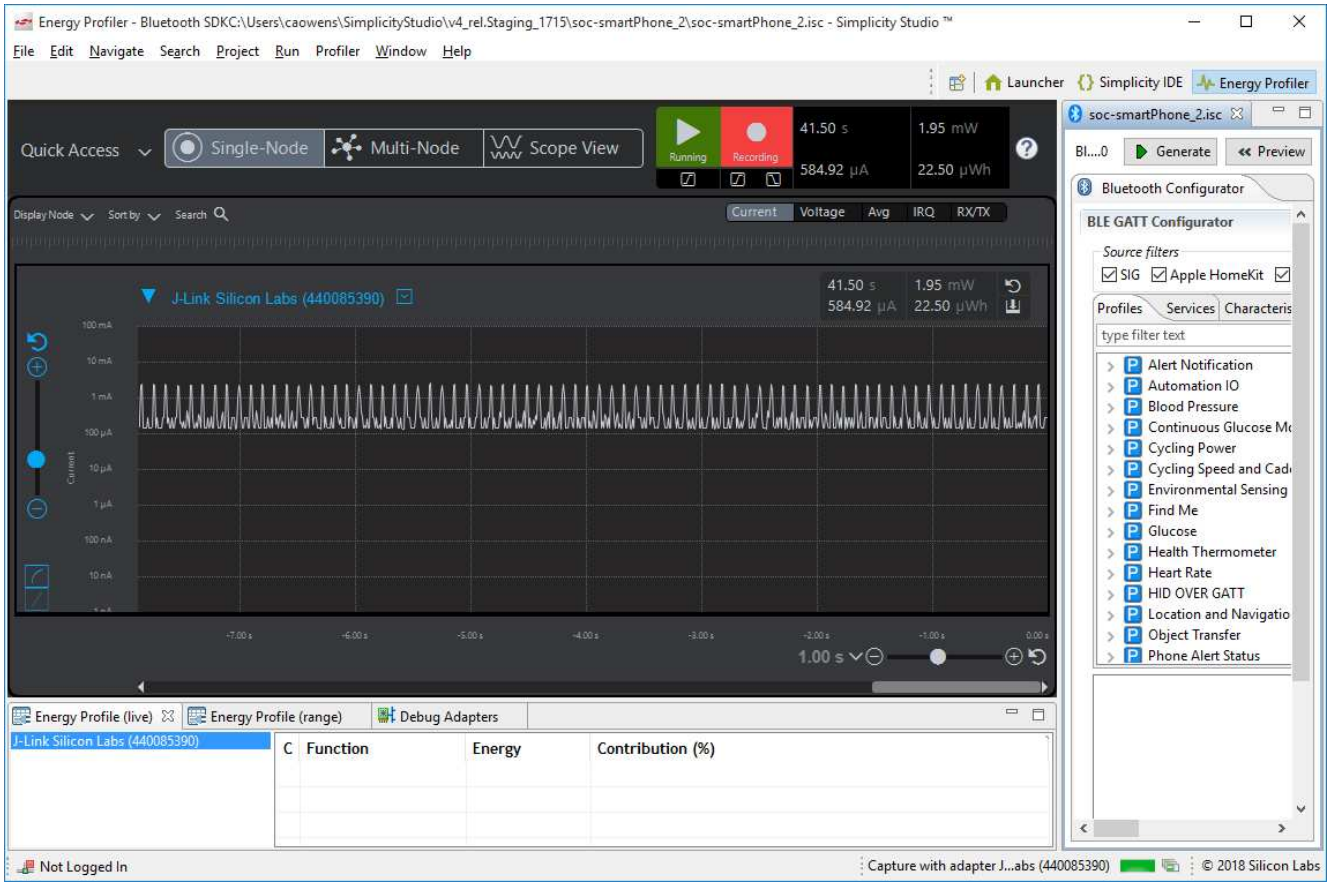
Next Steps

Next Steps

Multi-Node Energy Profiler

Multi-Node Energy profiler is an add-on tool, with which you can measure the energy consumption of your device in runtime. You can easily find peak and average consumption, and check for sleep mode current.

To profile the current project, drop down the Profile as menu in the Simplicity IDE perspective and select **Profile as / Simplicity Energy Profiler target**. This automatically builds your project, uploads it to the device, and starts Energy Profiler. A new Energy Profiler perspective appears, shown in the following figure.



See *UG343: Multi-Node Energy Profiler User's Guide* for details on how to use this tool. You can switch easily between Simplicity IDE and Energy Profiler perspectives using the Perspective buttons in the upper right corner of your current perspective.



You can see peaks in the energy consumption diagram. Pause profiling by clicking the green Play button, click one of the peaks, and zoom in with time axis (y-axis) zoom until you see sufficient details. These usually represent packet transmissions . You can also see the corresponding Tx events in the Rx/Tx bar below, provided that you enabled Rx/Tx view in the upper right corner. Note that the maximum consumption may now be greater than it appeared on the diagram before

Copyright © 2025 Silicon Laboratories. All rights reserved.

you zoomed in. This is because in zoomed-out mode, the displayed values are averaged. If you need exact values, always zoom in. To measure average consumption, click and drag your mouse over a time interval. A new window appears in the upper right corner showing consumption information for the given interval.. Overall average is measured as well, but this is influenced by transient events.



Multi-node Energy Profiler is also able to simultaneously measure the consumption of multiple devices. To start measuring a new device click the Quick Access menu (upper left corner) and select **Start Energy Capture**. To stop measuring, click the Quick Access menu, and select **End/Save session**.



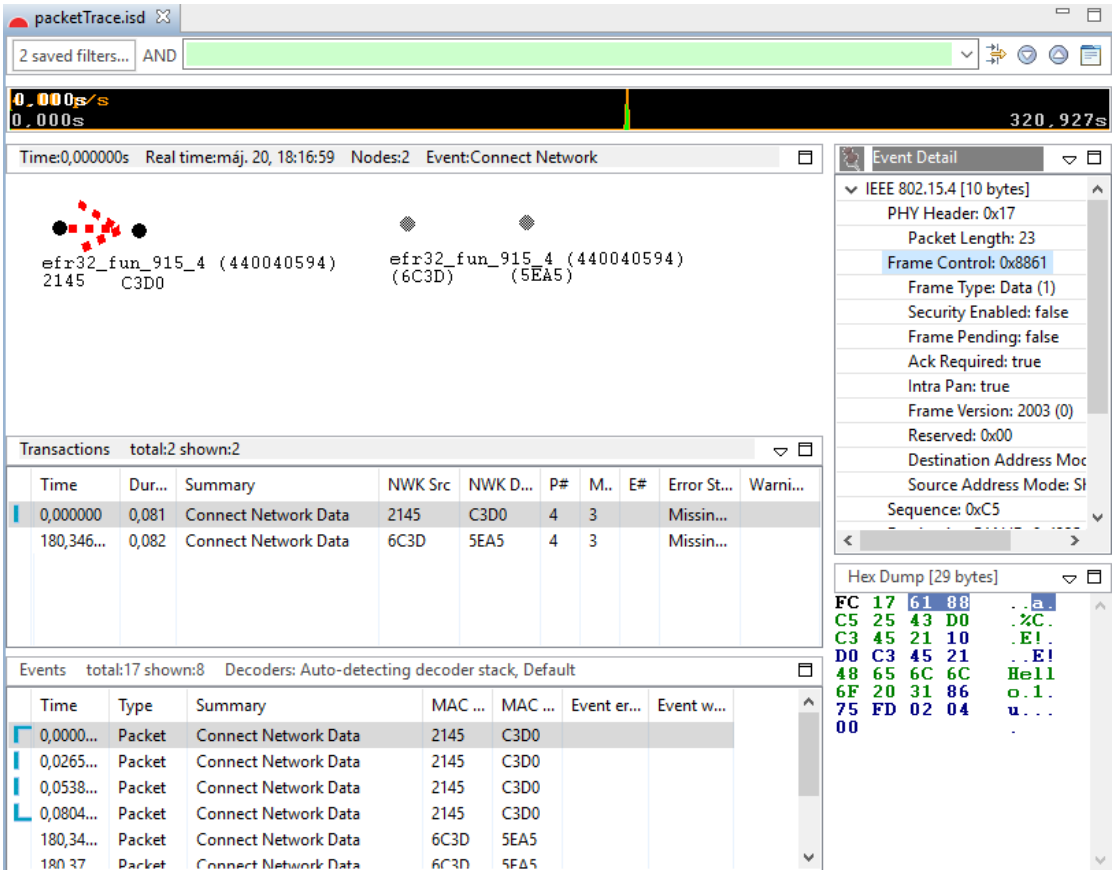
Copyright © 2025 Silicon Laboratories. All rights reserved.

To learn more about how to use this tool, see *UG343: Multi-Node Energy Profiler User's Guide*.

Network Analyzer

Silicon Labs Network Analyzer is a packet capture and debugging tool that can be used to debug connectivity between Wireless Geckos and other devices. It significantly accelerates the network and application development process with graphical views of network traffic, activity, and duration.

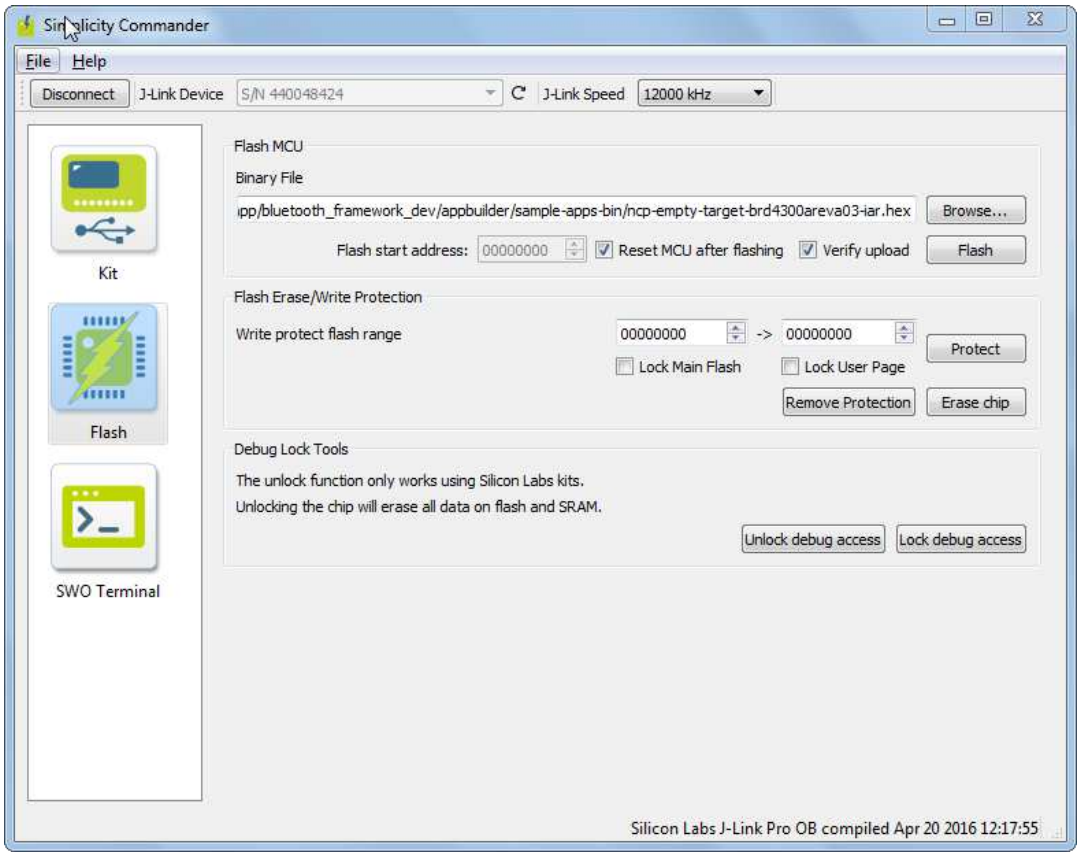
The Packet Trace application captures the packets directly from the Packet Trace Interface (PTI) available on the Wireless Gecko SoCs and modules. It therefore provides a more accurate capture of the packets compared to air-based capture.



Simplicity Commander

Simplicity Commander is a simple flashing tool, which can be used to flash firmware images, erase flash, lock and unlock debug access, and write-protect flash pages via the J-Link interface. Both GUI and CLI (Command Line Interface) are available. See *UG162: Simplicity Commander Reference Guide* for more information.

Copyright © 2025 Silicon Laboratories. All rights reserved.



Copyright © 2025 Silicon Laboratories. All rights reserved.

Overview

Connect Fundamentals

Silicon Labs has produced a series of documents on topics that provide useful background for Silicon Labs Connect developers.

- Silicon Labs Connect Fundamentals: Describes the features and functions of the Silicon Labs Connect stack, including its device types, network topologies, and its 'building block' development methodology using components.
- Using Silicon Labs Connect v3.x with IEEE 802.15.4 (PDF): Introduces the IEEE 802.15.4 standard on which Connect v3.x is based.
- Wireless Networking Application Development Fundamentals: For those new to wireless networking, introduces some fundamental concepts of wireless networking.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Connect Fundamentals

Silicon Labs Connect Fundamentals

NOTE: This section replaces *UG103.12: Silicon Labs Connect Fundamentals*. Further updates to this user guide will be provided here.

This guide describes the features and functions of Silicon Labs Connect, including its device types, modes of operation, stack structure, and the IEEE 802.15.4 support for RAIL vs. Connect.

Silicon Labs' *Fundamentals* series covers topics that project managers, application designers, and developers should understand before beginning to work on an embedded networking solution using Silicon Labs chips, networking stacks such as EmberZNet PRO or Silicon Labs Bluetooth, and associated development tools. These guides can be used as a starting place for anyone needing an introduction to developing wireless networking applications, or who is new to the Silicon Labs development environment. The *Connect User's Guide* provides in-depth information for developers who are using the Silicon Labs Connect Stack for their application development. Two series are published, one for Connect SDK v2.x and one for Connect SDK v3.x. Refer to *UG235.01: Developing Code with Silicon Labs Connect v2.x* and *UG435.01: Developing Code with Silicon Labs Connect v3.x* for an overview of the chapters in their respective *Connect User's Guide*. Proprietary is supported on all EFR32FG devices. For others, check the device's data sheet under Ordering Information > Protocol Stack to see if Proprietary is supported. In Proprietary SDK version 2.7.n, Connect is not supported on EFR32xG22.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Overview

Overview

Silicon Labs is developing products designed to meet customer demands as we move to an ever-connected world of devices in the home, often referred to as the IoT (Internet of Things). At a high level, the IoT goals for Silicon Labs are to:

- Connect all devices in the home with best-in-class mesh networking, whether with Ember ZigBee PRO or other emerging standards.
- Leverage the company's expertise in low-power, constrained devices.
- Enhance established low-power, mixed-signal chips.
- Provide low-cost bridging to existing Ethernet and Wi-Fi devices.
- Enable cloud services and connectivity to smartphones and tablets that promote ease of use and a common user experience for customers.

Achieving these goals will increase adoption rates and user acceptance for IoT devices in the Connected Home.

A common challenge in the IoT is managing devices requiring low power consumption, such as battery-powered devices where long battery life is essential. To meet this challenge, Silicon Labs has developed the Silicon Labs Connect stack. Connect provides a fully-featured, easily customizable wireless networking solution optimized for devices that require low power consumption and are used in a simple network topology. Connect is configurable to be compliant with regional communications standards worldwide. Each RF configuration is designed for maximum performance under each regional standard.

The Silicon Labs Connect stack supports many combinations of radio modulation, frequency, and data rates. The stack provides support for end nodes, coordinators, and range extenders. It includes all wireless MAC (Medium Access Control) layer functions such as scanning and joining, setting up a point-to-point or star network, and managing device types such as sleepy end devices, routers, and coordinators. With all this functionality already implemented in the stack, users can focus on their end application development and not worry about the lower-level radio and network details.

The Connect stack should be used in applications with simple network topologies, such as a set of data readers feeding information directly to a single central collection point (star or extended star topology), or a set of nodes in the same range exchanging data in a single-hop fashion (direct devices or MAC devices). It does not provide a full mesh networking solution such as that provided by the EmberZNet PRO or Silicon Labs Thread stacks.

The Connect stack is part of the Silicon Labs Flex SDK (Software Development Kit), installed through Simplicity Studio. Connect runs on top of RAIL (Radio Abstraction Interface Layer), also included with the Flex SDK. RAIL provides an intuitive, easily-customizable radio interface layer that is designed to support proprietary or standards-based wireless protocols. For more information, see RAIL Fundamentals.

The Connect stack supports efficient application development through its "building block" plug-in design. When used with the Simplicity Studio IDE (Integrated Development Environment), developers can easily select the functions that should be included in the application. The resulting applications are completely portable, in that they can be recompiled for different regions and EFR32 devices.

Copyright © 2025 Silicon Laboratories. All rights reserved.

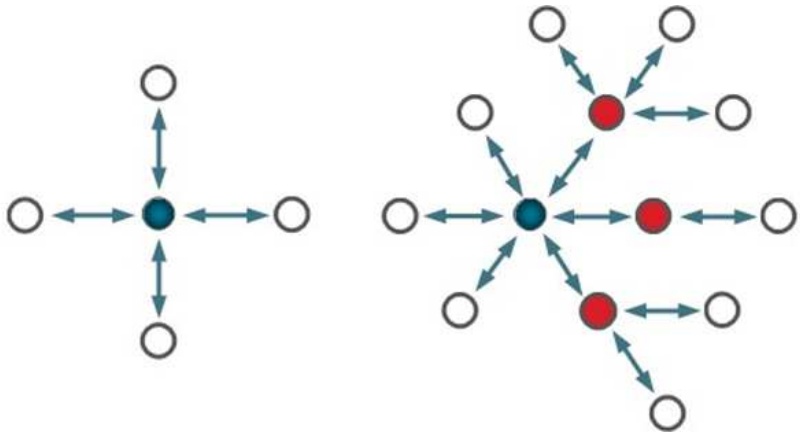
Connect Modes of Operation

Connect Modes of Operation

Connect supports three distinct modes of operation. Only one mode is allowed in any single network and there is no simple way to upgrade from one mode to the other. As a result, you should select the mode carefully early in the design process.

Extended Star Mode

In this mode, Connect supports star and extended star topology networks per the following figure.



Extended Start Mode supports the following device types.

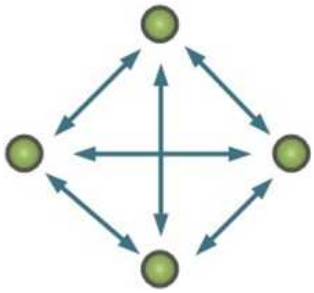
Device	Indicator	Description
Star coordinator	Blue circle	The star coordinator forms and manages the star or extended star network. The star coordinator also communicates with other range extenders and end nodes. Each Connect star network has a single coordinator.
Star range extender	Red circle	A device between the star coordinator and one or more star end nodes that can be used to extend the range of the star end nodes.
Star node	White circle	Joins to a star coordinator or a star range extender.

Data message routing between any two devices is supported by the network layer in this mode. Extended star topology is a centralized network. Hence, joining to the network requires acceptance by the Personal Area Network (PAN) coordinator. Short address allocation and assignment can be handled by the PAN coordinator. This mode is not fully IEEE 802.15.4 compliant.

Direct Mode

In this mode, Connect only provides connections between devices that are in range of each other (see the figure below). This is not a centralized topology.

Copyright © 2025 Silicon Laboratories. All rights reserved.



Direct mode supports the **Direct device** type, which send (and receive) messages to (and from) other (Direct) devices in range on the same PAN, with no star topology restrictions. Message forwarding can be implemented in an upper layer.

The Connect network layer is still enabled in this mode, but it does not provide routing, only endpoints. Routing protocols can be implemented in the application layer. Any device can join the PAN by setting the right PAN parameters. Short address allocation is not provided by the Connect stack and address duplication must be avoided by the application. This mode is not fully IEEE 802.15.4 compliant.

MAC Mode

MAC mode supports the **MAC device** type, which send and receive standard 802.15.4 messages from other 802.15.4 devices in range. Message forwarding can be implemented in an upper layer.

MAC mode is a fully IEEE 802.15.4 compliant setup of the Connect MAC layer; the network layer is not enabled. The API is more complex compared to Direct mode and requires some knowledge of the IEEE 802.15.4 standard. To make it fully IEEE 802.15.4 compliant, make sure to set up an IEEE802.15.4 compliant radio configuration.

Examples of Connect Modes and Network Topologies

An example of a Connect network is a network of temperature and humidity sensor end nodes installed throughout a home. Each end node periodically takes a reading and transmits that data either directly to a coordinator (sink) or, for those sensors placed farther away from the coordinator, to a range extender. The range extenders take data from the sensors and forward them to the coordinator. The coordinator not only forms and manages the network, but also sends the compiled data to an environmental management system that is part of another network.

Another example is a topology of two minimally-featured nodes that exchange data in both directions. This topology can be used as a generic wire replacement.

A third example is a topology of N direct devices or MAC devices all in range that exchange data between any pair of two nodes in both directions in a single-hop fashion.

Copyright © 2025 Silicon Laboratories. All rights reserved.

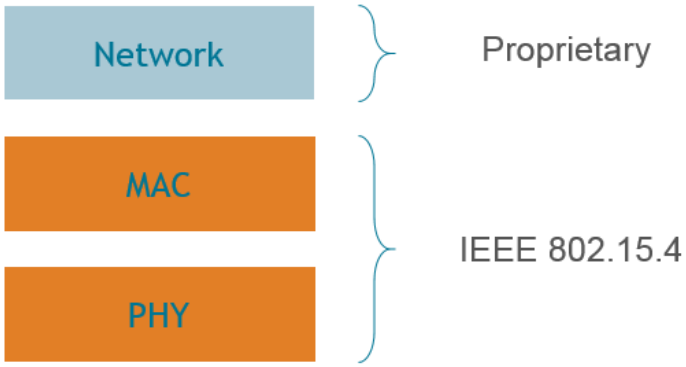
Stack Structure

Stack Structure

The Connect stack provides code organized into three functional layers, as shown in the following figure:

- PHY (physical)
- MAC (Media Access Control)
- Network

The PHY and MAC layers are based on the IEEE 802.15.4-2011 standard (abbreviated to IEEE 802.15.4). The Network layer is based on a proprietary protocol.



Additionally, the Connect Application Framework provides a complete tool and API infrastructure over the underlying stack layers. Functionality within the Application Framework and the Connect stack layers is provided in the form of individual building blocks called plugins. Details of the plugins for each layer are provided in the *Silicon Labs Connect Application Framework API Reference* included in the Connect stack documentation.

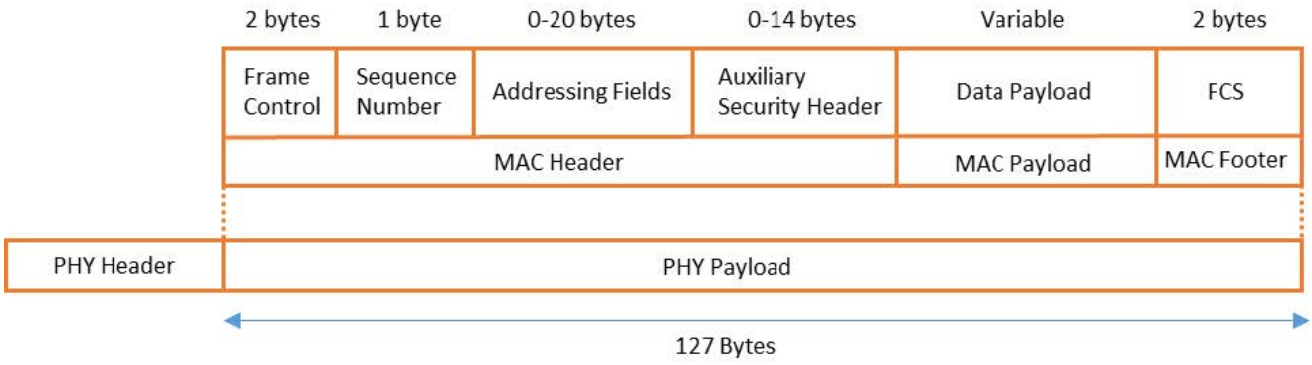
PHY and MAC Layers

The IEEE 802.15.4 specification is a standard for wireless communication that defines the MAC and PHY operating at subGHz frequencies and in the 2.4GHz band. IEEE 802.15.4 was designed with low power in mind. The Connect stack is based on the IEEE 802.15.4 standard, however it does not fully comply with the standard. On the other hand, some of the difference is beneficial, as the Connect PHY is highly customizable (whereas the IEEE 802.15.4 specification is rigidly restrictive).

The 802.15.4 MAC layer is used for basic message handling and congestion control. The Connect network layer builds on these underlying mechanisms to provide end-to-end communications in the network. The MAC layer includes a CSMA/CA (Carrier Sense Multiple Access / Collision Avoidance) mechanism for devices to listen on and determine whether the active channel is clear, as well as handle retries and acknowledgement of messages for reliable communications between adjacent devices. The MAC layer also provides security functionality (authentication, encryption, and replay attack protection). The MAC auxiliary header indicates which security scheme is used for a given packet. Optional security schemes can be implemented through the AES (Advanced Encryption Standard) plugin. The destination node looks at the auxiliary header and uses the correct security scheme (if it supports it) to decrypt and authenticate the incoming packet. The Connect stack supports both short (2-byte) and long (8-byte) identifiers. A network is identified by a 2-byte PAN ID.

Copyright © 2025 Silicon Laboratories. All rights reserved.

One of the characteristics derived from the need for low power and limiting the BER (Bit Error Rate) is enforcing smaller sized packets to be sent over the air. These can be up to a maximum of 127 bytes at the PHY layer payload. The MAC layer payload can vary depending on the security options and addressing type as illustrated in the figure below.



Network Layer

In extended star mode, the proprietary network layer provides network formation and full routing support, meaning every node in the network can communicate with any other node in the network in both directions. Routing is transparent to the application layer. Routing is not performed for direct devices.

The network formation functionality offers an association mechanism that, while similar to that in the 802.15.4 protocol, has been improved and made more secure by providing a special encrypted association request command not present in the 802.15.4 protocol. Network formation also includes centralized address allocation at the star coordinator. The network layer additionally provides APIs for commissioning node network parameters like node ID, PAN ID, channel, and transmission power. Direct devices can join to a network using the commissioning APIs. MAC devices can join to a network using either the standard 802.15.4 joining procedure or the commissioning APIs.

Application Framework

The Connect Application Framework leverages the Ember Application Framework v6 and its bookkeeping functionality implemented in callbacks such as `init()`, `tick()`, and `stackStatus()`. Application Framework plugins can provide callbacks and can implement the callbacks of other plugins.

Functionality Blocks

Every Connect application includes the following functionality blocks. The HAL (hardware abstraction layer) and Simulated EEPROM functionality blocks reside below the Connect stack. The PHY, event system, and message builder/parser functionality blocks are part of the stack itself.

- **HAL**: Drivers for devices and their peripherals such as SPI (Serial Peripheral Interface), UART (Universal Asynchronous Receiver/Transmitter), timers, and so on.
- **Simulated EEPROM**: Wear-leveled persistent storage of network and application data.
- **Event system**: System that allows the stack and the application to schedule code to run after some specified time interval. Events are also useful when an ISR (Interrupt Service Routine) needs to initiate an action that should run outside the ISR context.
- **PHY**: Software module that interfaces with the transceiver and provides basic radio TX, RX, and radio sleep functionality.
- **Message builder/parser**: Provides a 15.4-like PHY/MAC packet format builder/parser and a proprietary network layer format builder/parser.
- **Dynamic memory allocation**: A generic lightweight module that provides dynamic memory allocation and garbage collection.

Copyright © 2025 Silicon Laboratories. All rights reserved.

IEEE 802.15.4 Support RAIL vs Connect

IEEE 802.15.4 Support: RAIL vs Connect

The Connect stack relies on hardware-accelerated IEEE 802.15.4 functionality implemented by the RAIL library, and this interface is also available to application code. However, there are some differences in the developer experience (and application solution space) between Connect-based and "bare metal" RAIL-based 802.15.4 applications.

In most cases, using Connect for building an IEEE 802.15.4 application is easier than implementing the same feature set based on the RAIL library directly. Connect already provides the following features:

- Association procedure to connect nodes to a network
- CSMA/CA to minimize collision and enable higher throughput / less lost packets
- Security (authentication, encryption, and replay attack protection)
- Frame assembly (the customer only needs to provide the message payload)

As these features are not implemented in RAIL, it is the customer's responsibility to implement them (if desired) when not using Connect.

Cases when the RAIL library may be preferred over the Connect stack for IEEE 802.15.4-based communication are those where the device lacks sufficient code memory for the Connect stack, or where a non-standard implementation is necessary.

The RAIL library IEEE 802.15.4 API is available at IEEE 802.15.4 for reference.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Next Steps

Next Steps

See *QSG138: Getting Started with the Silicon Labs Flex Software Development Kit for the Wireless Gecko (EFR32™) Portfolio* for instructions on using Simplicity Studio and the WSTK to develop a Connect-based application using the Flex SDK.

See *UG235: Silicon Labs Connect User's Guide* for detailed Connect stack documentation.

See *AN902: Building Low Power Networks with the Silicon Labs Connect Stack* for instructions specific to low-power implementations.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Wireless Network App Development Fundamentals

Wireless Networking Application Development Fundamentals

NOTE: This section replaces *UG103.01: Wireless Networking Application Development Fundamentals*. Further updates to this user guide will be provided here.

This volume of Silicon Labs' *Fundamentals* series introduces some fundamental concepts of wireless networking. Other documents in the series frequently refer to these concepts. If you are new to wireless networking, you should read this guide first. Silicon Labs' *Fundamentals* series covers topics that project managers, application designers, and developers should understand before beginning to work on an embedded networking solution using Silicon Labs chips, networking stacks, such as the EmberZNet stack or Silicon Labs Bluetooth, and associated development tools. These guides can be used as a starting place for anyone needing an introduction to developing wireless networking applications, or who is new to the Silicon Labs development environment.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Overview

Overview

As embedded system design has evolved and the Internet of Things (IoT) has emerged, the need for networking support has become a basic design requirement. Like more general-purpose computers, embedded systems have moved toward wireless networking. Most wireless networks have pushed toward ever-higher data rates and greater point-to-point ranges. But not all design applications require high-end wireless networking capabilities. Low-data-rate applications have the potential to outnumber the classic high-data-rate wireless networks worldwide. Simple applications such as lighting control, smart utility meters, Heating, Ventilation, and Air Conditioning (HVAC) control, fire/smoke/CO alarms, remote doorbells, humidity monitors, energy usage monitors, and countless other devices function very well with low-data-rate monitoring and control systems. The ability to install such devices without extensive wiring decreases installation and maintenance costs. Increased efficiencies and cost savings are the primary motives behind this applied technology.

A wireless sensor network (WSN) is a wireless network consisting of distributed devices using sensors at different locations to cooperatively monitor physical or environmental conditions, such as temperature, sound, vibration, pressure, motion, or pollutants.

In addition to one or more sensors, each node in a sensor network is typically equipped with a radio transceiver or other wireless communications device, a small microcontroller, and an energy source, usually a battery. The size of a single sensor node can vary from shoebox-sized nodes down to devices the size of coins. The cost of sensor nodes is similarly variable, depending on the size of the sensor network and the complexity required of individual sensor nodes. Size and cost constraints on sensor nodes result in corresponding constraints on resources such as energy, memory, computational speed, and bandwidth.

Wireless personal area networks have emerged partially as a result of the IEEE 802.15.4 standard for low data rate digital radio connections between embedded devices. Above the medium access control (MAC) and physical (PHY) layers, there have been a number of standards formed such as The Connectivity Standards Alliance and the Thread Group, which were formed to standardize industry efforts to supply technology for networking solutions that are based on 802.15.4, have low data rates, consume very little power, and are therefore characterized by long battery life. The Zigbee Standard makes possible complete and cost-effective networked homes and similar buildings where all devices are able to communicate for monitoring and control. The Thread Group brings IPv6 to these small embedded devices.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Embedded Networking

Embedded Networking

While the term wireless network may technically be used to refer to any type of network that functions without the need for interconnecting wires, the term most commonly refers to a telecommunications network, such as a computer network. Wireless telecommunications networks are generally implemented with radios for the carrier or physical layer of the network.

One type of wireless network is a wireless local area network or LAN. It uses radios instead of wires to transmit data back and forth between computers on the same network. The wireless LAN has become commonplace at hotels, coffee shops, and other public places. The wireless personal area network (WPAN) takes this technology into a new area where the distances required between network devices is relatively small and data throughput is low.

In the control world, embedded systems have become commonplace for operating equipment using local special-purpose computer hardware. Wired networks of such devices are now common in manufacturing environments and other application areas. Like all computer networks, the interconnecting cable systems and supporting hardware are messy, costly, and sometimes difficult to install. To overcome these problems, wireless networking of embedded systems (that is, embedded networking) has become commonplace. However, the costly embedded networking solutions have only been justifiable in high-end applications where the costs are a secondary consideration. Low-cost applications with low data rate communications requirements did not have a good standardized solution until the IEEE 802.15.4 standard for wireless personal area PHY and MAC layers was released in 2003.

> **Note**: The current version as of this writing is the IEEE 802.15.4-2006 standard.

The Connectivity Standards Alliance was formed to establish networking and application-level standards on top of the IEEE 802.15.4 standards, to allow flexibility, reliability, and interoperability. The Zigbee 802.15.4-2003 Specification 1.0 was ratified in 2005 and the Zigbee 2006 Specification was announced in 2006, obsoleting the 2004 stack. Working Groups (WGs) have been formed within the Internet Engineering Task Force (IETF) to establish open standard approaches for routing (roll WG) and interfacing low-powered wireless devices to IPV6 networks (6LoWPAN WG). More recently, the Thread Group was formed in 2014 to utilize open IP standards, mesh networking, and 802.15.4 to support a wide array of home networking products.

Although wireless networks eliminate messy cables and enhance installation mobility, the downside is the potential for interference that might block the radio signals from passing between devices. This interference may be from other wireless networks or from physical obstructions that interfere with the radio communications. Interference from other wireless networks can often be avoided by using different channels. Zigbee, for example, has a channel-scanning mechanism on start-up of a network to avoid crowded channels. Standards-based systems, such as Thread, Zigbee and Wi-Fi, use mechanisms at the MAC layer to allow channel sharing. In addition, Zigbee and Thread provide an interoperable standard for multi-hop wireless networking, allowing signals to reach their destination by traveling through multiple relay points. These networks can be comprised of many such relay points or "routers," each one within range of one or more other routers, creating an interconnected "mesh" of devices that can provide redundant paths for data within the network that are automatically rediscovered and used to avoid interference in a local area. This concept is collectively referred to as "mesh networking."

Another potential problem is that wireless networks may be slower than those that are directly connected through a cable. Yet not all applications require high data rates or large data bandwidth. Most embedded networks function very well at reduced throughputs. Application designers need to ensure their system data rates are within the range achievable with the system being used.

Wireless network security is also a problem, because the data can easily be overheard by eavesdropping devices. Zigbee and Thread have a set of security services designed around Advanced Encryption Standard (AES) 128 encryption, so that

Copyright © 2025 Silicon Laboratories. All rights reserved.

application designers have a choice of security levels based on the needs of their applications. Careful design around these standards helps maintain high levels of network security.

Other networking standards exist such as Bluetooth Classic. Each standard has its own unique strengths and essential areas of application. The bandwidth of Bluetooth is 1 Mbps, while 802.15.4-based protocols are one-fourth of this value. The strength of Bluetooth Classic lies in its ability to allow interoperability and replacement of cables. Zigbee and Thread's strengths are low cost, long battery life, and mesh networks for large network operation. Bluetooth is meant for point-to-point applications such as handsets and headsets, whereas Zigbee and Thread are focused on the sensors and remote controls market, large distributed networks, and highly reliable mesh networking. Bluetooth Low Energy is emerging rapidly as another low power wireless technology very well suited for point-to-point applications such as controlling a single device with your smartphone.

While in the past each chip supported a single protocol, new developments have allowed two protocols to operate on the same chip. This multiprotocol operation may be as simple as chip technology that supports different protocols, with the application protocol of choice loaded during manufacturing. More recently, chips such as Silicon Labs EFR32 products and shared operational infrastructure allow two protocols to share the same radio. For example, a dynamic multiprotocol implementation might allow an end user to use a smartphone app working with a Bluetooth Low Energy application on the device to control the device or perform diagnostics, while the device remains connected to its Zigbee home automation network. Multiprotocol Fundamentals describes the four different multiprotocol modes, their operational requirements, and discusses some considerations to take into account when implementing a multiprotocol device.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Radio Fundamentals

Radio Fundamentals

Radio is the wireless transmission of signals by modulation of electromagnetic waves with frequencies below those of visible light. Electromagnetic waves are, in the case of radio, a form of non-ionizing radiation, which travels by means of oscillating electromagnetic fields that pass through electrical conductors, the air, and the vacuum of space. Electromagnetic radiation does not require a medium of transport like a sound wave. Information can be imposed on electromagnetic waves by systematically changing (modulating) some property of the radiated waves, such as their amplitude or their frequency. When radio waves pass an electrical conductor, the oscillating fields induce an alternating current in the conductor. This can be detected and transformed into sound or other signals that reproduce the imposed information.

The word 'radio' is used to describe this phenomenon, and radio transmission signals are classed as *radio frequency emissions*. The range or spectrum of radio waves used for communication has been divided into arbitrary units for identification. The Federal Communications Commission (FCC) and National Telecommunications and Information Association (NTIA) arbitrarily define the radio spectrum in the United States as that part of the natural spectrum of electromagnetic radiation lying between the frequency limits of 9 kilohertz and 300 gigahertz, divided into various sub-spectrums for convenience.

The following names are commonly used to identify the various sub-spectrums:

Name	Sub-Spectrum
Very Low Frequencies (VLF)	3 kHz to 30 kHz
Low Frequencies (LF)	30 kHz to 300 kHz
Medium Frequencies (MF)	300 kHz to 3,000 kHz
High Frequencies (HF)	3,000 kHz to 30,000 kHz
Very High Frequencies (VHF)	30,000 kHz to 300,000 kHz
Ultra High Frequencies (UHF)	300,000 kHz to 3,000,000 kHz
Super High Frequencies (SHF)	3,000,000 kHz to 30,000,000 kHz
Extremely High Frequencies (EHF)	30,000,000 kHz to 300,000,000 kHz

Each of the sub-spectrums listed above are further subdivided into many other sub-portions or ‘bands.’ For example, the American AM Broadcast Band extends from 535 kHz to 1705 kHz, which is within the portion of the spectrum classified as *Medium Frequencies*.

Frequency Bands

The radio spectrum is regulated by government agencies and by international treaties. Most transmitting stations, including commercial broadcasters, military, scientific, industrial, and amateur radio stations, require a license to operate. Each license typically defines the limits of the type of operation, power levels, modulation types, and whether the assigned frequency bands are reserved for exclusive or shared use. Three frequency bands can be used for transmitting radio signals without requiring licensing from the United States Government:

Band	Description
900 MHz	The 900 MHz band was used extensively in different countries for different products including pagers and cellular devices. This band was considered to have good range characteristics. However, it can be less popular for products because it is not a worldwide unlicensed band, and products therefore need to be modified depending on where they are being used.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Band	Description
2400 MHz	The 2400 MHz band is a very commonly-used frequency band. This band was one of the first worldwide unlicensed bands and therefore became popular for wireless consumer products. Typical wireless technologies that use this band are 802.11b (1-11 Mbps), 802.11g (1-50 Mbps) and 802.15.4, as well as numerous proprietary radio types.
5200-5800 MHz	The 5200 MHz band has three sub-bands, the lowest being for indoor home use only, while the 5800 MHz frequencies can be used for long distance wireless links at very fast speeds (30 – 100 Mbps).

A common strategy is to use 2400 MHz in residential and home environments. The Connectivity Standards Alliance and Thread Group endorse the use of this band, and both are looking at other SubGHz bands to expand their capabilities

Signal Modulation

Modulation is the process of changing the behavior of a signal so that it transfers information. Modulation can also be thought of as a way to encode information to be transmitted to a receiver that decodes, or demodulates, the information into a useful form.

The basic radio frequency (RF) signal has a fundamental frequency that can be visualized as an alternating current whose frequency is referred to as the *carrier wave frequency*. The earliest method used for encoding information onto the carrier wave involved switching the carrier wave on and off in a specific time duration pattern. This was known as continuous wave (CW) mode. The carrier frequency can also be varied in its amplitude (that is, signal strength) or its frequency. These two modulation methods are called amplitude modulation (AM) and frequency modulation (FM), respectively. It is possible to impose a signal onto the carrier wave using these three basic modulation techniques and creative variations of these techniques.

The Silicon Labs EFR32 integrated circuit family uses a form of *offset quadrature phase-shift keying* (OQPSK) to modulate the carrier wave. *Phase-shift keying* (PSK) is a digital modulation scheme that conveys data by changing, or modulating, the phase of a reference signal such as the carrier wave. PSK is a derivative of FM techniques.

All digital modulation schemes use a finite number of distinct signals to represent digital data. In the case of PSK, a finite number of phases are used. Each of these phases is assigned a unique pattern of binary bits. Usually, each phase encodes an equal number of bits. Each pattern of bits forms the symbol that is represented by the particular phase. The demodulator, which is designed specifically for the symbol set used by the modulator, determines the phase of the received signal and maps it back to the symbol it represents, thus recovering the original data. To do so, the receiver must compare the phase of the received signal to a reference signal. Such a system is termed *coherent*.

Antennas

An antenna (or aerial) is an arrangement of electrical conductors designed to emit or capture electromagnetic waves. The ability of an antenna to emit a signal that can be detected by another antenna is referred to as *radio propagation*. Antennas are made to a certain size based on the operating frequencies. An antenna from a 2400 MHz radio cannot be used effectively on a 5800 MHz radio, or vice versa. However, an antenna from one type of 2400 MHz technology, such as Wi-Fi or Bluetooth, can be used in another 2400 MHz technology, such as Zigbee or Thread.

Two fundamental types of antennas are described with reference to a specific three-dimensional space:

- Omni-directional: radiates equally in all directions
- Uni-directional (also known as directional): radiates more in one direction than in the other. All antennas radiate some energy in all directions in free space, but careful construction results in substantial transmission of energy in certain directions and negligible energy radiated in other directions.

In general, because of the nature of mesh networking, an omni-directional antenna is desired to provide as many communication paths as possible.

How Far Signals Travel

The distance a radio signal will travel and the amount of information that can be transmitted is based on:

- The amount of power the antenna is transmitting into the air.

Copyright © 2025 Silicon Laboratories. All rights reserved.

The distance between the transmitting and receiving stations.
- How much radio signal strength the receiving radio needs.
- What types of physical/electrical obstructions are in the way.

Radio Transmit Power

Radio transmit power is measured in watts, and typically discussed in terms of dBm, decibels referenced to 1 milliwatt. Converting wattage to dBm allows radio link calculations using simple addition and subtraction (dBm= 10*log10(P/ 0.001)).

For example, a typical power amplified wireless radio card transmits at 100 milliwatts (or mW), which translates to a power output of 20dBm.

If 1 mW, or 0 dBm, is the baseline for power in decibels, then +3 dBm is some power level above 1 mW (2 mW to be specific). The EFR32 platforms include some variants that go to +13 dBm and others that go up to +20 dBm, both without the aid of an external power amplifier.

Signal Degradation

The radio also needs to be able to hear a radio signal at a certain level. The minimum signal strength required for a receiver to understand the data is called the *receive sensitivity*.

As the radio signal travels through the air, it weakens. When a radio signal leaves the transmitting antenna, the dBm is a high number (for example, 20dBm). As it travels through the air, it loses strength and drops to a negative number. At some point, a minimum value for dBm is reached, below which the radio will no longer successfully receive the transmission. This value represents the "receive sensitivity" or "Rx sensitivity." This value will vary with the type of radio used but is typically between -90dBm and -100dBm. (Refer to the datasheet for your radio chip for specific receive sensitivity figures.)

If you can achieve a signal level of -75dBm and your radio has an Rx sensitivity of -95dBm, you have 20dBm of extra signal to accommodate interference and other issues. This is called *margin*.

How Far Can the Radio Signal Go

If you know the power out and the receiver sensitivity, you can determine whether you can broadcast over a given distance. In the following example, you want to know if you can receive a signal over five miles. To do so, you need to know the free space loss between the radio transmitter and the receiver.

For example, free space loss of a 2.4 GHz signal at 5 miles is 118.36 dB. So, you can estimate signal strength over the range of the network as:

What	Add or subtract it	The value
Transmitter power	+	15 dBm
Transmitter antenna gain	+	14 dBi
Receiver antenna gain	+	14 dBi
Transmitter's coaxial cable loss	-	2 dB
Receiver's coaxial cable loss	-	2 dB
Free Space Loss @ 5 miles	-	118.36 dB
	Total	-79.36 dBm

In other words, a 15 dBm radio hooked into a 14 dBi antenna, transmitting 5 miles through free space to another radio hooked up to a 14 dBi antenna, yields approximately -79 dBm of signal. Note that antennas with gain are necessarily directional, and would need to be aimed at each other. However, physical obstructions such as buildings or trees would have a substantial impact on these calculations. Typical Zigbee and Thread networks use smaller, lower-cost antennas without the gain increase and only use power amplifiers if extended range is required. Occasionally, external low-noise amplifiers (LNAs) may also be employed to boost RX sensitivity of incoming signals just before they reach the radio.

This calculation is provided as an example. See *AN718: Manufacturing Test Overview*, section 4, for a summary of the tools that can be used for performing range tests with different protocols.

Copyright © 2025 Silicon Laboratories. All rights reserved.

The Flex SDK comes with the RangeTest application. It has a version that can be used to measure Bluetooth and IEEE 802.15.4 ranges, called "RangeTest BLE and IEEE 802.15.4". For more information, see Range Test Demo User's Guide in Flex SDK v3.x or *UG147: Range Test Demo User's Guide in Flex SDK v2.x*. Silicon Labs recommends that basic range testing be conducted in the expected environment to evaluate whether extended range is required.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Networking Basic Concepts

Networking: Basic Concepts

A *network* is a system of computers and other devices (such as printers and modems) that are connected in such a way that they can exchange data. This data may be informational or command-oriented, or a combination of the two.

A *networking system* consists of hardware and software. Hardware on a network includes physical devices such as computer workstations, peripherals, and computers acting as file servers, print servers, and routers. These devices are all referred to as *nodes* on the network.

If the nodes are not all connected to a single physical cable, special hardware and software devices must connect the different cables in order to forward messages to their destination addresses. A *bridge or repeater* is a device that connects networking cables without examining the addresses of messages or making decisions as to the best route for a message to take. In contrast, a *router* contains addressing and routing information that lets it determine, from a message's address, the most efficient route for the message. A message can be passed from router to router several times before being delivered to its target destination.

In order for nodes to exchange data, they must use a common set of rules defining the format of the data and the manner in which it is to be transmitted. A *protocol* is a formalized set of procedural rules for the exchange of data. The protocol also provides rules for the interactions among the network's interconnected nodes. A network software developer implements these rules in software applications that carry out the functions required by the protocol.

Whereas a router can connect networks only if they use the same protocol and address format, a *gateway* converts addresses and protocols to connect dissimilar networks. Such a set of interconnected networks can be referred to as an internet, intranet, wide area network (WAN), or other specialized network topology. The term Internet is often used to refer to the largest worldwide system of networks, also called the World Wide Web. The basic protocol used to implement the World Wide Web is called the Internet protocol, or IP.

A networking protocol commonly uses the services of another, more fundamental protocol to achieve its ends. For example, the Transmission Control Protocol (TCP) uses the IP to encapsulate the data and deliver it over an IP network. The protocol that uses the services of an underlying protocol is said to be a client of the lower protocol; for example, TCP is a client of IP. A set of protocols related in this fashion is called a *protocol stack*.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Wireless Networking

Wireless Networking

Wireless networking mimics the wired network, but replaces the wire with a radio signal as the data interconnection medium. Protocols are essentially the same as used in wired networks, although some additional functionality has been added so the two types of networks remain interoperable. However, wireless networks have emerged that do not have a wired counterpart requiring interoperability. These specialized networks have their own hardware and software foundations to enable reliable networking within the scope of their unique environments.

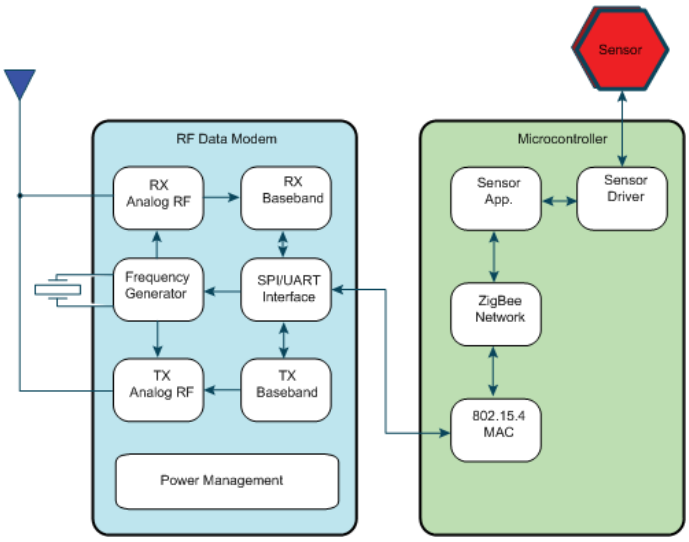
> **Note**: Most networking protocols are based, to some degree, on the Open Systems Interconnection (OSI) Model.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Networking Devices

Networking Devices

Silicon Labs has developed networking hardware (the EFR32xG family) and SDKs containing protocol libraries, application examples, and development tools to facilitate implementation of a wireless personal area network of devices for sensing and control applications. The following figure represents a typical wireless device using Zigbee, one of the Silicon Labs networking technologies. The RF data modem is the hardware responsible for sending and receiving data on the network. The microcontroller represents the computer control element that originates messages and responds to any information received. The sensor block can be any kind of sensor or control device. Such a system can exist as a node on a network without any additional equipment. Any two such nodes, with compatible software, can form a network. Large networks can contain thousands of such nodes.



Chips such as the EFR32xG family provide both the RF and microcontroller portions of the figure above. When used as "network coprocessors," the chips provide only the RF and networking part of the system, acting as a co-processor to any microcontroller, Digital Signal Processing (DSP), or similar device required for the application.

For details of the various networking technologies, see the specific Fundamentals document(s) for that technology.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Overview

Connect Developers Guide

The Developer's Guide content is organized in the following groups:

- Developing and Debugging: A description of development resources as well as detailed information on a variety of topics.
- Bootloading: Information on using the Gecko Bootloader with Connect applications.
- Multiprotocol: Background on implementing multiprotocol applications and information on different multiprotocol models.
- Non-Volatile Data Storage: Background on managing device memory.
- Security: Describes Silicon Labs security resources and how to manage Connect security.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Overview

Developing and Debugging Connect Applications

These pages provide details on developing and debugging applications using the Connect stack. Content includes:

- **Architecture of the Silicon Labs Connect Stack v3.x (PDF)**: Describes the architecture of the Silicon Labs Connect stack v3.x an how it implements IEEE 802.15.4.
- **Customizing Applications with Silicon Labs Connect v3.x (PDF)**: Describes how to use components, callbacks, and events on top of the Gecko Platform application framework to configure features and application behavior.
- **Network Co-Processor Applications with Silicon Labs Connect v3.x (PDF)**: Describes how to run the Silicon Labs Connect stack in Network Co-Processor (NCP) mode, where the NCP runs on the EFR32 while the Host application and the Co-processor Communication daemon (CPCd) run on the Host device.
- **Using Real Time Operating Systems with Silicon Labs Connect v3.x (PDF)**: Describes the process to implement a Connect-based application on top of one of the supported Real Time Operating Systems (RTOS).
- **Energy Saving with Silicon Labs Connect v3.x (PDF)**: Describes the features available in Connect v3.x to reduce power consumption. Using those features is described in *Building Low Power Networks with the Silicon Labs Connect Stack v3.x*.
- **Building Low Power Networks with the Silicon Labs Connect Stack v3.x**: Illustrates reducing power consumption in a Connect v3.x application using the sensor example.
- **EFR32 Radio Configurator Guide for Simplicity Studio 5 (PDF)**: Documents the Radio Configurator tool, which can be used to create customizations at the PHY level.
- **PHY Limitations and Timing Optimization**: Connect specific recommendations on radio config customizations.

Development Tools

Simplicity Studio and the Simplicity IDE: Simplicity Studio is the unified development environment for all Silicon Labs technologies, SoCs, and modules. It provides you with access to the target device-specific web and SDK resources, software and hardware configuration tools, and an integrated development environment (IDE) featuring industry-standard code editors, compilers, and debuggers. See the silabs.com Simplicity Studio page to download the tools and for more information.

Network Analyzer: Simplicity Studio® 5 (SSv5)'s Network Analyzer enables debugging of complex wireless systems. This tool captures a trace of wireless network activity that can be examined in detail live or at a later time. See the Network Analyzer section of the Simplicity Studio 5 User's Guide for more information.

Wireshark: Wireshark is the recommended network protocol analyzer for the use with Wi-SUN networks. Download instructions are provided for Windows/Mac users or Linux users. Simplicity Studio® 5 supports live interaction between the application running on a Silicon Labs device and Wireshark.

Energy Profiler: Simplicity Studio® 5 (SSv5)'s Energy Profiler enables you to visualize the energy consumption of individual devices, multiple devices on one target system, or a network of interacting wireless devices to analyze and improve the power performance of these systems. Real-time information on current consumption is correlated with the program counter providing advanced energy software monitoring capabilities. It also provides a basic level of integration with the Network Analyzer network analysis tool. See the Energy Profiler section of the Simplicity Studio 5 User's Guide for more information.

Simplicity Commander: Simplicity Commander is a single, all-purpose tool to be used in a production environment. It is invoked using a simple Command Line Interface (CLI) that is also scriptable. Simplicity Commander enables customers to complete essential tasks such as configuring and building applications and bootloaders and flashing images to their devices. Simplicity Commander is available through Simplicity Studio or can be downloaded through system-specific installers. The **Simplicity Commander User's Guide** provides more information.

Silicon Labs Configurator (SLC): SLC offers command-line access to application configuration and generation functions. **Software Project Generation and Configuration with SLC-CLI** provides instructions on downloading and using the SLC-CLI tool.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Building Low Power Networks with the Silicon Labs Connect Stack v3.x

Building Low Power Networks with the Silicon Labs Connect Stack v3.x

> NOTE: This section replaces *AN1252: Building Low Power Networks with the Silicon Labs Connect Stack v3.x*. Further updates to this application note will be provided here.

This application note illustrates techniques for reducing power consumption in a sensor network developed using Simplicity Studio and the Silicon Labs Connect Stack v3.x, distributed as part of the Silicon Labs Proprietary Flex Software Development Kit (SDK) v3.x with Simplicity Studio® 5. Techniques include creating sleepy end devices that, by their nature, consume less energy, reducing reporting frequency, and removing unnecessary peripheral energy use. The techniques are illustrated with the sensor and sink example applications provided with the Flex SDK. For more in-depth background on energy savings in Connect networks see *UG435.07: Energy Saving with Silicon Labs Connect v3.x*, part of the *Connect v3.x User's Guide* series.

Proprietary is supported on all EFR32FG devices. For others, check the device's data sheet under Ordering Information > Protocol Stack to see if Proprietary is supported. In Proprietary SDK version 2.7.n, Connect is not supported on EFR32xG22.

Key Points

- Building example sensor and sink applications.
- Creating a sensor-sink network.
- Monitoring baseline average current consumption.
- Reducing energy use in the network.

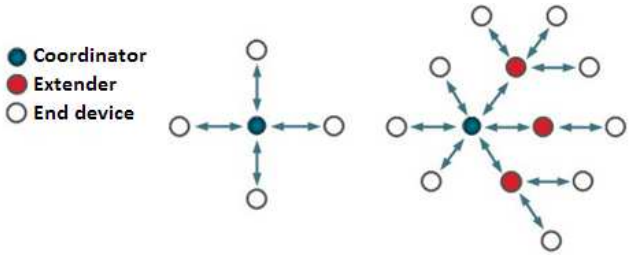
Copyright © 2025 Silicon Laboratories. All rights reserved.

Introducing Low Power Sensor Networks

Introducing Low Power Sensor Networks

A typical low power network includes a single "always-on" device that serves as the coordinator for the network and one or more end nodes operating as sleepy end devices. Such devices spend most of their time in deep sleep mode, waking only briefly to transmit data to the central coordinator. An example of a low power network is a simple sensor network where the sensors are all sleepy end devices and the central hub serves as the always-on coordinator, or sink, for the network.

The Silicon Labs Flex SDK includes example sensor and sink applications, based on the Connect stack, for exactly this scenario. Supported network structures are star and extended star topologies:



The example applications are:

- Connect (SoC): Sensor
- Connect (SoC): Sink

Together the applications demonstrate a star topology. Bi-directional communication is possible between the sensor(s) and the sink nodes. This document reviews using Simplicity Studio to build a sensor-sink network, and then to modify it for the lowest power consumption.

If you are not familiar with building the example applications in Simplicity Studio, review the Proprietary Flex SDK v3.x Quick Start Guide. If you are not familiar with the features and functions of the Silicon Labs Connect stack, see Silicon Labs Connect Fundamentals.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Create the Baseline Network

Create the Baseline Network

This section describes how to:

1. Build and run the example coordinator/sink application.
2. Build and run the sensor application.
3. Create the network.
4. Evaluate baseline energy consumption.

You will need at least two devices connected to a computer running Simplicity Studio.

Build and Run the Coordinator/Sink Application

1. In the Simplicity Studio Launcher perspective, select the device to be used as the coordinator/sink.
2. Start a new project by clicking **[Create New Project]**.
3. Create the project by selecting the **Connect: (SoC) Sink** example and completing the steps in the New Project wizard.
4. Compile (hammer icon) and debug (bug icon) the project. The coordinator application is now installed on the device.

To configure the coordinator, terminal software is necessary to set up the parameters. Use a separate application or the Simplicity Studio's built-in console tool. Issue `help` in the terminal to see some basic help from the firmware. When the firmware starts, issue `form 0` (form a network on channel 0) and `pjoin 255` (enable join of nodes for unlimited time) commands:

At this point, the coordinator is up and running, and ready to accept incoming joining requests, so sensor nodes can connect to the sink.



Build and Run the Sensor Application

1. Return to the Simplicity Studio Launcher perspective and select the device to be used as the sensor.
2. Start a new project by clicking **[Create New Project]**.
3. Create the project by selecting the **Connect: SoC Sensor** example, and completing the steps in the New Project wizard. All the settings can be kept as default. Be sure the settings of the radio configuration of the sink and the sensor match.
4. Compile and debug/run the application.

Create the Network

The two most important terminal commands for the sensor are `join` and `join-sleepy`. On issuing `join`, the device attempts to connect to the coordinator and, if the join process is successful, the node becomes part of the network. The device will not try to enter to sleep or idle mode at all, keeping the device awake with relatively high current consumption.

Copyright © 2025 Silicon Laboratories. All rights reserved.

The `join-sleepy` command enables the device to enter the sleep state. After a successful join process, the device starts sending the sensor data to the sink periodically (with a 1s rate by default).

At this point, a network consisting of a coordinator/sink and one node/sensor should work, the sensor data should be sent by the sensor and received by the sink with 1 s periodicity. More nodes can be added to the network by downloading the sensor firmware into additional devices. The status code of a successful join is 0x90: Network up, and the firmware prints Network up on the console.

```
Sensor
> join_sleepy 0

join sleepy 0x0
> Network up
TX: Advertise Request (unicast), status 0x0
RX: Advertise from 0x 0
TX: Pair Request to 0x 0: 0x0
RX: Pair Confirm from 0x 0
TX: Data to 0x 0: 41 7d 0 0 5c 93 0 0: 0x0
TX: Data to 0x 0: 41 7d 0 0 63 93 0 0: 0x0
TX: Data to 0x 0: 36 7d 0 0 6b 93 0 0: 0x0
```

The firmware prints numerical status codes on the console, instead of a textual description, if the sensor fails to join the network.

- `0x40` : No ACK received, the node to which a message was transmitted did not respond.
- `0x41` : Indirect MAC timeout, may occur if the sender did not get a poll request to send its pending message.
- `0x70` : Invalid call, usually when an unexpected operation is carried out (for example, joining when joined or sending when not joined, and so on).
- `0xAB` : No valid beacons, the other radio cannot be heard (for example, no antenna attached).

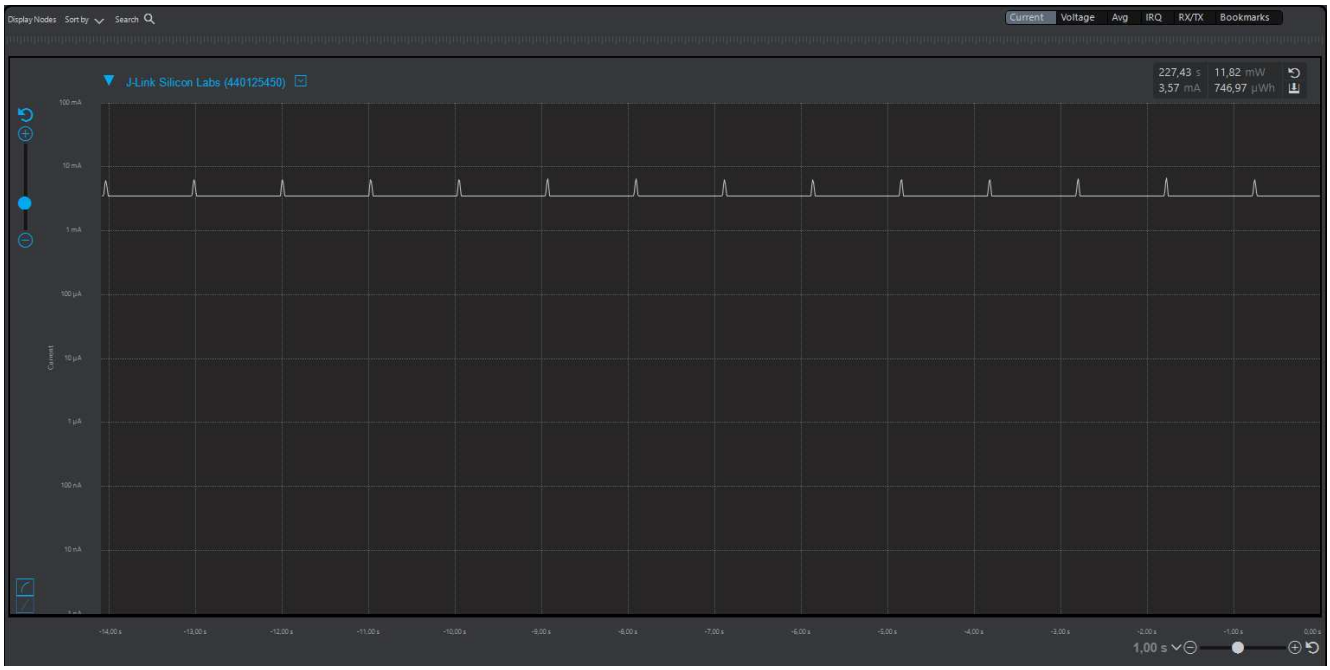
Before each step in the experiment with the sensor/sink examples it is a good practice to issue the `leave` command on both the coordinator/sink and sensors. For example, the coordinator maintains information about the previously connected nodes, based on their EUI64 ID. The nodes also store information and try to reconnect to the coordinator automatically at power-up based on this information. By issuing the `leave` command this information will be cleared.

Monitor Baseline Energy Use

The Energy Profiler is a useful tool to inspect current consumption real-time. For details on using the Energy Profiler see *UG343: Multi-Node Energy Profiler User's Guide*.

The following graph shows the consumption of a non-optimized sensor node.

Copyright © 2025 Silicon Laboratories. All rights reserved.



Copyright © 2025 Silicon Laboratories. All rights reserved.

Reduce Power Consumption

Reduce Power Consumption

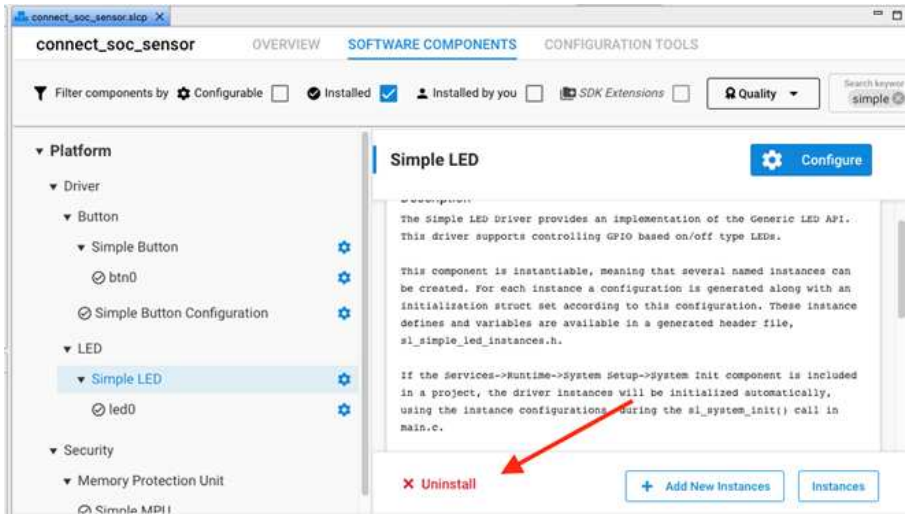
This section describes techniques for reducing power consumption, including:

- Disable Network LED
- Disable UART reception to enable sleep mode
- Change Report Periodicity
- Initialize the SPI Flash Chip

Finally, the reduced power consumption results are described.

Disable Network LED

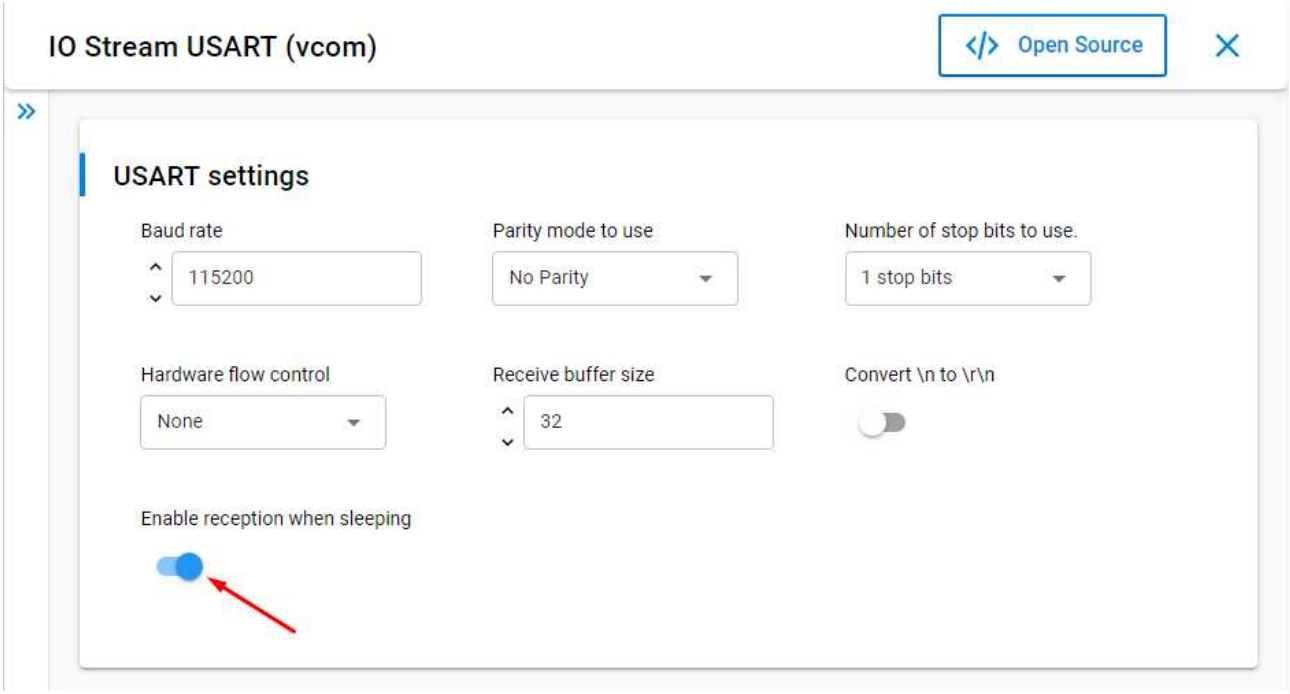
The sensor example sets an LED when the device is joined to a network, which draws a constant current, depending on the hardware (e.g., about 500uA on BRD4001). The easiest way to disable it is to remove the simple LED component.



Disable UART Reception during Sleep

Next, the step that reduces the consumption the most is disabling UART reception during sleep. This enables entering EM2 mode when it is possible. This option can be enabled in the Component Editor under **IO Stream USART (vcom)** settings.

Copyright © 2025 Silicon Laboratories. All rights reserved.



> **Note**: If reception is disabled during sleep, issuing commands through the CLI (UART) is not possible. The sensor example has a feature where PB0 toggles between enabled and disabled reception mode.

Change Report Periodicity

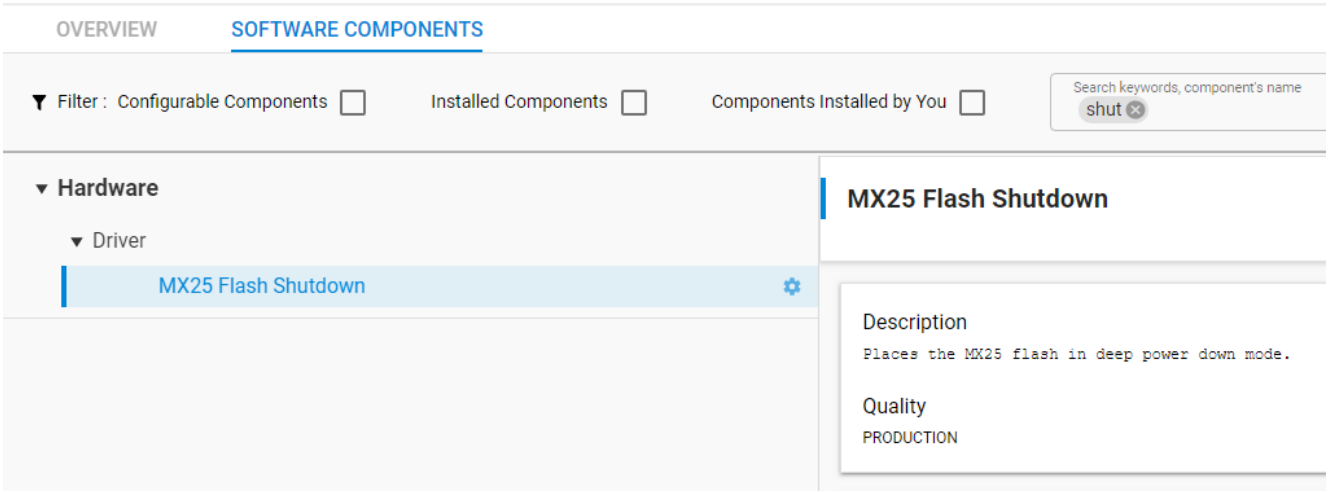
Changing the report period can significantly reduce the average consumption if the sleep consumption is significantly lower than consumption in active periods. To change the report period, open **app_process.c** and modify the `sensor_report_period_ms` value.

```
// -----------------------------------------------------------------------------
//                                Global Variables
// -----------------------------------------------------------------------------
/// report timing event control
EmberEventControl report_control;
/// report timing period
uint16_t sensor_report_period_ms =  (1 * MILLISECOND_TICKS_PER_SECOND);
/// TX options set up for the network
EmberMessageOptions tx_options = EMBER_OPTIONS_ACK_REQUESTED;
```

Initialize the SPI Flash Chip

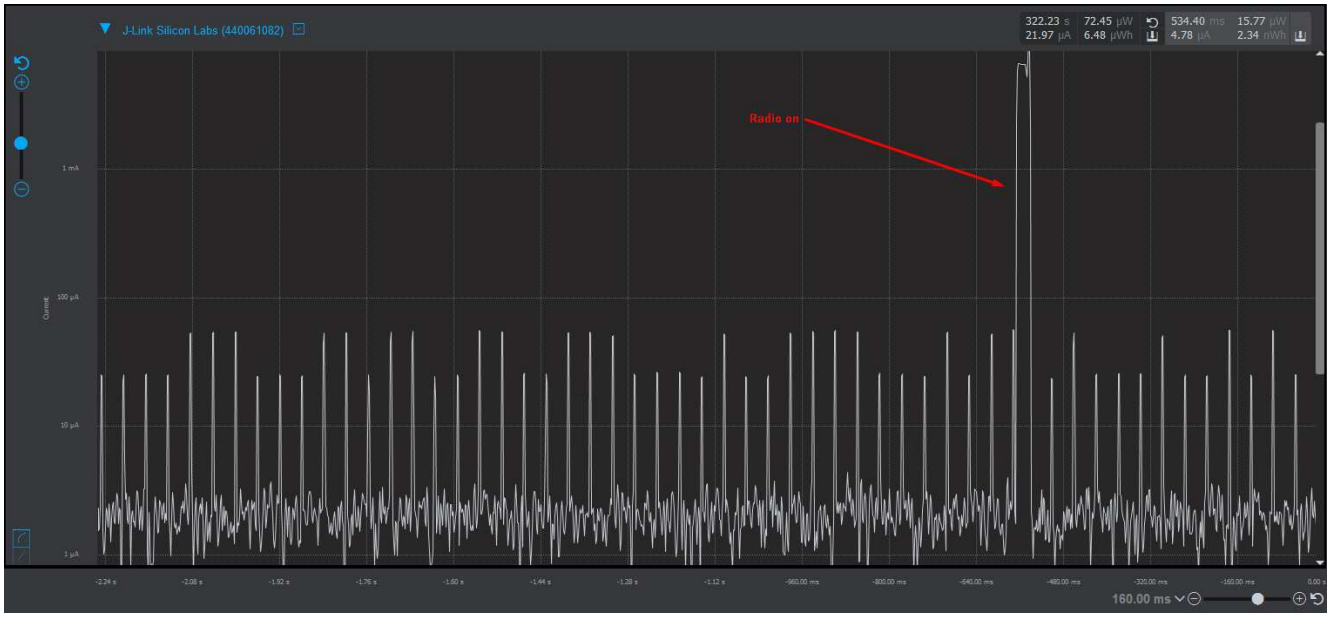
This step shouldn't be necessary, as it is always implemented if you create the project for a Silicon Labs radio board. However, if your design includes an external flash, or if you decided to create the project for the given MCU part, not the Silicon Labs radio board, it might be necessary. All the radio boards designed by Silicon Labs contain an SPI flash chip that consumes ~8-10 µA if left uninitialized. To reduce the SPI Flash current consumption, the **MX25 Flash Shutdown** component must be installed. In the sensor application, it is installed by default.

Copyright © 2025 Silicon Laboratories. All rights reserved.



Reduced Power Consumption Results

Finally, if these modifications are applied, the sleep mode current consumption should be reduced significantly.



> **Note**: The built-in current measurement of the WSTK board is not really accurate in the µA range and below. The sleep mode current consumption of the sensor node, once all modifications have been applied, should be in the 2-5 µA range.

Copyright © 2025 Silicon Laboratories. All rights reserved.

PHY Limitations and Timing Optimization

PHY Limitations and Timing Optimization

Supported profiles

Connect supports a wide range of radio configurations, including pre-configured and customized PHYs. The pre-configured PHYs are under the following profiles:

- Connect Profile
- Connect OFDM Profile
- Long Range Profile

Under these profiles, any customization is supported by Connect. Note, however, that using the radio configurator requires basic knowledge of digital modulations. It is possible to misconfigure modulation parameters to the level where communication might be impossible.

Base Profile

The Base profile allows the widest configuration options. However, Connect is only compatible with the IEEE 802.15.4 frame configuration. For the modulations available in the Base Profile, Connect only supports 1 Byte PHR frame configurations. To set this up, follow these settings:

- Check Header Enable
- Set Header Size to 1
- Set Frame Length Encoding to VARIABLE_LENGTH
- Set Frame Bit Endian to LSB_FIRST
- Check Length Includes CRC Bytes
- Set Minimum Length to 0
- Set Maximum Length to 127
- Set Variable Length Bit Size to 7
- Set Variable Frame Length Adjust to 0
- Set Variable Length Bit Endian to LSB_FIRST
- Set Variable Length Bit Location to 0

The frame setting also requires a 2 Byte long CRC, so make sure to select a 2 Byte long CRC Polynomial.

Although Connect can handle 2 Byte PHR and the PHR required for OFDM, those settings can only be activated by selecting a PHY from the Connect profiles.

OFDM Settings

The Connect OFDM PHYs are essentially same as Wi-SUN OFDM PHYs. However, without the limitation of the Wi-SUN spec, we allow changing the channel map and carrier frequency.

OFDM bitrate depends on bandwidth and MCS (modulation coding scheme). The bandwidth can be configured on the radio configurator, while MCS can be changed run-time, with the `emberOfdmSetMcs()` API. The default MCS is 0.

The available bitrates in kb/s, depending on bandwidth and MCS:

	0.2 MHz	0.4 MHz	0.8 MHz	1.2 MHz
MCS=0	12.5	25	50	100
MCS=1	25	50	100	200
MCS=2	50	100	200	400

Copyright © 2025 Silicon Laboratories. All rights reserved.

	0.2 MHz	0.4 MHz	0.8 MHz	1.2 MHz
MCS=3	100	200	400	800
MCS=4	150	300	600	1200
MCS=5	200	400	800	1600
MCS=6	300	600	1200	2400

Multi-PHY Considerations

Connect has limited support for the Multi-PHY capabilities of EFR32. Channel-based Multi-PHY (i.e., configuration changes are applied by changing the channel) is fully supported. However, Connect always loads the first protocol in a protocol-based Multi-PHY configuration. For more details on this term, see AN1253 (PDF).

Optimizing Connect for a PHY

Although Connect is set up to be usable with a wide range of PHYs, there are certain features that depend on the PHY. These are usually configured conservatively, so it will probably work with most configurations, but not all. Furthermore, even if it works, it might be necessary to optimize these parameters for more effective operation.

CSMA/CA

The default configuration in Connect follows the recommendation from IEEE 802.15.4. All timing parameters are symbolrate dependent, while the threshold is -65dBm. All of the parameters can be changed via `emberSetMacParams()`. These parameters very rarely need modification. Perhaps, on a very low symbolrate, it is worth setting an overall timout via `csmaTimeout`.

Turnaround Time and Acknowledgement Timeout

Turnaround time (delay between received packet and transmitted acknowledgement) in Connect is not configurable; it is always 12 symbol time.

Acknowledgement timeout by default is 25ms. This is much more than most PHY needs. Turnaround time + acknowledgement frame transmission time should be enough in theory. This can be tuned via `emberSetMacParams()`, with the `ackTimeout` argument.

Note though that in practice, optimizing the timeout doesn't improve efficiency much. The timeout only blocks the stack to transmit another packet while waiting on the acknowledgement.

Active Scan Duration

Active scan is used when a device joins a network. The joining device sends a beacon request, and waits a predefined duration for beacons of the joinable devices. This is the active scan duration, which is 960*2^5=30720 symbol time by default. This is however way too much for low bitrate; e.g. at 9.6kbps, this results in 3.2s.

This can be configured via `emberSetActiveScanDuration()`. The beacons are transmitted with a 50ms jitter after the beacon request frame (with additional delay caused by CSMA/CA).

Copyright © 2025 Silicon Laboratories. All rights reserved.

Overview

Bootloading Embedded Applications

Bootloading allows you to update application firmware images on your devices. This section provides background information about bootloading using the Silicon Labs Gecko Bootloader.

- **Bootloader Fundamentals**: Introduces bootloading for Silicon Labs networking devices. Discusses the Gecko Bootloader as well as legacy Ember and Bluetooth bootloaders, and describes the file formats used by each.
- **Bootloading and OTA with Silicon Labs Connect v3.x (PDF)**: Explains standalone (serial) and application (OTA) bootloader options available for use within Connect v3.x-based applications.
- **Using the Gecko Bootloader with Silicon Labs Connect**: Includes detailed information on using the Silicon Labs Gecko Bootloader with Connect. It supplements the general Gecko Bootloader implementation information provided in *Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher*.
- **Silicon Labs Gecko Bootloader User's Guide for Series 3 and Higher**: Describes the high-level implementation of the Silicon Labs Gecko Bootloader for Series 3 microcontrollers, SoCs (System on Chips), and NCPs (Network Co-Processors), and provides information on different aspects of configuring the Gecko Bootloader.
- **Gecko Bootloader User's Guide for GSDK 4.0 and Higher**: Describes the high-level implementation of the Silicon Labs Gecko Bootloader for EFR32 SoCs and NCPs, and provides information on how to get started using the Gecko Bootloader with Silicon Labs wireless protocol stacks in GSDK 4.0 and higher.
- **Series 2 Secure Boot with RTSL**: Contains detailed information on configuring and using the Secure Boot with hardware Root of Trust and Secure Loader on Series 2 devices, including how to provision the signing key. This is a companion document to UG266: Silicon Labs Gecko Bootloader User's Guide.
- **Transitioning to the Updated Gecko Bootloader in GSDK 4.0 and Higher**: Gecko Bootloader v2.x, introduced in GSDK 4.0, contains a number of changes compared to Gecko Bootloader v1.x. This document describes the differences between the versions, including how to configure the new Gecko Bootloader in Simplicity Studio 5.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Bootloader Fundamentals

Bootloader Fundamentals

NOTE: This section replaces *UG103.06: Bootloader Fundamentals*. Further updates to this user guide will be provided here.

This guide introduces bootloading for Silicon Labs networking devices. It describes the concepts of standalone and application bootloaders and discusses their relative strengths and weaknesses. In addition, it looks at design and implementation details for each method. Finally, it describes the bootloader file format.

Silicon Labs' *Fundamentals* series covers topics that project managers, application designers, and developers should understand before beginning to work on an embedded networking solution using Silicon Labs chips, networking stacks such as EmberZNet PRO or Silicon Labs Bluetooth, and associated development tools. These guides can be used as a starting place for anyone needing an introduction to developing wireless networking applications, or who is new to the Silicon Labs development environment.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Overview

Introduction

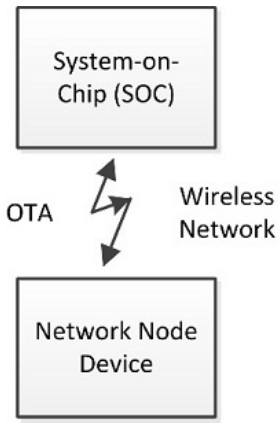
The bootloader is a program stored in reserved flash memory that can initialize a device, update firmware images, and possibly perform some integrity checks. Firmware image update occurs on demand, either by serial communication or over the air. Production-level programming is typically done during the product manufacturing process yet it is desirable to be able to reprogram the system after production is complete. More importantly, it is valuable to be able to update the device's firmware with new features and bug fixes after deployment. The firmware image update capability makes that possible.

Silicon Labs supports devices that do not use a bootloader, but this requires external hardware such as a Debug Adapter (Silicon Labs ISA3 or Wireless Starter Kit (WSTK)) or third-party SerialWire/JTAG programming device to update the firmware. Devices without a bootloader have no supported way of updating the firmware once they are deployed, which is why Silicon Labs strongly advocates implementing a bootloader.

In March of 2017, Silicon Labs introduced the Gecko Bootloader, a code library configurable through Simplicity Studio's IDE to generate bootloaders that can be used with a variety of Silicon Labs protocol stacks. The Gecko Bootloader can be used with EFM32 and EFR32 Series 1 and later devices. The Gecko Bootloader was restructured into a component-based design and released with Gecko SDK Suite (GSDK) 4.0 in December of 2021. This new version is documented in Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher (series 1 and 2 devices) or Silicon Labs Gecko Bootloader User's Guide for Series 3 and Higher, along with other documents. Documentation for older versions is installed with their respective SDKs.

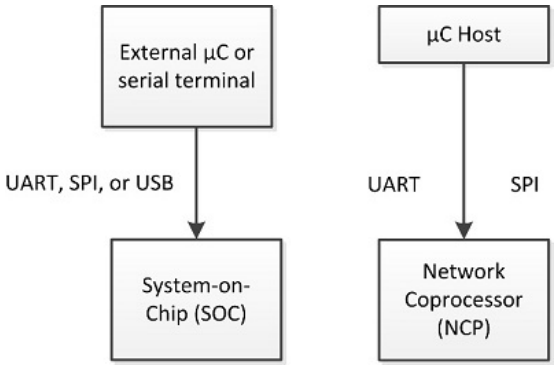
The Gecko Bootloader uses a customized update image file format. The update image file consumed by a Gecko Bootloader-generated application bootloader is a GBL (Gecko BootLoader) file, and is described in Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher (series 1 and 2 devices) or Silicon Labs Gecko Bootloader User's Guide for Series 3 and Higher.

Bootloading a firmware update image can be accomplished in two ways. The first is Over-The-Air (OTA), that is, through the wireless network, as shown in the following figure.



The second is through a hardwired link to the device. The following figure represents the serial bootloader use cases for SoCs (System on Chips) using either a UART (Universal Asynchronous Receiver/Transmitter), SPI (Serial Protocol Interface), or USB (Universal Serial Bus) interface, and for NCPs (Network Coprocessors) using either UART or SPI.

Copyright © 2025 Silicon Laboratories. All rights reserved.



Silicon Labs networking devices use bootloaders that perform firmware updates in two different modes: standalone (also called standalone bootloaders) and application (also called application bootloaders). Application bootloaders are further divided into those that use external storage for the download update image, and those that use local storage. These bootloader types are discussed in the next two sections.

The firmware update situations described in this document assume that the source node (the device sending the firmware image to the target through a serial or OTA link) acquires the new firmware through some other means. For example, if a device on the local Zigbee network has an Ethernet gateway attached, this device could get or receive these firmware updates over the Internet. This necessary part of the firmware update process is system-dependent and beyond the scope of this document.

Standalone Bootloader

A standalone bootloader is a program that uses an external communication interface, such as UART or SPI, to get an application image. Standalone firmware update is a single-stage process that allows the application image to be placed into flash memory, overwriting the existing application image, without the participation of the application itself. Very little interaction occurs between the standalone bootloader and the application running in flash. In general, the only time that the application interacts with the bootloader is when it requests a reboot into the bootloader. Once the bootloader is running, it receives firmware update packets containing the (new) firmware image either by physical connections such as UART or SPI, or by the radio (over-the-air).

When the firmware update process is initiated, the new code overwrites the existing stack and application code. If any errors occur during this process, the application cannot be recovered, and the process must start over. For information about configuring the Gecko Bootloader as a standalone bootloader, see Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher (series 1 and 2 devices) or Silicon Labs Gecko Bootloader User's Guide for Series 3 and Higher.

Application Bootloader

An application bootloader begins the firmware update process after the running application has completely downloaded the update image file. The application bootloader expects that the image either lives in external memory accessible by the bootloader or in a portion of main flash memory (if the chip has sufficient memory to support this local storage model).

The application bootloader relies on the application to acquire the new firmware image. This image can be downloaded by the application in any way that is convenient (UART, over-the-air, etc.) but it must be stored into a region referred to as the download space. The download space is typically an external memory device such as an EEPROM or dataflash, but it can also be a section of the chip's internal flash when using a local storage variant of the application bootloader. Once the new image has been stored, the application bootloader is then called to validate the new image and copy it from the download space to flash.

Since the application bootloader does not participate in acquiring the image, and the entire image is downloaded before the firmware update process is started, download errors do not adversely affect the running image. The download process can be restarted or paused to acquire the image over time. The integrity of the downloaded update image can be verified before initiating the firmware update process, to prevent a corrupt or non-functional image from being applied.

The Gecko Bootloader can be configured to accept a list of multiple upgrade images to attempt to verify and apply. This allows the Gecko Bootloader to store what is in effect a backup copy of the update image, which it can access if the first

Copyright © 2025 Silicon Laboratories. All rights reserved.

image is corrupt.

Note that the EmberZNet NCP platform does not utilize an application bootloader because the application code resides on the host rather than on the NCP directly. Instead a device acting as a serial coprocessor would utilize a standalone bootloader designed to accept code over the same serial interface as the expected NCP firmware uses. However, the host application (residing on a separate MCU from the NCP) can utilize whatever bootloading scheme is appropriate.

For more information on application bootloaders, see Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher (series 1 and 2 devices) or Silicon Labs Gecko Bootloader User's Guide for Series 3 and Higher.

Copyright © 2025 Silicon Laboratories. All rights reserved.

About the Gecko Bootloader

About the Gecko Bootloader

The Silicon Labs Gecko Bootloader is a configurable code library that can be used with all the newer Silicon Labs Gecko MCUs and wireless MCUs. It uses a specially-formatted update image file called a GBL file. The Gecko Bootloader has a two-stage design on Series 1 devices, where a minimal first stage bootloader is used to update the main bootloader. On Series 2 devices, the first stage bootloader is replaced by a Secure Engine and the Gecko Bootloader consists only of the main bootloader. The Secure Engine may be hardware-based, or virtual (software). If hardware-based, the implementation may be either with or without Secure Vault. Throughout this document, the following conventions will be used.

- **HSE**: Hardware Secure Engine, either with or without Secure Vault if not specified
- **VSE**: Virtual Secure Engine
- **SE**: Secure Engine (either HSE or VSE, in general)

Having a first stage bootloader or SE allows for field updates of the main bootloader, including adding new capabilities, changing communication protocols, adding new security features and fixes, and so on. The Gecko Bootloader consists of three component parts:

Core: The bootloader core contains the main function of both bootloader stages. It also contains functionality to write to the internal main flash, to perform a bootloader update, and to reset into the application flagging applicable reset reasons.

Driver: Different bootloading applications require different hardware drivers for use by the other components of the bootloader.

Component/Plugin: All parts of the main bootloader that are either optional or selectable for different configurations are implemented as components (in GSDK 4.0 and higher) or previously in plugins. Each component/plugin has a generic header file, and one or more implementations. The current release contains components for functionality like UART and SPI communication protocols, SPI flash storage, internal flash storage, and different cryptographic operations.

Features

Gecko Bootloader features include:

- Field-updatable
- Secure boot
- Signed GBL firmware update image file
- Encrypted GBL firmware update image file

These features are summarized in the following sections and described in more detail in Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher (series 1 and 2 devices) or Silicon Labs Gecko Bootloader User's Guide for Series 3 and Higher. Protocol-specific information about using the Gecko Bootloader may be found in the following documents:

- Using the Gecko Bootloader with EmberZNet
- *UG235.06: Bootloading and OTA with Silicon Labs Connect SDK v2.x*
- *UG435.06: Bootloading and OTA with Silicon Labs Connect SDK v3.x*
- Using the Gecko Bootloader with Silicon Labs Bluetooth Applications

Field-Updatable

Series 1

On EFM32 and EFR32 Series 1 devices, field update capability for the Gecko bootloader is provided by a two-stage design where the bootloader has a first stage and a main stage. The minimal first stage of the bootloader is not field updatable, and can only update the main bootloader by reading from and writing to fixed addresses in internal flash memory. To perform a main bootloader update, the running main bootloader verifies the integrity and authenticity of the bootloader update image, writes it to internal flash, and issues a reboot into the first stage bootloader. The first stage bootloader

Copyright © 2025 Silicon Laboratories. All rights reserved.

verifies the integrity of the main bootloader update image before copying it to the main bootloader location, completing the update

Series 2

On Series 2 devices, field update capability for the Gecko bootloader is provided by the SE. To perform a main bootloader update, the running main bootloader verifies the integrity and authenticity of the bootloader update image, writes it to internal flash, and requests that the SE installs the update. The SE optionally verifies the authenticity of the main bootloader update image before copying it to the main bootloader location, completing the update. The same mechanism can be used to update the SE itself.

Secure Boot

Secure boot is designed to prevent an untrusted application from running on the device. When Secure Boot is enabled, the bootloader enforces cryptographic signature verification of the application image on every boot using asymmetric cryptography. The signature algorithm used is ECDSA-P256-SHA256. The public key is written to the device during manufacturing, while the private key is kept secret. This ensures that the application was created and signed by a trusted party.

Signed GBL Update Image File

The Gecko Bootloader supports enforcing cryptographic signature verification of the update image file in addition to Secure Boot. This allows the bootloader and application to verify that the application or bootloader update comes from a trusted source before starting the update process. The signature algorithm used is ECDSA-P256-SHA256. The public key is the same key as for secure boot, written to the device during manufacturing, while the private key is never distributed. This ensures that the GBL file was created and signed by a trusted party.

Encrypted GBL Update File

The GBL update file can also be encrypted, to prevent eavesdroppers from getting hold of the plaintext firmware image. The encryption algorithm used is AES-CTR-128, and the encryption key is written to the device during manufacturing.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Memory Space for Bootloading

Memory Space for Bootloading

The first stage of the Gecko Bootloader on Series 1 devices takes up a single flash page. On devices with 2 kB flash pages, like EFR32MG1, this means that the first stage takes 2 kB.

The size of the main bootloader is dependent on the functionality required. With a typical bootloader configuration, the main bootloader for Series 1 devices takes up 14 kB of flash, bringing the total bootloader size to 16 kB.

Silicon Labs recommends reserving 16 kB for the bootloader for Series 1 and EFR32xG21 devices and 24 kB for EFR32xG22 devices.

On EFR32xG1 devices (Mighty Gecko, Flex Gecko, and Blue Gecko families), the bootloader resides in main flash.

- First stage bootloader @ 0x0
- Main bootloader @ 0x800
- Application @ 0x4000

On EFR32xG12 and later Series 1 devices, the bootloader resides in the bootloader area in the Information Block.

- Application @ 0x0
- First stage bootloader @ 0x0FE10000
- Main bootloader @ 0x0FE10800

On EFR32xG21, the main bootloader resides in main flash:

- Main bootloader @ 0x0
- Application @ 0x4000

On EFR32xG22, the main bootloader resides in main flash:

- Main bootloader @ 0x0
- Application @ 0x6000

On EFR32xG23, the main bootloader resides in main flash:

- Main bootloader @ 0x08000000
- Application @ 0x08006000

On EFR32xG24, the main bootloader resides in main flash:

- Main bootloader @ 0x08000000
- Application @ 0x08006000

Copyright © 2025 Silicon Laboratories. All rights reserved.

Design Decisions

Design Decisions

The decision of what bootloader type to deploy depends on many factors. Note that the platform type and available flash memory may limit bootloader choices.

Some questions related to this are:

- Where does the device get the new update image? Is this over-the-air via the networking protocol? Using a separate interface connected to the Internet?
- Will the device have an external memory chip to store a new update image? If not, is there enough internal flash memory to store both a current and a newly downloaded copy of the largest expected application image?
- If the device receives the new image over-the-air, will it be multiple hops away from the server holding the download image?
- What kind of image security is needed?
- What communications driver will be used (in the single protocol case)?
- Does the use case require more than one protocol?

The configurable design of the Gecko Bootloader platform means that developers can create bootloaders to fit almost any design choice. See Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher (series 1 and 2 devices) or Silicon Labs Gecko Bootloader User's Guide for Series 3 and Higher.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Using the Gecko Bootloader with Silicon Labs Connect

Using the Gecko Bootloader with Silicon Labs Connect

NOTE: This section replaces *AN1085: Using the Gecko Bootloader with Silicon Labs Connect*. Further updates to this application note will be provided here.

This application note includes detailed information on using the Silicon Labs Gecko Bootloader with the Silicon Labs Connect stack, part of the Silicon Labs Flex SDK (Software Development Kit). It supplements the general Gecko Bootloader implementation information provided in *UG266: Silicon Labs Gecko Bootloader User's Guide*. If you are not familiar with the basic principles of performing a firmware upgrade or want more information about upgrade image files, refer to *UG103.6: Bootloading Fundamentals*.

The *Connect User's Guide* is a series of documents that provides in-depth information for developers who are using the Silicon Labs Connect Stack for their application development. Refer to *UG435.06: Bootloading and OTA with Silicon Labs Connect* to learn about the bootloader options (standalone, application, and Over the Air (OTA)) available for use within Connect-based applications.

Proprietary is supported on all EFR32FG devices. For others, check the device's data sheet under Ordering Information > Protocol Stack to see if Proprietary is supported. In Proprietary SDK version 2.7.n, Connect is not supported on EFR32xG22

Copyright © 2025 Silicon Laboratories. All rights reserved.

Overview

Overview

The Silicon Labs Gecko Bootloader is a common bootloader for all the newer MCUs and wireless MCUs from Silicon Labs. The Gecko Bootloader can be configured to perform a variety of bootload functions, from device initialization to firmware upgrades. The Gecko Bootloader uses a proprietary format for its upgrade images, called GBL (Gecko Bootloader). These images are produced with the file extension ".gbl". Additional information on the GBL file format is provided in *UG103.6: Bootloading Fundamentals*.

The Gecko Bootloader has a two-stage design, where a minimal first stage bootloader is used to upgrade the main bootloader. The first stage bootloader only contains functionality to read from and write to fixed addresses in internal flash. To perform a main bootloader upgrade, the running main bootloader verifies the integrity and authenticity of the bootloader upgrade image file. The running main bootloader then writes the upgrade image to a fixed location in flash and issues a reboot into the first stage bootloader. The first stage bootloader verifies the integrity of the main bootloader firmware upgrade image, by computing a CRC32 checksum before copying the upgrade image to the main bootloader location.

The Gecko Bootloader can be configured to perform firmware upgrades in standalone mode (also called a standalone bootloader) or in application mode (also called an application bootloader), depending on the component configuration. Components can be enabled and configured through the Simplicity Studio IDE.

A standalone bootloader uses a communications channel to get a firmware upgrade image. NCP (network co-processor) devices always use standalone bootloaders. Standalone bootloaders perform firmware image upgrades in a single-stage process that allows the application image to be placed into flash memory, overwriting the existing application image, without the participation of the application itself. In general, the only time that the application interacts with a standalone bootloader is when it requests to reboot into the bootloader. Once the bootloader is running, it receives packets containing the firmware upgrade image by a physical connection such as UART or SPI. To function as a standalone bootloader, a component providing a communication interface such as UART or SPI must be configured.

An application bootloader relies on the application to acquire the firmware upgrade image. The application bootloader performs a firmware image upgrade by reprogramming the device's flash with the firmware upgrade image stored in a region of flash memory referred to as the download space. The application transfers the firmware upgrade image to the download space in any way that is convenient (UART, over-the-air, and so on). The download space is either an external memory device such as an EEPROM or dataflash or a section of the chip's internal flash. The Gecko Bootloader can partition the download space into multiple storage slots, and store multiple firmware upgrade images simultaneously. To function as an application bootloader, a component providing a bootloader storage implementation has to be configured.

OTA (over the air) is often confused with application bootloader, since application bootloader can be used to flash an image that was distributed OTA. However, OTA is an application feature. For example, an NCP gateway device typically uses the standalone bootloader, so the host can update it, but the Host-NCP application can work as an OTA server.

This document describes how to use both standalone and application models with Silicon Labs Connect.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Using the Gecko Standalone Bootloaders

Using the Gecko Standalone Bootloaders

A Gecko Bootloader-based standalone bootloader receives an application image onto a target device by serial transfer via SPI or UART. If using UART, you can establish a serial connection between a source device and a target device's serial interface and upload a new software image to it using the XModem protocol. If you need information on the XModem protocol, a good place to start is http://en.wikipedia.org/wiki/XMODEM, which should have a brief description and up-to-date links to protocol documentation.

Performing a Serial Upload – UART XMODEM Bootloader

Serial upload can be performed with any source device that provides the expected serial interface method. This can be a Windows-based PC, a Linux or Mac OS-based device, or an embedded MCU with no operating system. UART transfer can be done with a third-party serial terminal program like TeraTerm on Windows or Linux minicom or with user-compiled host code. However, drivers for SPI Master or UART may vary with operating systems, and serial terminal programs may vary in timing and performance, so if you are unsure about what driver or program to use on your source code, please consult Silicon Labs technical support.

To open a serial connection over UART, the source device connects to the target device at 115,200 baud, 8 data bits, no parity bit, and 1 stop bit (8-N-1), with no flow control by default. These options may be changed using the component options in the Bootloader project.

> **Note**: The UART-based serial bootloader configurations do not employ any flow control in the communication channel by default, because the XModem protocol used for image transfer already has built-in flow control mechanisms. However, Silicon Labs' normally-supplied NCP firmware does utilize either hardware-based (RTS/CTS) or software-based (XON/XOFF) flow control, so a host device must take care to temporarily disable the flow control when placing its NCP into serial bootloading mode. Alternatively, application designers can change the options in the provided bootloader project and customize the serial bootloader's handling of the UART to add hardware flow control at their discretion.

Once the connection with a UART-based serial bootloader is established:

1. The target device's bootloader sends output over its serial port after it receives a carriage return from the source device at the expected baud rate. This prevents the bootloader from prematurely sending commands that might be misinterpreted by other devices that are connected to the serial port. Note that serial bootloaders typically don't enforce any timeout when awaiting the initial serial handshake via carriage return, so the bootloader will wait indefinitely in this mode until guided by the source device or until the chip is reset.
2. After the bootloader receives a carriage return from the target device, it displays a menu with the following ASCII-based output, (where X.Y.A corresponds to the major, minor, and sub-minor fields of the bootloader version number, respectively):

```
Gecko Bootloader vX.Y.A
1. upload gbl
2. run
3. ebl info
BL >
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

> **Note**:The third menu option has no effect. While current menu options should remain functionally unchanged, the menu title and options text is liable to change, and new options might be added.

After listing the menu options, the bootloader's “BL >” prompt displays, and the ASCII character corresponding to the number of each option can then be entered by the source to select the described action, such as ‘2’ (ASCII code 0x32) to run the firmware presently loaded in the application area. Here again, no timeout is enforced by the bootloader, so it will wait indefinitely until a character is received or the chip is reset. Note that while the menu interface is designed for human interaction, the transfer can still be performed programmatically or through a scripted interface, provided the source device sends the expected ASCII characters to the target at appropriate times.

> **Note**: Scripts that interact with the bootloader should use only the “BL >” prompt to determine when the bootloader is ready for input.

Selecting menu option 1 initiates upload of a new software image to the target device, which unfolds as follows:

1. The target device awaits an XModem CRC upload of a GBL file over the expected serial interface, as indicated by the stream of C characters that its bootloader transmits.
2. If no transaction is initiated within 60 seconds, the bootloader times out and returns to the menu.
3. Once uploading begins (first XModem SOH data packet received), the bootloader expects each successive XModem SOH packet within 1 second, or else a timeout error will be generated and the session will abort.
4. After an image successfully uploads, the XModem transaction completes and the bootloader displays ‘Serial upload complete’ before redisplaying the menu.

Errors and Status Codes

If an error occurs during the upload, the UART serial bootloader displays the message ‘Serial upload aborted,’ followed by a more detailed message and a hex error code. Some of the more common errors are shown in the following table. The UART serial bootloader then redisplays the bootloader menu.

The following tables describe the normal status codes, error conditions, and special characters or enumerations used by the Gecko Bootloader. For additional status codes, see the Gecko Bootloader API documentation installed with your SDK in the platform/bootloader/documentation folder.

Serial Uploading Statuses and Error Messages

Hex code	Constant	Description
0x00	BL_SUCCESS	Default success status.
0x01	BL_ERR	General error processing packet.
0x1C	BLOCK_TIMEOUT	The bootloader timed out waiting for some part of the XModem frame.
0x21	BLOCKERR_SOH	The bootloader did not find the expected start of header (SOH) character at the beginning of the XModem frame.
0x22	BLOCKERR_CHK	The bootloader detected the sequence check byte of the XModem frame was not the inverse of the sequence byte.
0x23	BLOCKERR_CRCH	The bootloader encountered an error while comparing the high bytes of the received and calculated CRCx of the XModem frame.
0x24	BLOCKERR_CRCL	The bootloader encountered an error while comparing the low bytes of the received and calculated CRCs of the XModem frame.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Hex code	Constant	Description
0x25	BLOCKERR_SEQUENCE	The bootloader did not receive the expected sequence number in the current XModem frame.
0x26	BLOCKERR_PARTIAL	The frame that the bootloader was trying to parse was deemed incomplete (some bytes missing or lost).
0x27	BLOCKERR_DUPLICATE	The bootloader encountered a duplicate of the previous XModem frame.
0x40	BL_ERR_MASK	Bitmask for any bootloader error codes returned in CAN or NAK frame.
0x41	BL_ERR_HEADER_EXP	No GBL header was received when expected.
0x42	BL_ERR_HEADER_WRITE_CRC	Failed to write header or CRC.
0x43	BL_ERR_CRC	File or written image failed CRC check.
0x44	BL_ERR_UNKNOWN_TAG	Unknown tag detected in GBL image.
0x45	BL_ERR_SIG	Invalid GBL header contents.
0x46	BL_ERR_ODD_LEN	Trying to flash odd number of bytes.
0x47	BL_ERR_BLOCK_INDEX	Indexed past end of block buffer.
0x48	BL_ERR_OVWR_BL	Attempt to overwrite bootloader flash.
0x49	BL_ERR_OVWR_SIMEE	Attempt to overwrite SIMEE flash.
0x4A	BL_ERR_ERASE_FAIL	Flash erase failed.
0x4B	BL_ERR_WRITE_FAIL	Flash write failed.
0x4C	BL_ERR_CRC_LEN	End tag CRC wrong length.
0x4D	BL_ERR_NO_QUERY	Received data before query request/response.
0x4E	BL_ERR_BAD_LEN	An invalid length was detected in the upgrade image file.
0x4F	BL_ERR_TAGBUF	Insufficient tag buffer size or an invalid length was found in the GBL image.

Special Characters Used in Packet Types

Hex code	Constant	Description
0x01	SOH	Start of Header.
0x03	CTRL_C	Cancel (from sender).
0x04	EOT	End of Transmission.
0x06	ACK	Acknowledged.
0x15	NAK	Not acknowledged.
0x18	CAN	Cancel
0x43	C	ASCII ‘C’.
0x51	QUERY	ASCII ‘Q’.
0x52	QRESP	ASCII ‘R’.

Status Codes Returned in a Synchronous Response

Hex code	Constant	Description
0x16	TIMEOUT	Bootloader timed out expecting characters.
0x17	FILEDONE	EOT process successfully.
0x18	FILEABORT	Transfer aborted prematurely.
0x19	BLOCKOK	Data block processed OK.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Hex code	Constant	Description
0x1A	QUERYFOUND	Successful query.

Running the Application Image

For standalone bootloader variants that utilize an interactive menu, bootloader menu option 2 (run) resets the target device into the uploaded application image. If no application image is present, or an error occurred during a previous upload, the bootloader returns to the menu. For SPI-based variants, which don't use a menu, the application starts up immediately upon ACKing the EOT frame from the source device.

Performing a Bootloader Upgrade

To perform a bootloader upgrade with the standalone bootloaders, simply transmit two GBL files in succession. The first GBL file should contain a main bootloader upgrade image. After upload is completed, the device resets to upgrade the main bootloader. For standalone bootloader variants that utilize an interactive menu, bootloader menu option 2 (run) resets the target device into the first stage bootloader to perform the upgrade, before returning to the upgraded main bootloader. The second GBL file, containing an application upgrade image, can then be uploaded.

Upload Recovery

If an image upload fails, the target node is left without a valid application image. The standalone bootloader will re-enter firmware upgrade mode if it determines that the application image isn't valid. However, the application image may have a valid structure, but contain a bug preventing normal operation. Regardless of the serial interface supported by your standalone bootloader, a GPIO-based trigger can be used to facilitate recovery via serial upload.

You can configure your standalone bootloader to use a software-based GPIO pin check or other schemes of recovery mode activation such as button recovery by configuring the GPIO Activation component.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Using the Gecko Application Bootloader

Acquiring a New Image

The application bootloader relies on application code to obtain new code images. The bootloader itself only knows how to read a GBL image stored in the download space and copy the relevant portions to the main flash block. This approach means that the application developer is free to acquire the new code image in any way that makes sense (serial, OTA, and so on).

Typically, application developers choose to acquire the new code image over-the-air (OTA) since this is readily available on all devices. The acquisition of a new image in Connect is covered extensively in the Connect Users Guide, UG435.

Performing an Application Upgrade

The application and the bootloader have limited indirect contact. Their only interaction is through passing non-volatile data across module reboots.

Once the application decides to install a new image saved in the download space it calls the Ember HAL API `halAppBootloaderInstallNewImage()`. This call indicates to the bootloader that it should attempt to perform a firmware upgrade from storage slot 0, and reboots the device. This API is backwards-compatible with the legacy Ember bootloader. If performing a firmware upgrade from a different storage slot than slot 0 is desired, the Gecko Bootloader API should be used instead, by calling `bootloader_setBootloadList()`. If the bootloader fails to install the new image, it sets the reset cause to `RESET_BOOTLOADER_BADIMAGE` and resets the module. Upon startup, the application should read the reset cause with `halGetExtendedResetInfo()`. If the reset cause is set to `RESET_BOOTLOADER_BADIMAGE`, the application knows the install process failed and can attempt to obtain a new image. A printable error string can be acquired from calling `halGetExtendedResetString()`. Under normal circumstances, the application bootloader does not print anything on the serial line.

External Storage Application Bootloader

See *UG266: Silicon Labs Gecko Bootloader User's Guide* for information about memory configuration for external storage bootloaders.

Application bootloaders typically use an external device to store the downloaded application image. This device can be accessed over either an I2C or SPI serial interface. Refer to *UG266: Silicon Labs Gecko Bootloader User's Guide* for a list of supported Dataflash/EEPROM devices. It is important to select a device whose size is at least the size of your flash in order to fit the application being bootloaded.

The default recommendation for external SPI serial flash is the MX25R, as this is available in standard and smaller packages, is supported by standard drivers, and has very low software-enabled sleep current without the need for an external shutdown control circuit. Most radio boards from Silicon Labs are populated with the MX25R8035F for evaluation purposes. In general, customers should use parts that have low sleep current, don't require external shutdown control circuitry, and have software shutdown control. When using components with high idle/sleep current and no software shutdown control, an external shutdown control circuit is recommended to reduce sleep current.

For EFR32xG1, only the serial dataflash option is available; no internal storage application bootloader is presently offered.

> **Note**: Some of these chips have compatible pinouts with others, but there are several incompatible variations. Contact Silicon Labs for details on connecting I2C or other SPI dataflash chips to an EFR32.

Copyright © 2025 Silicon Laboratories. All rights reserved.

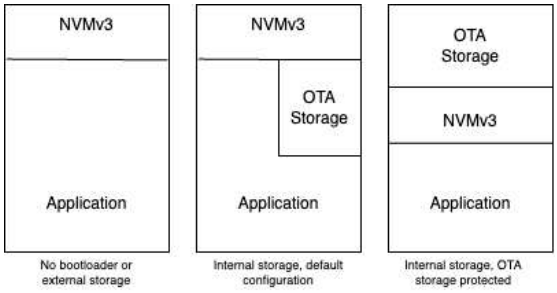
Read-Modify-Write pertains to a feature of certain dataflash chips that their corresponding driver exposes, and that is exploited by the bootloader library. Chips without this feature require a page erase to be performed before writing to that page, which precludes random-access writes by an application.

Internal Storage Application Bootloader

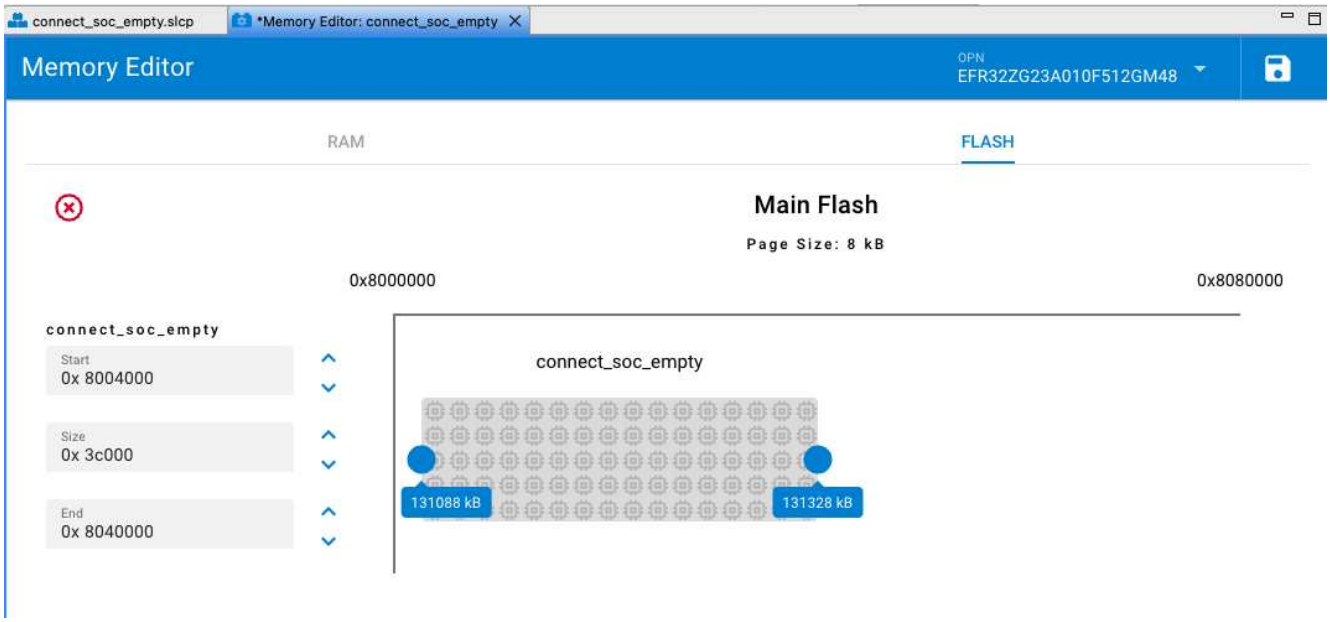
The internal storage bootloader is essentially an application bootloader with a data flash driver that uses a portion of the on-chip flash for image storage instead of an external storage chip. See *UG266: Gecko Bootloader User's Guide* for information about memory configuration for internal storage.

If you don't modify the flash layout of your application, the NVMv3 token storage is at the end of the flash memory. By default, our internal storage bootloader sets up a slot that protects at least the default 36k of token storage (48k on some parts). This configuration however has some drawbacks:

- You might want to optimize your storage space, and use a smaller NVMv3 token storage, in which case the bootloader protects more than what it needs to.
- Apart from the size outputs of the compiler, you won't get notified if you run out of flash space. If you program the device directly, not through the bootloader, you are allowed to overlap your application with OTA storage, which will cause failure the first time you erase the OTA storage (depicted in the middle below)

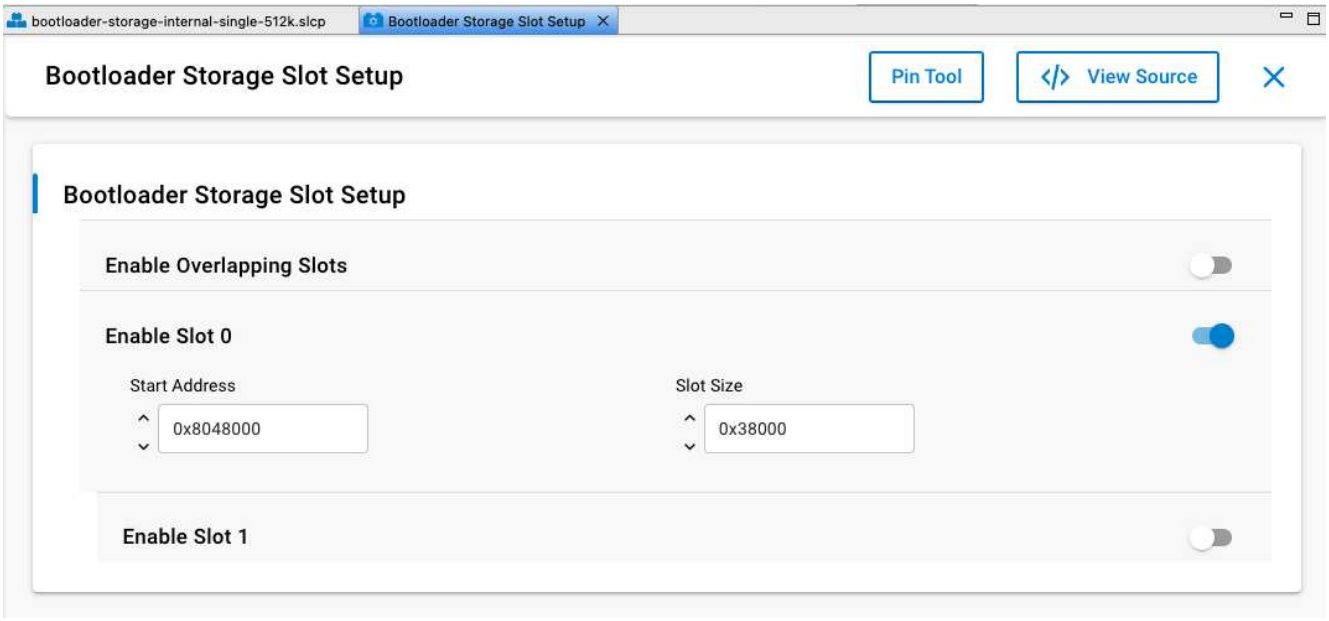


To avoid these problems, you can modify your main application's memory allocation in the memory editor. For example, a typical application will use half, or a bit more than half, of the flash for the application and NVMv3, while the remainder can be used as OTA storage. The OTA storage can be smaller, because it doesn't include the NVMv3 and because bootloader can be configured to use compressed images.



Copyright © 2025 Silicon Laboratories. All rights reserved.

Finally, you should configure the bootloader project's **Bootloader Storage Slot Setup** component to utilize the space you protected from your application:



> **Note**: On EFR32xG1 and Series 2 devices, the application is offset by 16 kB to accommodate the bootloader at the bottom of main flash. On other EFR32 Series 1 devices, the Gecko Bootloader resides in the Bootloader flash region outside of main flash block.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Gecko Bootloader User's Guide for Series 3 and Higher

Silicon Labs Gecko Bootloader User's Guide for Series 3 and Higher

This guide describes the high-level implementation of the Silicon Labs Gecko Bootloader for Series 3 microcontrollers, SoCs (System on Chips), and NCPs (Network Co-Processors), and provides information on different aspects of configuring the Gecko Bootloader.

The Silicon Labs Gecko Bootloader is a common bootloader for all the newer MCUs and wireless MCUs from Silicon Labs. This guide is specific to Series 3 devices and above. The Gecko Bootloader can be configured to perform a variety of functions, from device initialization to firmware upgrades. Key features of the bootloader are:

- Usable across Silicon Labs Gecko microcontroller and wireless microcontroller families
- In-field upgradeable
- Configurable
- Enhanced security features, including:
- **Secure Boot**: When Secure Boot is enabled, the bootloader enforces cryptographic signature verification of the application image on every boot, using asymmetric cryptography. This ensures that the application was created and signed by a trusted party.
- **Signed upgrade image file**: The Gecko Bootloader supports enforcing cryptographic signature verification of the upgrade image file. This allows the bootloader and application to verify that the application or bootloader upgrade comes from a trusted source before starting the upgrade process, ensuring that the image file was created and signed by a trusted party.
- **Encrypted upgrade image**: The image file can also be encrypted to prevent eavesdroppers from acquiring the plaintext firmware image.

For Series 3 devices and above, Gecko Bootloader uses a proprietary format for its upgrade images, called GBL V4 (Gecko Bootloader file). These files have the file extension ".gbl". See the *Gecko Bootloader File Format* page for more details. Series 3 and above devices do not support GBL versions lesser than v4.

On Series 3 devices, the Gecko bootloader consist of the main stage bootloader. The main bootloader is upgradable through the Secure Engine.

The Secure Engine provides functionality to install an image to code region 0 in the external flash.

To perform a bootloader upgrade, the running bootloader verifies the integrity and authenticity of the bootloader upgrade image file. The bootloader then writes the upgrade image to the upgrade location in external flash and requests that the Secure Engine installs it. On some devices, the Secure Engine is also capable of verifying the authenticity of the main bootloader update image against a root of trust. The Secure Engine itself is upgradable using the same mechanism. See the *Gecko Bootloader Operation - Secure Engine Upgrade* page for more details.

The bootloader consists of a common core, drivers, and a set of components that give the bootloader specific capabilities. The common bootloader core is provided as a full-source delivery. The common bootloader core contains functionality to parse GBL v4 files and flash their contents to the device.

The Gecko Bootloader can be configured to perform firmware upgrades in standalone mode (also called a standalone bootloader) or in application mode (also called an application bootloader), depending on the component configuration. Components can be installed in and configured through the Simplicity Studio IDE.

A standalone bootloader uses a communications channel to get a firmware upgrade image. NCP (network co-processor) devices always use standalone bootloaders. Standalone bootloaders perform firmware image upgrades in a single-stage process that allows the application image to be placed into flash memory, overwriting the existing application image, without the participation of the application itself. In general, the only time that the application interacts with a standalone bootloader is when it requests to reboot into the bootloader. Once the bootloader is running, it receives packets containing

Copyright © 2025 Silicon Laboratories. All rights reserved.

the firmware upgrade image by a physical connection such as UART or SPI. To function as a standalone bootloader with a physical connection, a component providing a communication interface such as UART or SPI must be configured.

An application bootloader relies on the application to acquire the firmware upgrade image. The application bootloader performs a firmware image upgrade by writing the firmware upgrade image to a region of flash memory referred to as the download space. The application transfers the firmware upgrade image to the download space in any way that is convenient (UART, over-the-air, Ethernet, USB, and so on). The download space is data flash or a section of the device's flash. The Gecko Bootloader can partition the download space into multiple storage slots and store multiple firmware upgrade images simultaneously. To function as an application bootloader, a component providing a bootloader storage implementation must be configured.

Silicon Labs provides example bootloaders that come with a preconfigured set of installed components for configuration in either standalone or application mode. See the *Configuring the Gecko Bootloader* page. The following sections provide an overview of the Gecko Bootloader common core, drivers, and components. For details, including details on error codes and conditions, see the Gecko Bootloader API Reference, shipped with the SDK in the platform/bootloader/documentation folder.

Core

The bootloader core contains the bootloader's main functions. It also contains functionality to write to the external flash, an image parser to parse and act upon the contents of GBL v4 upgrade files, and functionality to boot the application in main flash.

A version of the GBL v4 image parser without support for encrypted upgrade images is also available. This version can be used in flash space constrained bootloader applications where encryption of the upgrade image is not required.

Shared Memory

To exchange information between the bootloader and application, a section of SRAM is used. The contents of SRAM are preserved through a software reset, making SRAM suitable as a communication channel between bootloader and application.

The shared memory has a size of 4 bytes, and is located at the first address of SRAM, 0x20000000. It is used to store a single word containing the reason for a reset. The structure of the reset cause word is defined in the Reset Information part of the Application Interface, in the file **btl_reset_info.h**, as 16 bits containing the reason, and 16 bits of signature indicating if the word is valid or not. If the signature reads 0xF00F, the reset reason is valid.

All 16-bit reset reasons used by Silicon Labs have the most significant bit set to zero. If custom reset reasons are desired, it is recommended to set the most significant bit to avoid conflicting definitions.

Drivers

Different applications for firmware upgrade require different hardware drivers for use by the other components of the bootloader.

Driver modules include:

- Delay: Simple delay routines for use with components that require small delays or timeouts.
- SPI Slave: Flexible SPI Slave driver implementation for use in communication components implementing SPI protocols. This driver supports both blocking and non-blocking operation, with DMA (Direct Memory Access) backing the background transfers to support non- blocking operation.
- UART: Flexible serial UART driver implementation for use in communication components implementing UART protocols. This driver supports both blocking and non-blocking operations, with DMA backing the background transfers to support non-blocking operation. Additionally, support for hardware flow control (RTS/CTS) is included.

Components

Copyright © 2025 Silicon Laboratories. All rights reserved.

All parts of the bootloader that are either optional or that may be exchanged for different configurations are implemented as components. Each component may have a configuration header file, and one or more implementations. Components include:

- Communication
- UART: XMODEM
- UART: BGAPI
- Compression
- Debug
- GPIO Activation
- Security
- Storage

Communication

Communication components provide an interface for implementing communication with a host device, such as a computer or a micro- controller. Several components implement the communication interface, using different transports and protocols.

- BGAPI UART DFU: By enabling the BGAPI communication component, the bootloader communication interface implements the UART DFU protocol using BGAPI commands.
- UART XMODEM: By enabling the UART XMODEM communication component, the bootloader communication interface implements the XMODEM-CRC protocol over UART. This component makes the bootloader compatible with the legacy serial-uart-bootloader that was previously released with the EmberZNet wireless stack. See *AN760: Using the Ember Standalone Bootloader* for more information about legacy Ember standalone bootloaders.

Compression

Compression components provide capability for the bootloader GBL file parser to handle compressed GBL v4 upgrade images. Each compression component provides support for one (de)compression algorithm. At the time of writing, decompression of data compressed with the LZ4 and LZMA algorithms is supported, through the *GBL Compression (LZ4)* and *GBL Compression (LZMA)* components.

Debug

This component provides the bootloader with support for debugging output. If the component is configured to enable debug prints, short debug messages will be printed over Serial Wire Output (SWO), which can be accessed in multiple ways, including using Simplicity Commander, and by connecting to port 4900 of the Wireless Starter Kit TCP/IP interface.

To turn on debug prints, enable the Debug component and select **Debug prints**. Select **Debug asserts** to enable assertions in the source code.

GPIO Activation

This component provides functionality to enter firmware upgrade mode automatically after reset if a GPIO pin is active during boot. The GPIO pin location and polarity are configurable.

- GPIO: By enabling the GPIO activation component, the firmware upgrade mode can be activated by push buttons.

Security

Security components provide implementations of cryptographic operations as well as functionality to compute checksums and to read cryptographic keys from manufacturing tokens.

Modules include:

- AES: AES decryption functionality
- CRC16: CRC16 functionality
- CRC32: CRC32 functionality
- ECDSA: ECDSA signature verification functionality
- SHA-256: SHA-256 digest functionality

Storage

Copyright © 2025 Silicon Laboratories. All rights reserved.

These components provide the bootloader with multiple storage options for SoCs. All storage implementations must provide an API to access image files to be upgraded. This API is based on the concept of dividing the download space into storage slots, where each slot has a predefined size and location in memory and can be used to store a single upgrade image. Some storage implementations also support a raw storage API to access the underlying storage medium. This can be used by applications to store other data in parts of the storage medium that are not used for storing firmware upgrade images. Implementations include:

- **External Flash**: The external flash storage implementation uses the external flash of the device for upgrade image storage. Note that this storage area is only a download space and is separate from the portion of external flash used to hold the active application code.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Gecko Bootloader File Format v4

Gecko Bootloader File Format v4

The Gecko Bootloader supports GBL File format v4 for Series 3 device. This format has significant differences as compared to version 3, The file formats described in this section are generated by Simplicity Commander commands.

File Structures

The GBL v4 file format is composed of nested TLV (Type-Length-Value) elements. The format of the TLV is as follows:

Field	Length (Bytes)	Description
Type	4	Type indicates the content of the Value field
Length	4	Length of the Value field in bytes
Value	N	Length size field containing either more TLVs or data

There are two types of TLVs, data and container. Data TLVs value field contains data whose format is specified in the type field and the container TLVs value filed contains more TLVs, which in turn again may contain data or container TLVs.

TLV Descriptions

This section describes each of the TLVs.

GBLV4

This mandatory TLV is used to identify the GBLv4 file when parsing and tell the length of the file.

Type	Length	Value
0x84A617EB	Variable	Contains all the TLVs for the file

MANIFEST

MANIFEST TLV is present inside the GBLV4 TLV. It has a collection, TLVs, that describes the metadata of the GBLv4 file that is used during the update process.

Type	Length	Value
0xAA01012A	Variable	Contains metadata TLVs for applying the file.

MANIFEST_INFO

The MANIFEST_INFO TVL contains the `ManifestInfo_t` structure that describes the version dependency and the features of the GBL file.

Type	Length	Value
0x2B03032B	Variable	manifest_info structure

The `ManifestInfo_t` structure is as follows:

Copyright © 2025 Silicon Laboratories. All rights reserved.

Data Type	Field	Purpose
uint32_t	version	Minimum version of the GLBv4 parser required, used for future extensions.
uint32_t	features	Features requested from the parser (encryption, compression etc.)

BUNDLE_VERSION

This is the child TVL of MANIFEST TLV, required to identify the bundle/file and is used for versioning and rejection of downgrades.

Type	Length	Value
0x2B04042B	Fixed	bundle_version structure

The `BundleVersion_t` structure is as follows:

Data Type	Field	Purpose
Uint8_t	productId	128 bit unique product ID
uint32_t	bundleVersion	Reject if smaller than this exists
Uint32_t	minVersion	The minimum previous version to which this update file can be applied

CONTENT_HASH

An optional TLV that provides integrity checking on the remainder of the file. This TLV is applicable to batch updates only.

Type	Length	Value
0x0x2B05052BUL	Fixed	content_hash structure

The `content_hash` structure is as follows:

Data Type	Field	Purpose
uint32_t	hashType	The hash function used to validate (1=SHA, etc.)
uint8_t	sha[]	The hash of all contents following the manifest

MANIFEST_CERTIFICATE

Type	Length	Value
0x2B01012B	Fixed	ApplicationCertificate_t

The `ApplicationCertificate_t` structure is as follows:

Data Type	Field	Purpose
uint8_t	structVersion	Vesion of the certificate structure
uint8_t	flags	Reserved flags
uint8_t	key	Public key used to verify GBL file
uint32_t	version	Version number of this certificate
uint8_t	signature	The signature of the certificate itself (not the GBL file)

MANIFEST_SIGNATURE

Type	Length	Value
0x2B02022BUL	Variable	Manifest signature structure

Copyright © 2025 Silicon Laboratories. All rights reserved.

The `Manifest_signature` structure is as follows:

Data Type	Field	Purpose
uint32_t	Signature function	The function used for signing
uint8_t[]	Signature	The signature for authentication

UPDATE_PROCESS

This is a collection of TLVs that describes the update process. Parsing this TLV does not happen until the MANIFEST_TLV is authenticated. Each TLV contained in this is parsed and processed one at a time in the order they are listed.

Type	Length	Value
0xAB06062BUL	Variable	TLVs describing the steps to update the system

UPDATE_SE

This is an optional TLV that is used to update the SE.

Type	Length	Value
0x2C02022CUL	Fixed	UpdateSe_t structure

The `UpdateSe_t` structure is as follows:

Data Type	Field	Purpose
uint32_t	version	Version number of the SE update image.
uint32_t	tlvPosition	The absolute position of the SE update image Blob TLV

UPDATE_MEMORY_SECTION

The memory section's content is present in the file and needs to be updated.

Type	Length	Value
0x2C03032CUL	Fixed	UpdateMemorySection_t structure

The `UpdateMemorySection_t` structure is as follows:

Data Type	Field	Purpose
uint8_t	targetMemory	Which memory this definition is for (0=main memory, 0x1-0x7f = reserved, 0x80-0xff=defined by the user)
uint8_t	plainImageSize	Represents the size of the plain image in 3 bytes
uint32_t	targetAddr	The target memory address for the update
uint32_t	type	Bitfield representing the type of application
uint32_t	version	Version number for this application (customer-defined)
uint32_t	capabilities	Bitfield representing the capabilities of this application
uint32_t	memorySectionPos	The absolute position of the MEMORY_SECTION_TLV in the file
HashVaule_t	memSecHash	A hash value for verifying the integrity of the memory section

MANIFEST_FINISH

TLV to indicate the end of an update.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Length	Value
0x2C04042CUL	Fixed	Empty

MEMORY_SECTION

This TLV contains TLVs for a memory section update.

Type	Length	Value
0xBA01013AUL	Variable	TLVs for the region

MEMORY_SECTION_INFO

This TLV contains information about how the blob data is read and where the data is to be written to. This is a common wrapper for all data as it allows the data to be compressed, verified and processed in a unified manner.

The data can be divided into blocks, with each block having its own hash. This can then be compared with the hashes in the memory section info to authenticate the block. This is useful during streaming updates to avoid writing unauthenticated data to flash.

Type	Length	Value
0x3B01013BUL	Fixed	MemSectionPartialInfo_t

The `MemSectionPartialInfo_t` structure is as follows:

Data Type	Field	Purpose
uint8_t	compressionScheme	0 = No compression 1 = LZ4 compression 2 = LMZA compression
uint8_t	encryptionScheme	0 = No encryption 1 = AES CCM
uint8_t	secureBootScheme	Reserved for future use.
uint8_t	reserved	
uint16_t	signBlockSize	Block size if DFU authentication is done for each block.
uint16_t	numBlocks	Number of blocks inside the BLOB.
uint8_t[]	nonce	Nonce for encryption
uint8_t[]	finalImageHash	Reserved for future use.
uint8_t[]	secureBootSign	Reserved for future use.

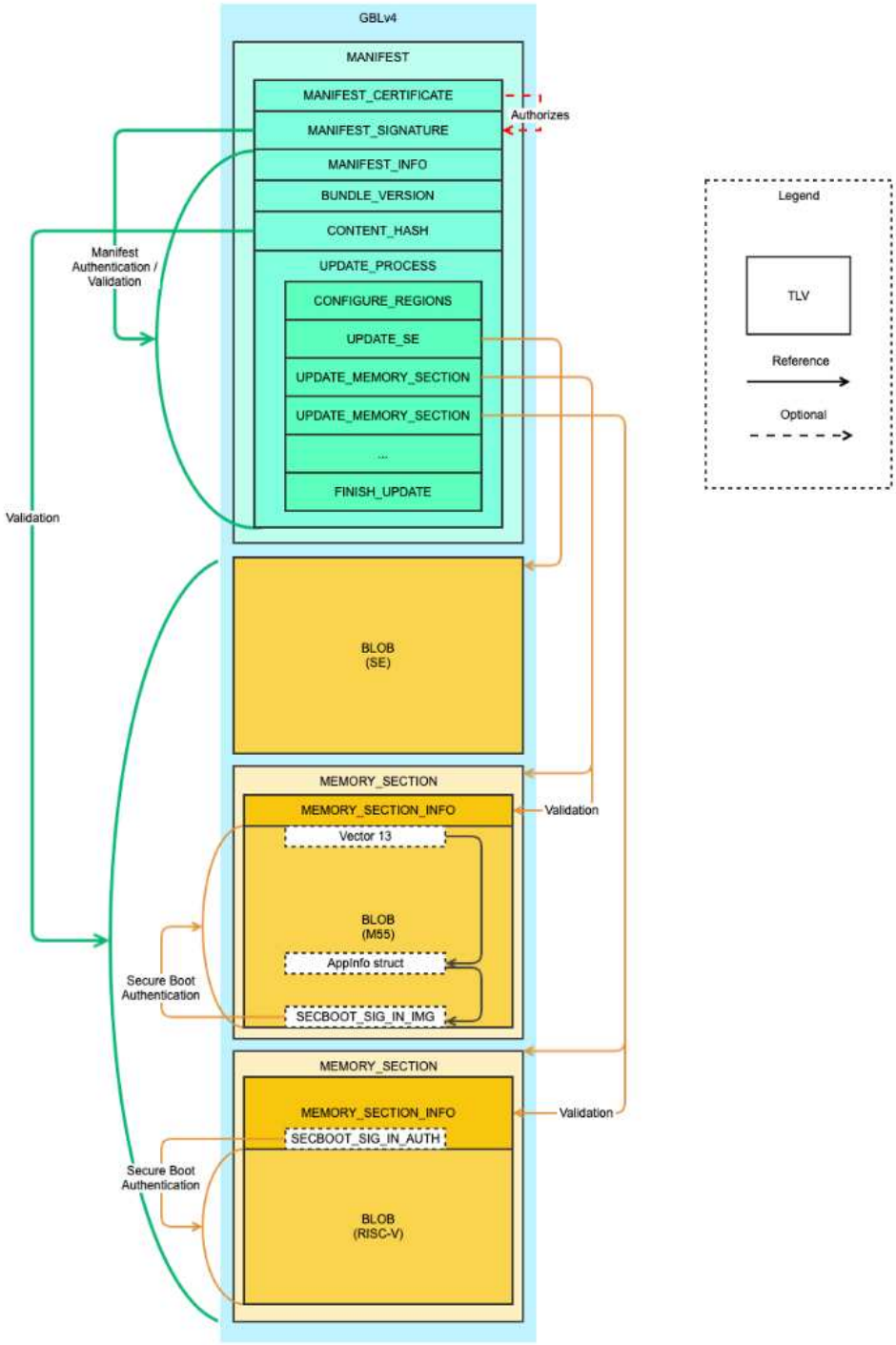
BLOB

This TLV contains variable length binary data.

Type	Length	Value
0x3B02023BUL	Variable	Binary Data

The figure below shows the structure of a GBL v4 file.

Copyright © 2025 Silicon Laboratories. All rights reserved.



Copyright © 2025 Silicon Laboratories. All rights reserved.

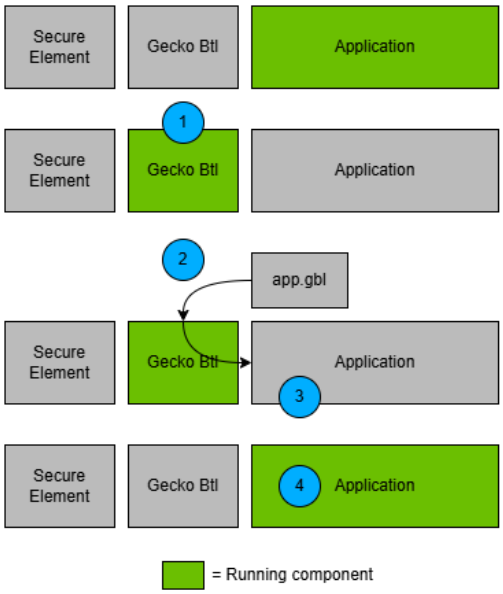
Gecko Bootloader Operation Application Upgrade

Gecko Bootloader Operation - Application Upgrade

This section summarizes Gecko Bootloader operation for updating application firmware, first if the Gecko Bootloader is configured in standalone mode and then if it is configured in application mode. The *Gecko Bootloader Operation - Bootloader Upgrade* provides the same information for updating the bootloader firmware.

Standalone Bootloader Operation

Standalone bootloader operation is illustrated in the following figure:



1. The device reboots into the bootloader.
2. A GBL file containing an application image is transmitted from the host to the device. If image encryption is enabled in the bootloader and the image is encrypted, decryption is performed during the process of receiving and parsing the GBL file.
3. The bootloader applies the application upgrade from the GBL upgrade file on-the-fly. If image authentication is enabled in the bootloader and the GBL file contains a signature, the authenticity of the image is verified before completing the process.
4. The device boots into the application. Application upgrade is complete.

Rebooting Into the Bootloader

The Gecko Bootloader supports multiple mechanisms for triggering the bootloader. If the **GPIO activation** component is installed, the host device can keep this pin low/high (depending on configuration) through reset to make the device enter the bootloader. The bootloader can also be entered through software. The `bootloader_rebootAndInstall` API first signals to the bootloader that it should enter firmware upgrade mode by writing a command to the shared memory location at the bottom of SRAM, and then performs a software reset. If the bootloader finds the correct command in shared memory upon boot, it will enter firmware upgrade mode instead of booting the existing application.

Downloading and Applying a GBL Upgrade File

When the bootloader enters firmware upgrade mode, it enters a receive loop waiting for data from the host device. The specifics of the receive loop depend on the protocol. Received packets are passed to the image parser, a state machine that parses the data and returns a callback containing any data that should be acted upon. The bootloader core implements this callback and flashes the data to external flash at the specified address. If GBL file authentication or encryption is

Copyright © 2025 Silicon Laboratories. All rights reserved.

enabled, the image parser will enforce this, and abort the image upgrade if the authentication fails The bootloader prevents a newly uploaded image from being bootable by holding back parts of the application vector table until the GBL file hash and GBL signature (if required) have been verified.

Booting Into the Application

When an application upgrade is completed, the bootloader triggers a reboot with a message in shared memory at the bottom of SRAM signaling that an application upgrade has been successfully completed. The application can use this reset information to learn that an application upgrade was just performed.

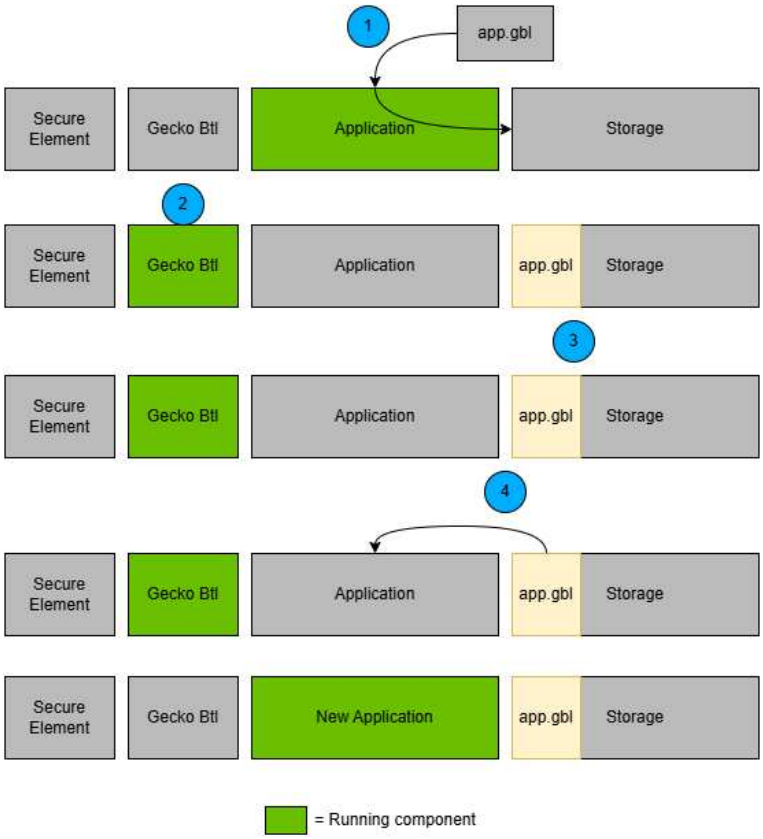
Before jumping to the main application, the bootloader verifies that the application is ready to run. This includes verifying if the Program counter and Stack Pointer are valid and also if the lock bit is set. If secure boot is enabled, the bootloader expects a signed application and attempts to validate the signature of the application. In scenarios where secure boot is not enabled, the bootloader attempts to validate if the Application properties pointer points to valid app properties structure in the flash. If valid app properties struct is found, the bootloader proceeds based on the signature type indicated by the application properties struct or else the bootloader assumes that the Application properties pointer points to the Reset Handler of the application (an app without application properties) and proceeds to boot into the application. In case the verification of the application fails at any stage, the bootloader enters the bootload mode instead of booting into the application.

Error Handling

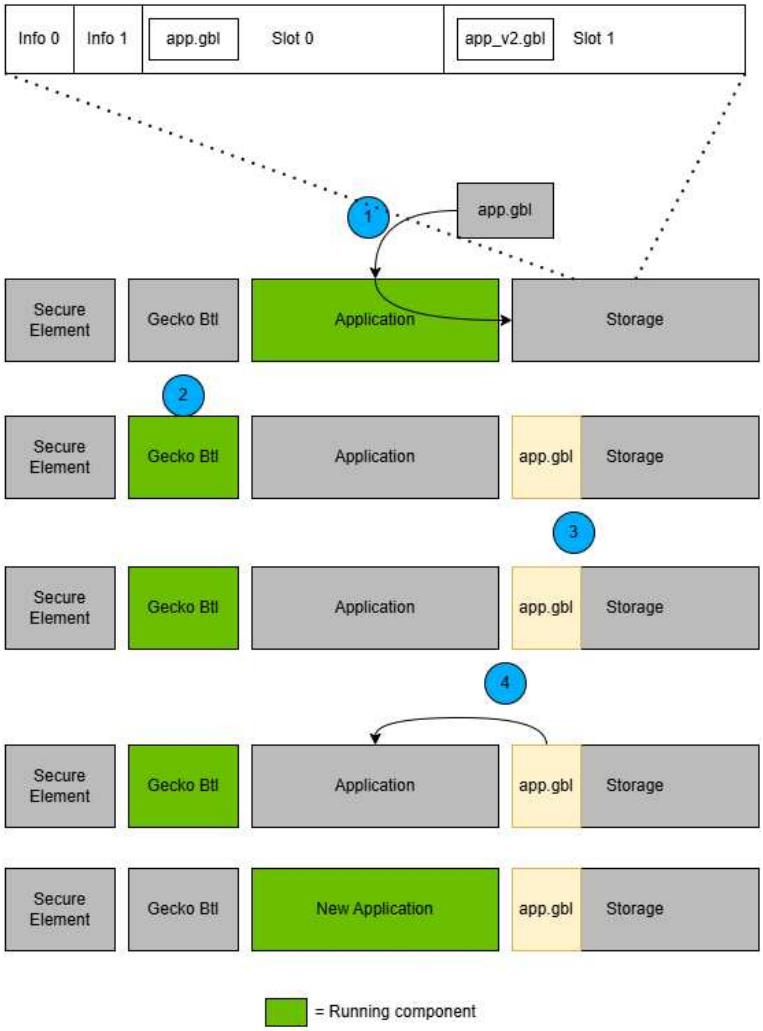
If the application upgrade is interrupted at any time, the device will be without a working application. The bootloader then resets the device, and re-enters firmware upgrade mode. The host device can easily restart the application upgrade process, to try loading the upgrade image again.

Application Bootloader Operation

The following figure illustrates the application bootloader operation both for a single image/single storage slot, and multiple images/ multiple storage slots.



Copyright © 2025 Silicon Laboratories. All rights reserved.



1. A GBL file is downloaded onto the storage medium of the device, as described below, and the presence of an upgrade image is indicated.
2. The device reboots into the bootloader, and the bootloader enters firmware upgrade mode.
3. The device applies the application upgrade.
4. The device boots into the application. Application upgrade is complete.

Downloading and Storing a GBL Upgrade Image File

To prepare for receiving an upgrade image, the application finds an available storage slot or erases an existing one using `bootloader_eraseStorageSlot`. If the bootloader only supports a single storage slot, a value of 0 should be used for the slot ID.

The application then receives a GBL file using an applicable protocol, such as Ethernet, USB, Zigbee, OpenThread or Bluetooth, and stores it in the slot by calling `bootloader_writeStorage`.

When the download is complete, the application can optionally verify the integrity of the GBL file by calling `bootloader_verifyImage`. This is also done by the bootloader before applying the image but can be done from the application to avoid rebooting into the bootloader if the received image was corrupt.

If multiple storage slots are supported, the application should write a bootload list by calling `bootloader_setImageToBootload`. The list is written to the two bootload info pages as shown in the figure above. The bootload list is a prioritized list of slots indicating the order the bootloader should use when attempting to perform a firmware upgrade. The bootloader attempts to verify the images in these storage slots in sequence and applies the first image to pass verification. If only a single storage slot is supported, the bootloader treats the entire download space as a single storage slot.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Rebooting and Applying a GBL Upgrade File

The bootloader can be entered through software. The `bootloader_rebootAndInstall` API signals to the bootloader that it should enter firmware upgrade mode by writing a command to the shared memory location at the bottom of SRAM, and then performs a software reset. If the bootloader finds the correct command in shared memory upon boot, it enters firmware upgrade mode instead of booting the existing application.

The bootloader iterates over the list of storage slots marked for bootload and attempts to verify the image stored in each. Once it finds a valid GBL upgrade file, firmware upgrade is attempted from this GBL file. If the upgrade fails, the bootloader moves to the next image in the list. If no images pass verification, the bootloader reboots back into the existing application with a message in the shared memory location in SRAM indicating that no good upgrade images were found.

Booting Into the Application

When an application upgrade is completed, the bootloader triggers a reboot with a message in shared memory at the bottom of SRAM signaling that an application upgrade has been successfully completed. The application can use this reset information to learn that an application upgrade was just performed.

Before jumping to the main application, the bootloader verifies that the application is ready to run. This includes verifying if the Program counter and Stack Pointer are valid as well as if the lock bit is set. If secure boot is enabled, the bootloader expects a signed application and attempts to validate the signature of the application. In scenarios where secure boot is not enabled, the bootloader attempts to validate if the Application properties pointer points to valid app properties structure in the flash. If valid app properties struct is found, the bootloader proceeds based on the signature type indicated by the application properties struct or else the bootloader assumes that the Application properties pointer points to the Reset Handler of the application (an app without application properties) and proceeds to boot into the application. In case the verification of the application fails at any stage, the bootloader enters the bootload mode instead of booting into the application.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Gecko Bootloader Operation Bootloader Upgrade

Gecko Bootloader Operation - Bootloader Upgrade

Bootloader upgrade functionality is provided by the Secure Engine on Series 3 devices. The Secure Engine itself is also upgradable. For more details, see the *Gecko Bootloader Operation - Secure Engine Upgrade* page.

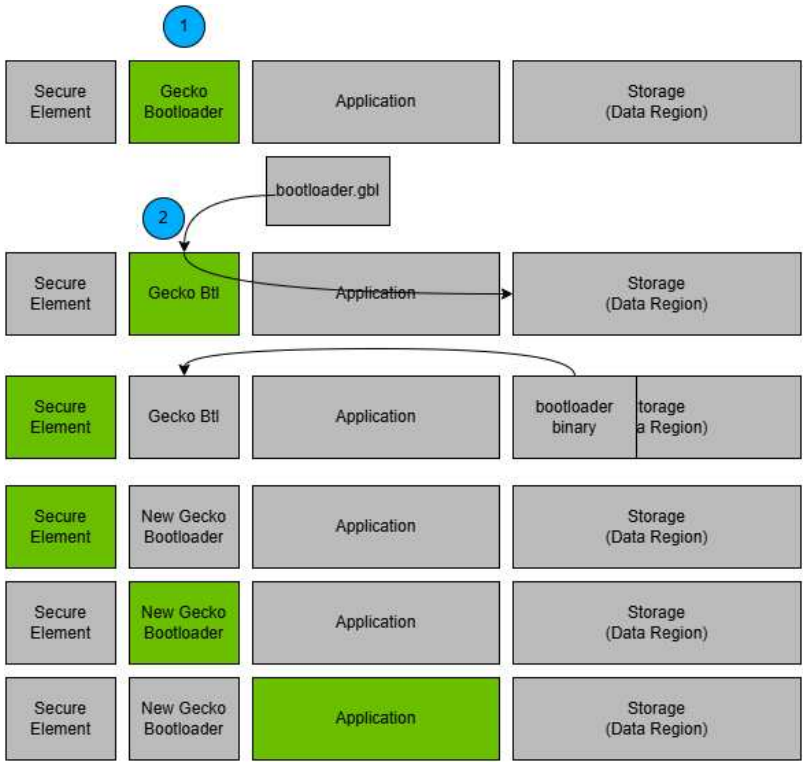
Requirements for upgrading the main bootloader vary depending on the bootloader configuration:

- Application bootloader with storage: Upgrading the main bootloader requires a single GBL file containing both bootloader and application upgrade images.
- Standalone bootloader with communication interface: Upgrading the bootloader requires a GBL file with only the bootloader upgrade image.

Security of the bootloader upgrade process is provided by signing the GBL file. See *Creating a Signed and Encrypted GBL Upgrade Image File from an Application* on the *Gecko Bootloader Security Features* page.

Bootloader Upgrade on Bootloaders with Communication Interface (Standalone Bootloaders)

The process is illustrated in the following figure:



1. The device reboots into the bootloader.
2. A GBL file containing only a bootloader upgrade image is transmitted from the host to the device.
3. The contents of the GBL Bootloader tag are written to the data region in flash
4. The device reboots into the Secure Engine.
5. The Secure Engine replaces the main bootloader with the new version found in the bootloader upgrade GBL.
6. The device boots into the new main bootloader. Bootloader upgrade is complete.

Copyright © 2025 Silicon Laboratories. All rights reserved.

A bootloader upgrade is started in the same way as an application upgrade.

Downloading and Applying a Bootloader GBL Upgrade File

When the bootloader has entered the receive loop, a GBL upgrade file containing a bootloader upgrade is transmitted to the bootloader. When a packet is received, it is passed to the image parser. The image parser parses the data and returns bootloader upgrade data in a callback. The bootloader core implements this callback and flashes the data to the flash at the bootloader upgrade location.

The bootloader prevents a newly uploaded bootloader upgrade image from being interpreted as valid by holding back parts of the bootloader upgrade vector table until the GBL file hash and GBL signature (if required) have been verified.

When a complete bootloader upgrade image is received, the bootloader signals the Secure Engine that it should enter firmware upgrade mode. Secure Engine communication is used to signal that bootloader upgrade is ready to be performed.

The authenticity of the main bootloader optionally can be verified before applying the bootloader upgrade. See *Setting a Version Number* on the *Configuring the Gecko Bootloader* page for more information about versioning bootloader images.

Upgrading Bootloaders without Secure Boot to Bootloaders with Secure Boot

A bootloader without the secure boot feature can be upgraded to a bootloader with the secure boot feature, using the following procedure:

1. Prepare a Gecko Bootloader image with secure boot enabled. The version of the bootloader needs to be higher than the bootloader on the device.
   - Turn on secure boot in Simplicity Studio by going to the **Bootloader Core** component and selecting the **Enable secure boot** option.
   - (Optional) In the **Bootloader Core** component, select the **Require signed firmware upgrade files** option. This means that the Gecko Bootloader will only accept signed GBL files.
2. Generate a public/private Signing Key pair. See the *Generating Keys* section on the *Gecko Bootloader Security Features* page for more information on creating a Signing Key pair.
3. Write the public key generated from the previous step to the device. The public key is stored as a manufacturing token in the device by default. Key locations are defined in the bootloader project file btl_security_tokens.c.
4. Create a GBL file using the Gecko Bootloader image. The GBL file needs to be signed/unsigned depending on the current configuration of the Gecko Bootloader running on the device. For more details on creating a GBL file, see the *Creating a Signed and Encrypted GBL Upgrade Image File from an Application* section on the *Gecko Bootloader Security Features* page.
5. Upload the GBL file. For more details on the upgrade process, see the *Bootloader Upgrade on Bootloaders with Communication Interface (Standalone Bootloaders)* section.

Enabling Secure Boot RTSL on Series 3 Devices

Secure Boot RTSL (Root of Trust and Secure Loader) can be enabled using the following procedure:

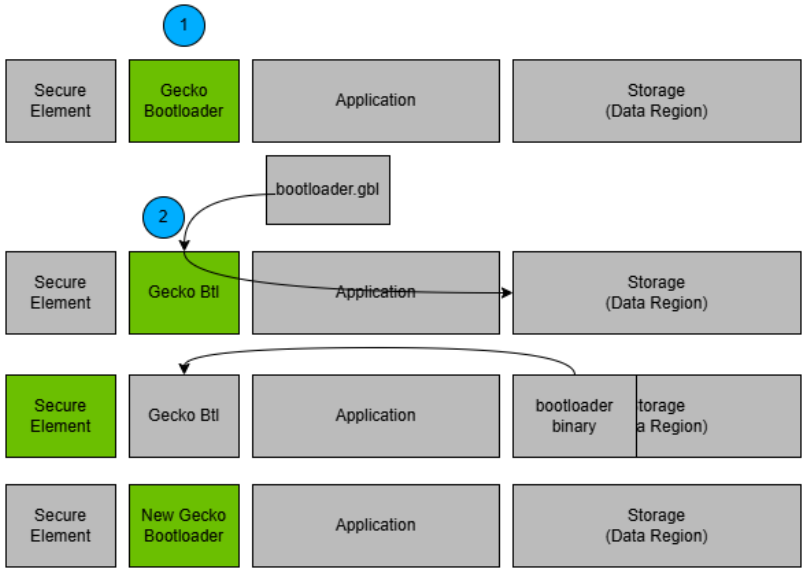
1. Prepare a Gecko Bootloader image with secure boot enabled. The version of the Gecko Bootloader needs to be higher than the Gecko Bootloader on the device.
   - Turn on secure boot in Simplicity Studio by going to the **Bootloader Core** component and selecting the **Enable secure boot** option.
   - (Optional) In the **Bootloader Core** component, select the **Require signed firmware upgrade files** option. This means that the Gecko Bootloader will only accept signed GBL files.
2. Generate a public/private Signing Key pair. See the *Generating Keys* section on the *Gecko Bootloader Security Features* page for more information on creating a Signing Key pair.
3. Prepare an application that installs the public key generated from step 2 to the Secure Engine One-time Programmable memory. Installing a key in the VSE requires a reset routine. Make sure that the application does not end up in the reset loop. Create an unsigned GBL file from this application and upload it. For more information on installing public keys, see the *Creating a Signed and Encrypted GBL Upgrade Image File from an Application* section on the *Gecko Bootloader Security Features* page.
4. Sign the Gecko Bootloader image generated from step 1 using the private key generated in step 2. See the *Signing an Application Image for Secure Boot* section on the *Gecko Bootloader Security Features* page for more information on signing binaries.

Copyright © 2025 Silicon Laboratories. All rights reserved.

5. Make a custom application that turns on secure boot on the Secure Engine and sign this application binary with the private key generated from step 2.
6. Create a GBL file using the Gecko Bootloader image from step 4.
7. Create a GBL file using the application from step 5. The GBL file need to be signed if the **Bootloader Core** component option **Require signed firmware upgrade files** was selected in step 1.
8. Upload the GBL file containing the Gecko Bootloader image.
9. Upload the GBL file containing the application.

Bootloader Upgrade on Application Bootloaders with Storage

The process is illustrated in the following figure.



1. A single GBL file containing a bootloader upgrade image is downloaded onto the storage medium of the device.
2. The device reboots into the main bootloader.
3. The main bootloader verifies the integrity of the upgrade image and then resets the device with reset reason BOOTLOADER_RESET_REASON_UPGRADE to apply the upgrade.
4. The device reboots into the Secure Engine.
5. The Secure Engine replaces the main bootloader with the new version.
6. The device boots into the new main bootloader.

A bootloader upgrade is started in the same way as an Application Upgrade. A GBL file containing a bootloader is written to storage by the application, and the bootloader is entered.

The bootloader iterates over the list of storage slots marked for bootload and attempts to verify the GBL file stored within. Verification returns information about whether the GBL file contains a bootloader, or both a bootloader and an application. The image parser parses the file. If the GBL file contains a bootloader, the bootloader upgrade data is returned in a callback. The bootloader core implements this callback and flashes the data to internal flash at the bootloader upgrade location.

The bootloader prevents a newly uploaded bootloader upgrade image from being interpreted as valid by holding back parts of the bootloader upgrade vector table until the GBL file hash and GBL signature (if required) have been verified.

Secure Engine communication interface is used to signal the Secure Engine that a bootloader upgrade is ready to be performed.

On Series 3 devices, the authenticity of the main bootloader optionally can be verified before applying the bootloader upgrade. See the *Setting a Version Number* section on the *Configuring the Gecko Bootloader* page for more information about versioning bootloader images.

The new main bootloader is entered, and the images in the list of storage slots marked for bootload are verified. When the image parser parses the slot containing the GBL file with the bootloader + application upgrade, the version number of the bootloader upgrade is equal to the running main bootloader version, so another bootloader upgrade will not be performed.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Instead, the application upgrade data are returned in a callback. Bootloading of the new application proceeds as described in the *Application Bootloader Operation* section on the *Gecko Bootloader Operation Application Upgrade* page.

Storage Space Size Configuration

The storage space size must be configured to have enough space to store the upgrade images.

Upgrading Bootloaders without Secure Boot to Bootloaders with Secure Boot

A bootloader without the secure boot feature can be upgraded to a bootloader with the secure boot feature. The procedure is as follows:

1. Prepare a Gecko Bootloader image with secure boot enabled. The version of the bootloader needs to be higher than the bootloader on the device.
   - Turn on secure boot from the **Bootloader Core** component in Simplicity Studio by selecting the **Enable secure boot** option.
2. Generate a public/private Signing Key pair. See the *Generating Keys* section on the *Gecko Bootloader Security Features* page for more information on creating a Signing Key pair.
3. Write the public key generated from the previous step to the device. The public key is stored as a manufacturing token in the device by default. This key can be written by application code running on the device. The Gecko Bootloader prepared from step 1 can be modified to look for the decryption and signature keys in a different location. Key locations are defined in the bootloader project file `btl_security_tokens.c`.
4. Prepare a signed application image using the private key generated in step 2. See the *Signing an Application Image for Secure Boot* section on the *Gecko Bootloader Security Features* page for more information on signing an application.
5. Create a GBL file using the Gecko Bootloader image and the signed application image. The GBL file needs to be signed/unsigned depending on the configuration of the Gecko Bootloader running on the device. For more details on creating a GBL file, see the *Creating a Signed and Encrypted GBL Upgrade Image File from an Application* section on the *Gecko Bootloader Security Features* page.
6. Upload the GBL file. For more details on the upgrade process, see the *Bootloader Upgrade on Application Bootloaders with Storage* section.

Enabling Secure Boot RTSL on Series 3 Devices

Secure Boot RTSL can be enabled by using the following procedure:

1. Prepare a Gecko Bootloader image with secure boot enabled. The version of the Gecko Bootloader needs to be higher than the Gecko Bootloader on the device.
   - Turn on secure boot from the **Bootloader Core** component in Simplicity Studio by selecting the **Enable secure boot** option.
   - (Optional) In the **Bootloader Core** component, select the **Require signed firmware upgrade files** option. This means that the Gecko Bootloader will only accept signed GBL files.
2. Generate a public/private Signing Key pair. See the *Generating Keys* section on the *Gecko Bootloader Security Features* page for more information on creating a Signing Key pair.
3. Prepare an application that installs the public key generated from step 2 to the Secure Engine One-time Programmable memory. Installing a key in VSE requires a reset routine. Make sure that the application does not end up in the reset loop. Create an un- signed GBL file from this application and upload it. For more details on creating a GBL file, see the *Creating a Signed and Encrypted GBL Upgrade Image File from an Application* section on the *Gecko Bootloader Security Features* page.
4. Sign the Gecko Bootloader image generated from step 1 using the private key generated in step 2. See the *Signing an Application Image for Secure Boot* section on the *Gecko Bootloader Security Features* page for more information on signing binaries.
5. Make a custom application that turns on secure boot on the Secure Engine and sign this application binary with the private key generated from step 2. For more details on how to turn on secure boot on the Secure Engine.
6. Create a GBL file using the Gecko Bootloader image from step 4 and the application from step 5. The GBL file must be signed if the **Bootloader Core** component option **Require signed firmware upgrade files** was selected in step 1. For more details on creating a GBL file, see the *Creating a Signed and Encrypted GBL Upgrade Image File from an Application* section on the *Gecko Bootloader Security Features* page.
7. Upload the GBL file containing the Gecko Bootloader image and the application.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Gecko Bootloader Operation Secure Engine Upgrade

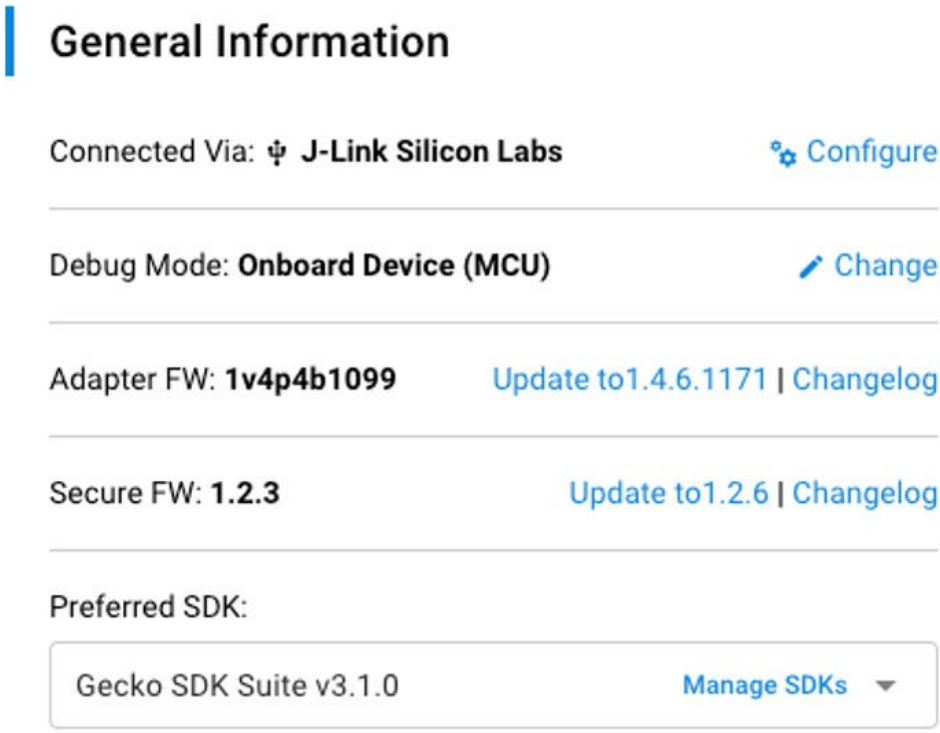
Gecko Bootloader Operation - Secure Engine Upgrade

The Secure Engine is upgradable and for both application and standalone type of bootloader a GBL file containing the Secure Engine Upgrade image has to be flashed or sent to the bootloader.

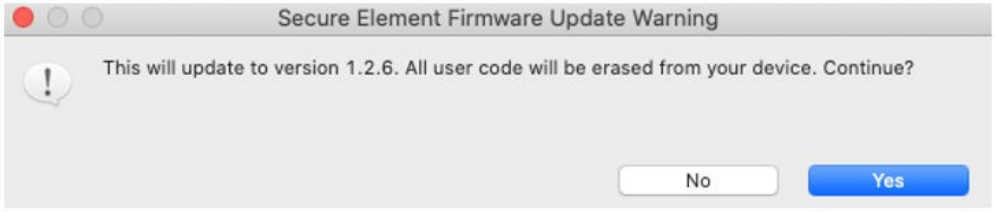
A bootloader upgrade can also be included in the same GBL file in application mode, or as a second GBL file in standalone mode. The figures that illustrate Gecko Bootloader operation in this section do not provide information about the bootloader memory layouts for different devices. For more details refer to the *Memory Space for Bootloading* section in Bootloader Fundamentals.

Signed and encrypted Secure Engine upgrade images are provided by Silicon Labs through Simplicity Studio. Upgrade images with the same or lower version number than the running Secure Engine will be ignored.

To download Secure Engine firmware images, connect a Series 3 device and select a preferred SDK. The Secure Firmware **Update to x.x.x** link appears in the Launcher Perspective, as shown in the following figure.



Click **Update to x.x.x** next to Secure FW: x.x.x. A warning dialog box appears. Click **Yes** to continue.



Copyright © 2025 Silicon Laboratories. All rights reserved.

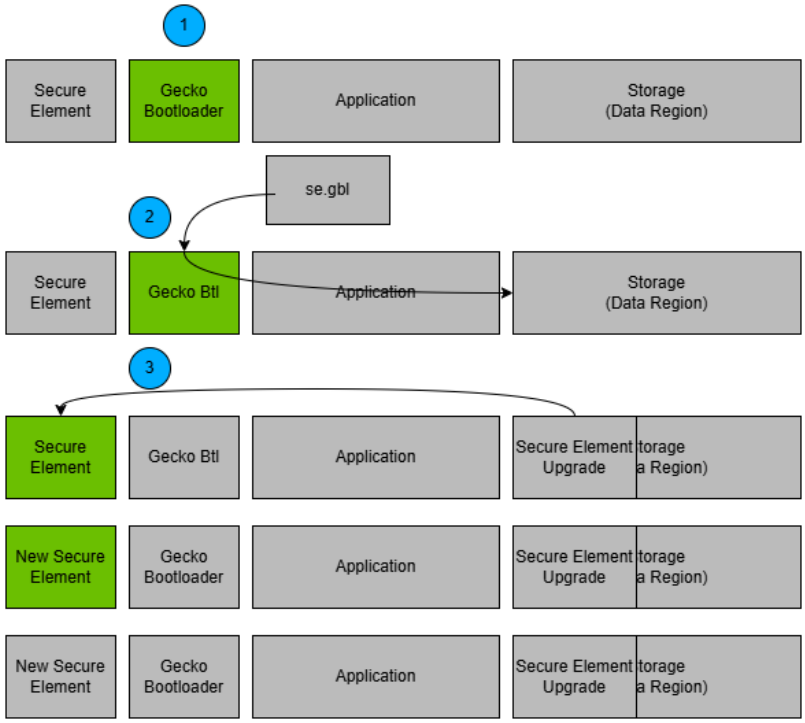
The Launcher Perspective is then updated so that the current Secure Firmware version and link version are the same.



The Secure Engine firmware images can be found in the *util/se_release/public* directory under the Gecko SDK. Simplicity Studio displays the SE firmware version available in the Gecko SDK selected.

Secure Engine Upgrade on Bootloaders with Communication Interface (Standalone Bootloaders)

The process is illustrated in the following figure.



1. The device reboots into the bootloader.
2. A GBL file containing only a Secure Engine upgrade image is transmitted from the host to the device.
3. The device reboots into the Secure Engine.
4. The Secure Engine is replaced by the new version found in the pre-configured upgrade location.
5. The device boots into the bootloader.

Downloading and Applying a Secure Engine GBL Upgrade File

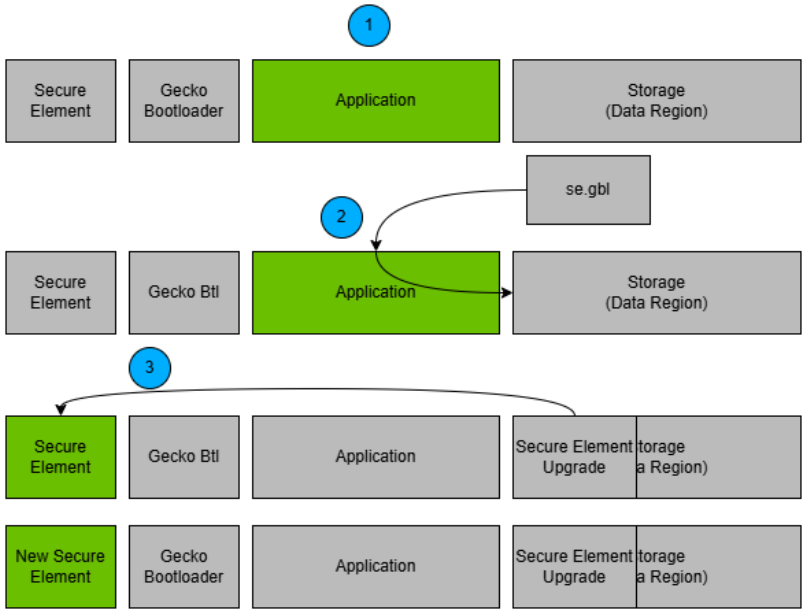
When the bootloader has entered the receive loop, a GBL upgrade file containing a Secure Engine upgrade is transmitted to the bootloader. When a packet is received, it is passed to the image parser. The image parser parses the data and returns Secure Engine upgrade data in a callback. The bootloader core implements this callback and flashes the data to flash at the pre-configured storage data region.

Copyright © 2025 Silicon Laboratories. All rights reserved.

When a complete Secure Engine upgrade image is received, the bootloader signals the Secure Engine that it should enter firmware upgrade mode. This is done by the Secure Engine communication interface that is used to signal that bootloader upgrade is ready to be performed.

Secure Engine Upgrade on Application Bootloaders with Storage

The process is illustrated in the following figure.



1. A single GBL file containing a Secure Engine upgrade image is downloaded onto the storage medium of the device.
2. The device reboots into the bootloader.
3. The upgrade image will be fetched directly from the GBL file in storage instead of first copying the image to the pre-configured upgrade location.
4. The device reboots into the Secure Engine.
5. The Secure Engine is replaced by the new version found in the pre-configured upgrade location (or directly from storage, ref. 3b).
6. The device boots into the main bootloader.
7. The bootloader applies the application image from the GBL upgrade file.
8. The device boots into the application. Secure Engine upgrade is complete.

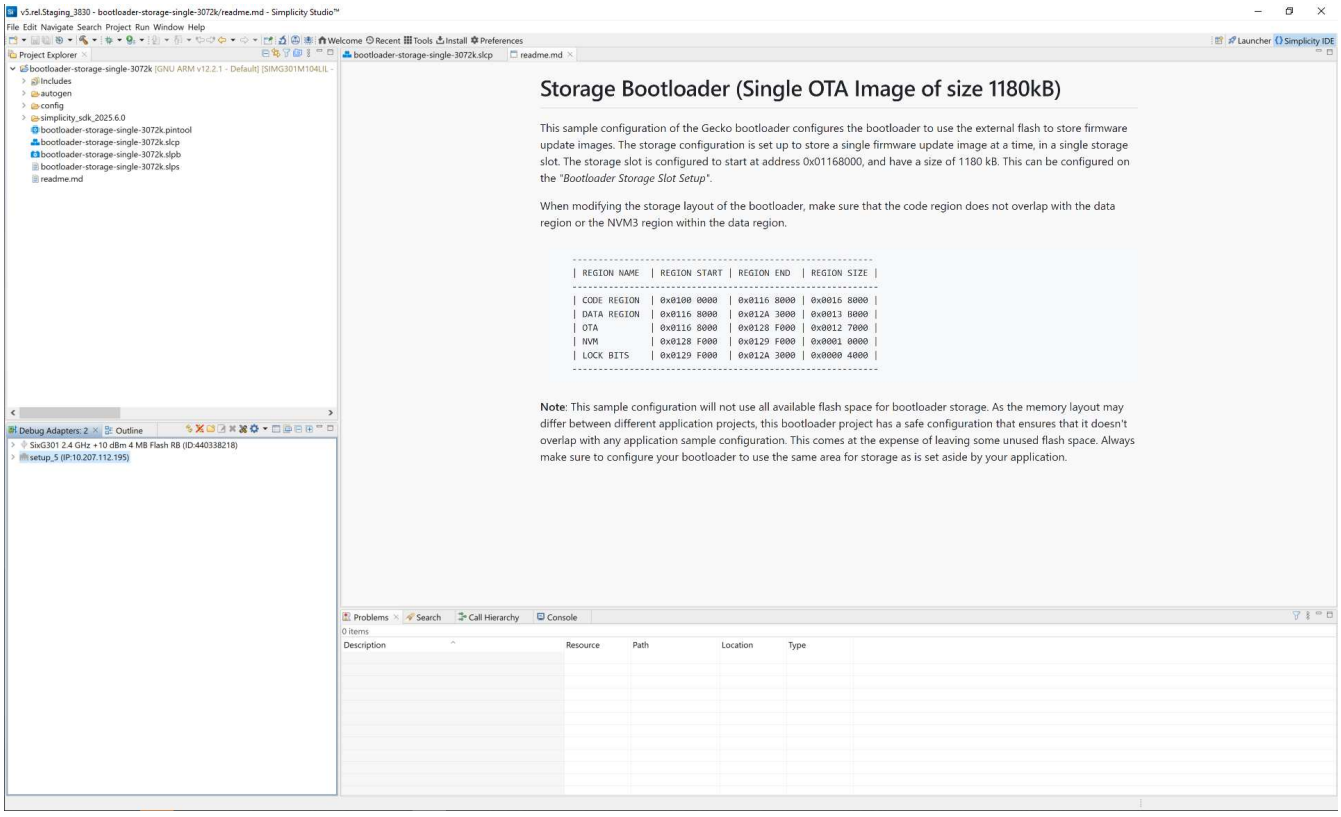
Copyright © 2025 Silicon Laboratories. All rights reserved.

Getting Started with the Gecko Bootloader

Getting Started with the Gecko Bootloader

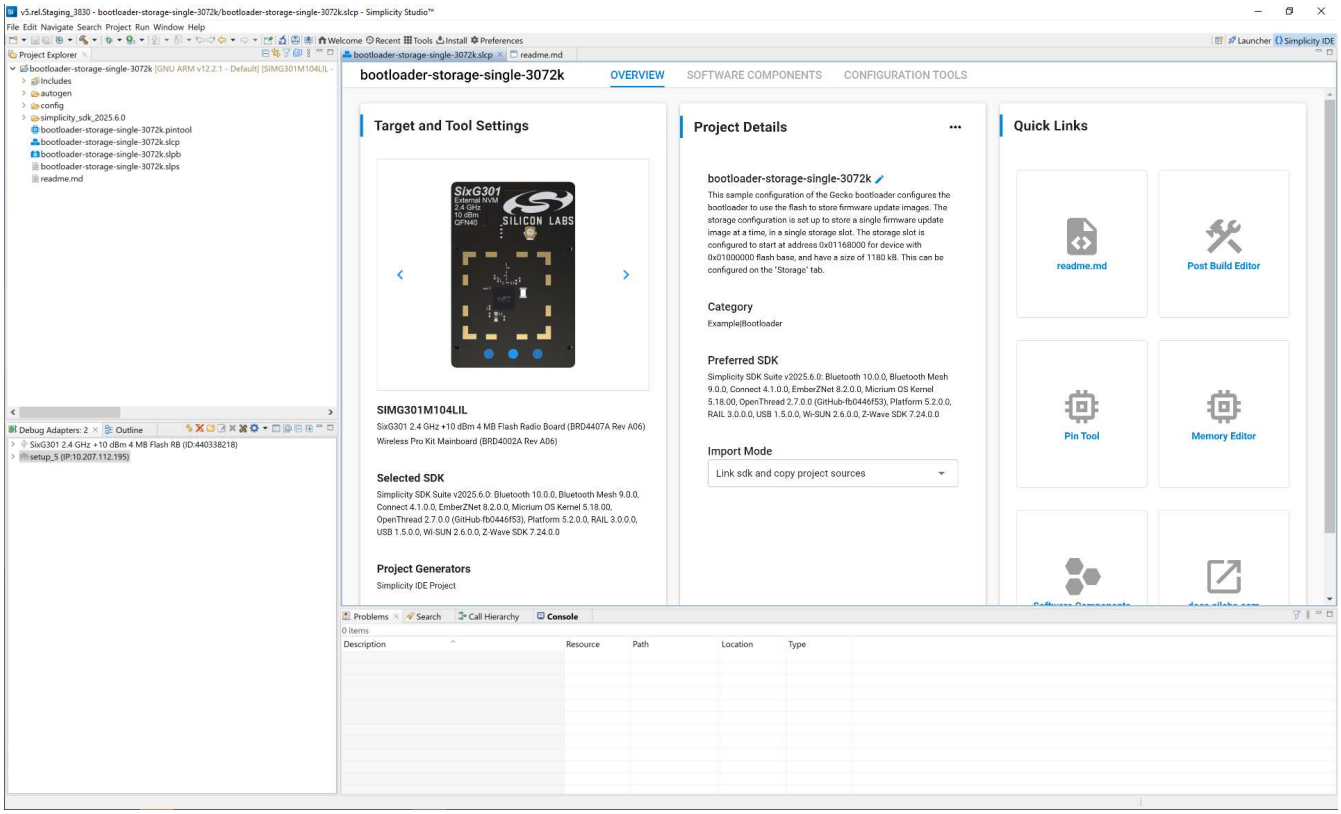
This section describes how to build a Gecko Bootloader from one of the provided examples. These instructions assume that you have installed Simplicity Studio 5, the SiSDK and associated utilities as described in the SDK's quick start guide, and that you are familiar with generating, compiling, and flashing an example application in the relevant version.

1. Create a project based on the Gecko Bootloader example of your choice. The project opens with a tab describing the example.

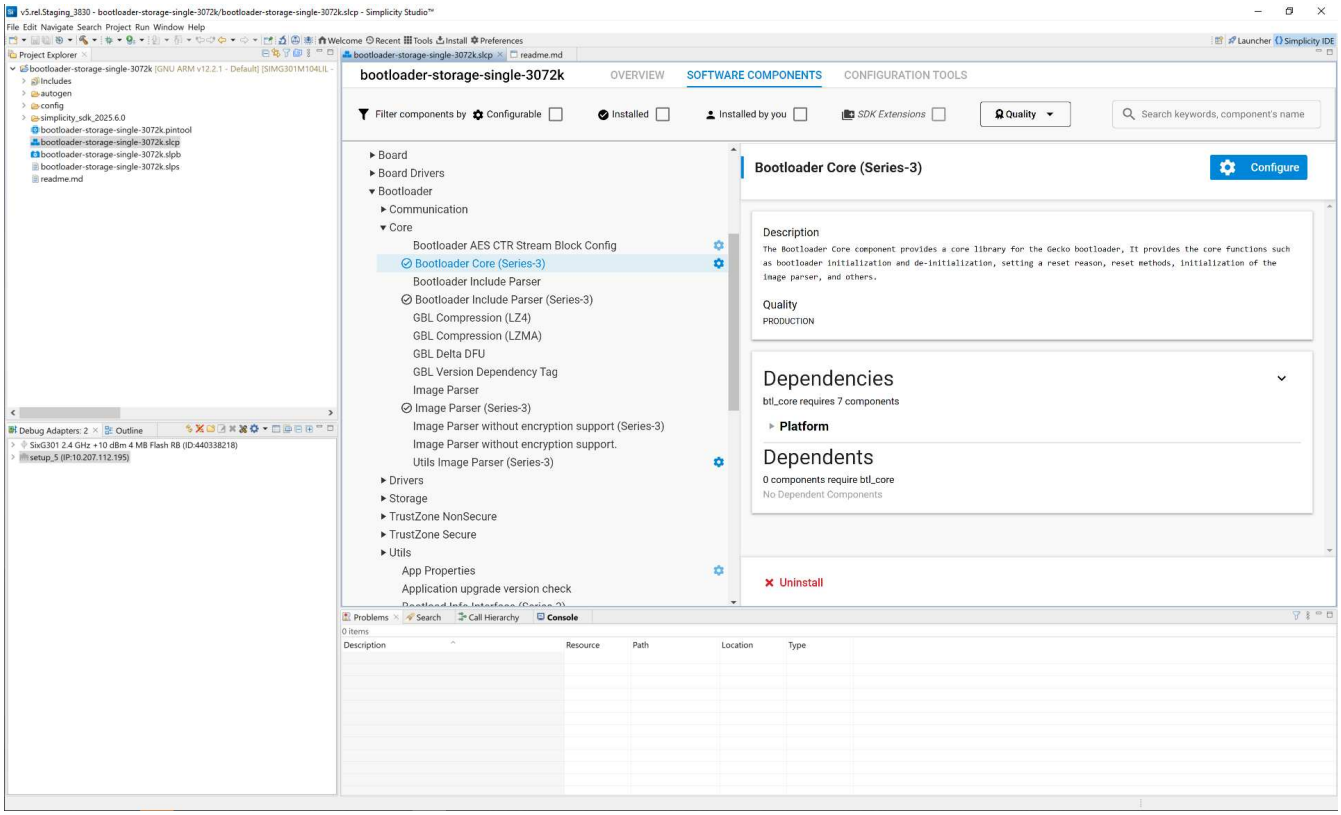


2. Click the project (*.slcp) tab to move to the Project Configurator interface.

Copyright © 2025 Silicon Laboratories. All rights reserved.

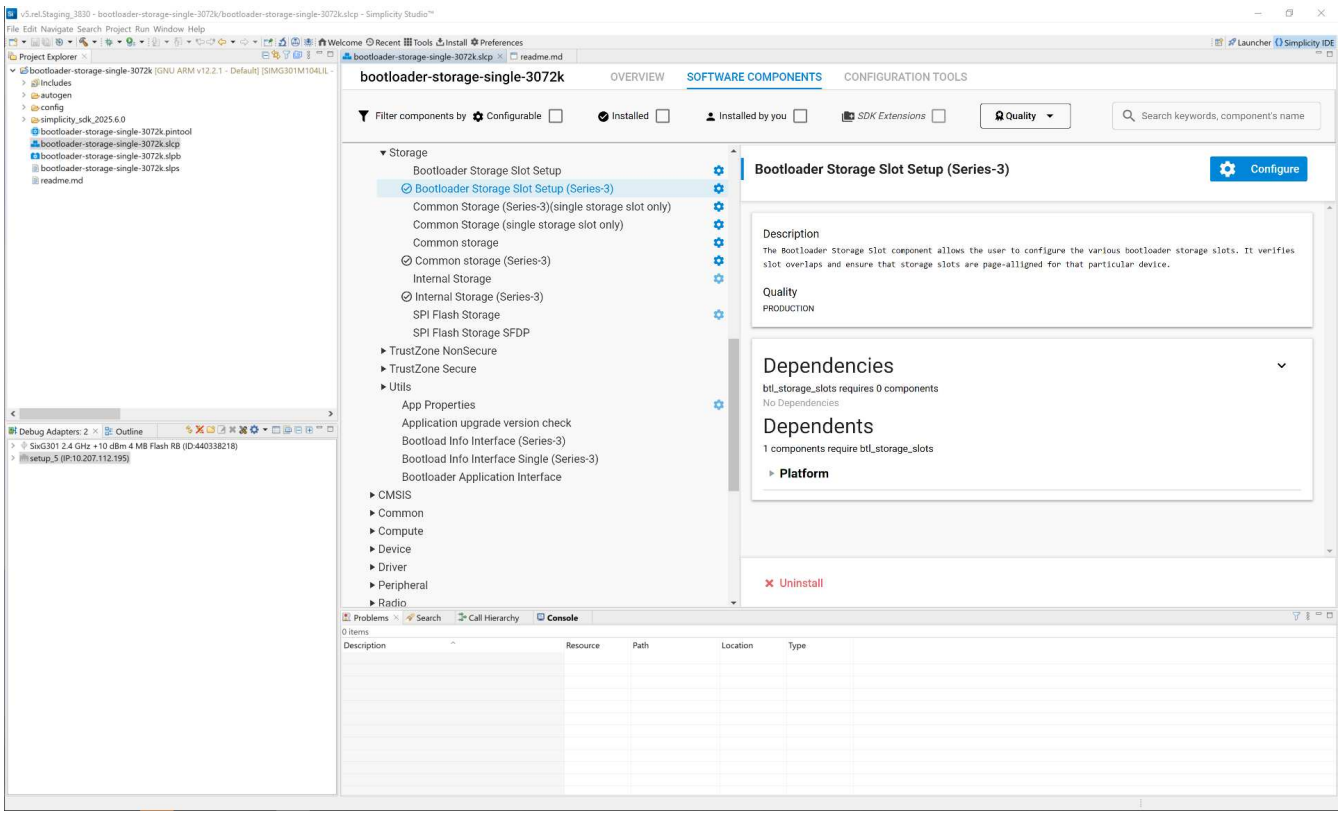


3. The Software Components tab shows the list of available components that can be installed in the project.

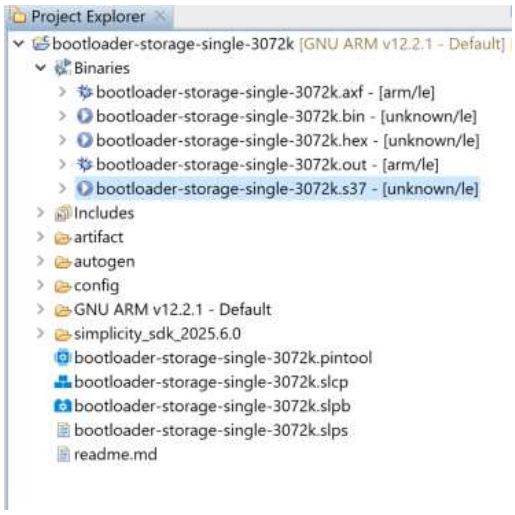


Copyright © 2025 Silicon Laboratories. All rights reserved.

4. The **Storage Slot Setup** component allows you to configure storage slots to be used if a storage component is also installed. The default configuration matches the target part and bootloader type. This component supports a maximum of three storage slots.



5. Click the **Build** (hammer) icon.
6. After the build is complete, the bootloader binaries are available in the **artifact** folder as depicted in the image below.



The image containing only a bootloader must be used to create a GBL file for bootloader upgrade.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Configuring the Gecko Bootloader

Configuring the Gecko Bootloader

Configuring Storage

Gecko Bootloaders configured as application bootloaders must include an API to store and access image files. This API is based on the concept of storage slots, where each slot has a predefined size and location in memory and can be used to store a single upgrade image. Slots are configured in the **Bootloader Storage Slot Setup** component.

When multiple storage slots are configured, a bootload list is used to indicate the order in which the bootloader should access slots to find upgrade images. If multiple storage slots are supported, the application should write the bootload list by calling bootloader_setImageToBootload before rebooting into the bootloader to initiate a firmware upgrade process. The bootloader attempts to verify the images in these storage slots in sequence and applies the first image to pass verification. If only a single storage slot is supported, the bootloader uses this slot implicitly. A maximum of three slots may be configured in the **Bootloader Storage Slot Setup** component.

Storage Configuration

When configuring a Gecko Bootloader to obtain images fromflash, modify the following.

The **base address of the storage area** should be configured in the **Common Storage** component. This is the address at which the bootloader will place the prioritized list of storage slots to attempt to bootload from, if more than one storage slot is configured. In the default configuration, only a single storage slot is configured, so this value is set to 0, and isn't used. If more than one storage slot is configured, this value needs to be configured too.

The **location and size of the storage slots** can be configured using the **Bootloader Storage Slot Setup** Component (supports a maximum of three configurable storage slots). The addresses input here are absolute addresses (they are not offsets from the base address). If more than a single slot is configured, enough space must be reserved between the base address as configured in the **Common Storage** component and the first storage slot configured in the **Bootloader Storage Slot Setup** component. Enough space to fit two copies of the bootload list must be reserved. These two copies need to reside on different flash pages, to provide redundancy in case of power loss during writing. Two full flash pages therefore need to be reserved. The following figure illustrates how the storage area can be partitioned.

Storage with one storage slot

0	512kB
Application	Storage Slot 0 Size = 512kB

Storage with two storage slot

0	340kB	344kB	348kB	652kB
Application	Bootload list	Bootload list	Storage slot 0 Size = 340kB	Storage slot 1 Size = 340kB

> **Note**: The storage area partitioning in the example for two storage slots above does not take any NVM system into account. If using an NVM3 , take care to place and size the storage area in such a way that bootloader storage does not overlap with NVM3.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Compressed Upgrade Images

The Gecko Bootloader optionally supports compressed GBL files. In a compressed GBL file, only the application upgrade data and bootloader upgrade data is compressed, any metadata (if present) stays uncompressed. Signature and encryption operations on a compressed GBL work identically to on an uncompressed GBL.

To be able to use compressed upgrade images, a decompressor for the relevant compression algorithm must be added to the Gecko bootloader. The following table shows which compression algorithms are supported by the Gecko Bootloader, and which Bootloader component should be added to enable the feature. The table also shows how much space the decompressor takes up in the bootloader, and how big of a size reduction to expect for the compressed application upgrade image. Be aware of the bootloader size requirement. The bootloader space might be too small to fit the decompressors, depending on the device and enabled components.

Compression Algorithm	Component	Bootloader Size Requirement	Application Upgrade Size Reduction (typical)
LZ4	GBL Compression (LZ4)	< 1 kB	~ 10%
LZMA	Bootloader Compression (LZMA)	~5 kB flash, 18 kB RAM	~ 30%

It is important to note that the compressed GBL file stays compressed while being transferred to the device, and while it is stored in the upgrade area. It is decompressed by the bootloader when the upgrade is applied. This means that the running application in main flash will be identical to one that was installed using an uncompressed (normal) GBL file.

Compressed GBL files can only be decompressed by the bootloader when running standalone, not through the Application Interface. This means that upgrade image verification performed by the application prior to reboot will not attempt to decompress the application upgrade, it will only verify the signature of the compressed payload. After rebooting into the bootloader, it will decompress the image as part of the upgrade process.

LZMA Compression Settings

LZMA decompression is only supported for images compressed with certain compression settings. Simplicity Commander automatically uses these settings during gbl4 creation.

- Probability model counters: lp + lc <= 2. Simplicity Commander uses lp=1, lc=1.
- Dictionary size no greater than 8 kB. Simplicity Commander uses 8 kB.

Together, these settings cause the decompressor to require 18 kB of RAM for decompression – 10 kB for the counters and 8 kB for the dictionary.

The Gecko bootloader only supports decompressing payloads that contain the end mark as the last 8 bytes of the compressed stream.

Bootloader Example Configurations

The following sections describe the key configuration options for the example bootloader applications.

> **Note**: Security features are disabled for all example configurations. In development, Silicon Labs strongly recommends enabling security features to prevent unauthorized parties from uploading untrusted program code. See the *Using Application Image Security Features* section on the *Gecko Bootloader Security Features* page to learn how to configure the security features of the Gecko Bootloader.

UART XMODEM Bootloader

Standalone bootloader for Series 3 devices running the EmberZNet PRO and Silicon Labs Connect protocol stacks, using XMODEM-CRC over UART.

Copyright © 2025 Silicon Laboratories. All rights reserved.

In this configuration, the **UART XMODEM** communication component, **XMODEM Parser** component, and **Bootloader UART Driver** component are installed. For the example application to run on a custom board, the GPIO ports and pins used for UART need to be configured in the **Bootloader UART Driver** component. Here, Hardware Flow Control can be enabled or disabled, and the baud rate and pinout can be configured.

The **GPIO activation** component is also installed by default, allowing bootloader entry into firmware upgrade mode by activating a GPIO through reset. The GPIO pin used can be configured here. This component can be uninstalled if this functionality is not desired.

BGAPI UART DFU Bootloader

Standalone bootloader for the Bluetooth protocol stack, using the BGAPI protocol for UART DFU. This bootloader should be used for all NCP-mode Bluetooth applications.

In this configuration, the **BGAPI UART DFU** communication component and **Bootloader UART Driver** component are installed. For the example application to run on a custom board, the GPIO ports and pins used for UART need to be configured in the **Bootloader UART Driver** component. Here, Hardware Flow Control can be enabled or disabled, and the baud rate and pinout can be configured.

The GPIO activation component is also installed by default, allowing bootloader entry by activating a GPIO through reset. The GPIO pin used can be configured here. This component can be uninstalled if this functionality is not desired.

Storage Bootloader

Application bootloader for all wireless protocol stacks, using flash to store upgrade images received over the air by the application.

Multiple examples are provided, including configurations for 2 MB, 3 MB, 4 MB & 8 MB flash memory devices . **The storage layout should be modified before running the bootloader on any other devices**. In this configuration, the flash and common storage components are installed. The base address of the storage area is configured in the **Common Storage** component. The location and size of the storage slots can be configured using the **Bootloader Storage Slot Setup** component (provides up to three configurable storage slots). Default example applications are provided with configurations for both single storage slot and multiple storage slots.

The default storage slot configurations provided by the Gecko Bootloader **must** be configured to match the use-case-specific application configurations, as shown below.

Sample Applications	Storage Offset	Storage Size
Storage Bootloader (Single OTA Image of size 792kB)	0x010E0000	0x000C6000 (811,008 bytes)
Storage Bootloader (Single OTA Image of size 1180kB)	0x0128F000	0x00127000 (1,210,368 bytes)
Storage Bootloader (Single OTA Image of size 1540kB)	0x01379000	0x00181000 (1,577,984 bytes)

Image Acquisition Application Example Configuration

These examples illustrate applications that acquire and store a GBL upload image for an application bootloader. For the running bootloader to accept an application upgrade, the new application version must be higher than the existing version.

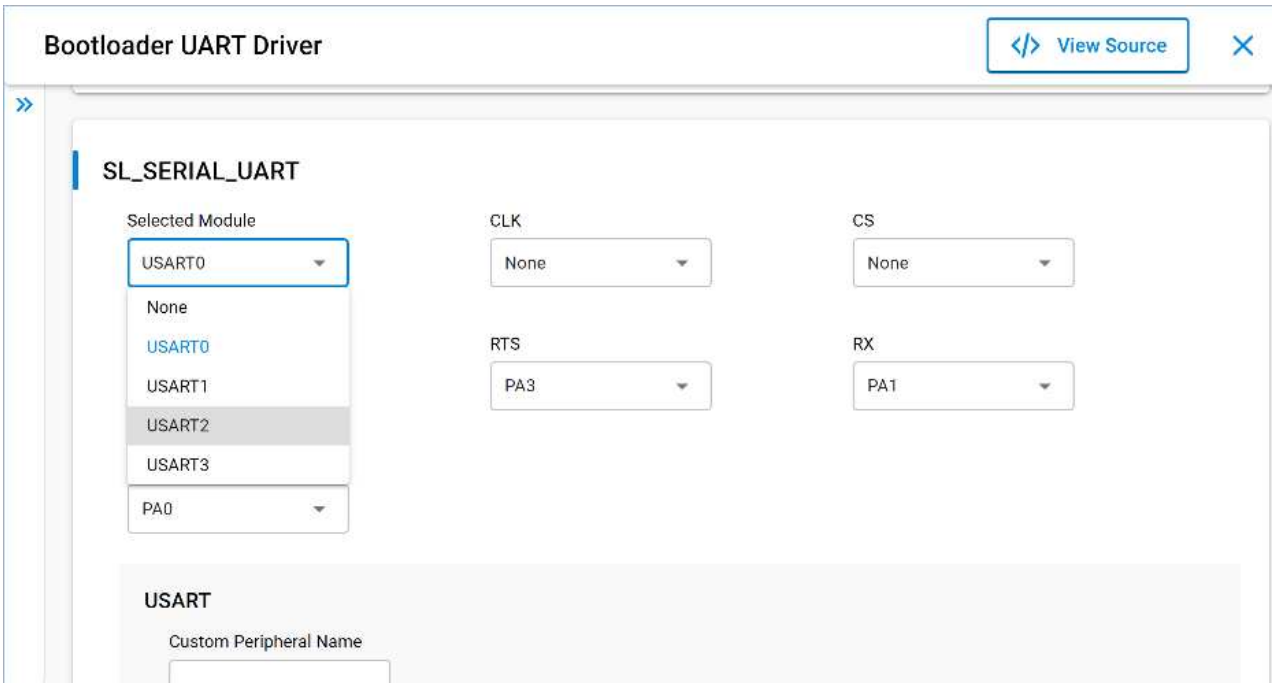
Setting a Version Number

To distinguish between different builds of the Gecko Bootloader, it is useful to set a version number. To perform a bootloader upgrade, not only must the running bootloader pass its integrity checks (see the *Downloading and Applying a Bootloader GBL Upgrade File* section on the *Gecko Bootloader Operation Bootloader Upgrade* page), but the bootloader upgrade image must also have a higher version number than the running bootloader image. A version number can be set using Simplicity Studio by configuring the **Bootloader Version Main Customer** option of the **Bootloader Core** component. This macro will be picked up by the config file **btl_config.h**, where it is combined with the version number of the Gecko Bootloader files provided by Silicon Labs.

Hardware Configuration

Copyright © 2025 Silicon Laboratories. All rights reserved.

The Gecko Bootloader uses the Pin Tool for configuration of pinout and other hardware-related settings. When Pin Tool configuration is available for a bootloader component, the relevant settings are shown in the Component Editor for that component.



The standalone Pin Tool User Interface can also be used to configure settings for Gecko Bootloader if desired.

While the Pin Tool provides configuration for many different peripherals, the Gecko Bootloader uses only the following Pin Tool modules:

- SERIAL is used by the UART Driver component to configure baud rate, flow control mode and pinout.
- VCOM is used by the UART Driver component to enable the serial interface if necessary (only on Silicon Labs Wireless Starter Kits).
- BTL_BUTTON is used by the GPIO Activation component.
- CMU HFXO frequency setting is used by the delay driver to calibrate timing if the core is running from the HFXO.

Other settings, like CMU oscillator configuration or DCDC configuration, are not taken into consideration by the default bootloader code. If using these configuration settings is desired, the required code must be added in btl_main.c.

> **Note**: While the delay driver uses the HFXO frequency setting from Pin Tool, the HFXO enable setting is not used to initialize the HFXO on startup. This setting is only used when calling the bootloader through the Application Interface, and the application has switched to the HFXO prior to calling the Bootloader Application Interface API.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Simplicity Commander and the Gecko Bootloader

Simplicity Commander and the Gecko Bootloader

Simplicity Commander is a single, all-purpose tool to be used in a production environment. It is invoked using a simple CLI (Command Line Interface) that is also scriptable. You can use Simplicity Commander to perform these essential tasks:

- Generating key files for signing and encryption
- Signing application images for Secure Boot
- Creating GBL images (encrypted or unencrypted, signed, or unsigned)
- Parsing GBL images

Simplicity Commander is used throughout the examples in the following sections. For more information on executing commands to complete these tasks, see the Simplicity Commander User Guide.

> **Note**: Simplicity Commander also offers a GUI (Graphical User Interface) that can be used in the lab for typical tasks such as flashing device images. The functions described in this User Guide are performed from the CLI.

Creating GBL V4 Files Using Simplicity Commander

To create an unsigned GBL V4 file from an application **myapp.s37**, execute `commander gbl4 create myapp.gbl --config configfile.yaml`.

Example of a config file is as follows:

```
manifest:
 product_id:"00000000000000000000000000000000"
updates:
  -data:myapp.s37
   block_size: 0
```

To create an unsigned GBL V4 file from a bootloader upgrade **mybootloader.s37**, execute `commander gbl4 create mybootloader.gbl --config confligfile.yaml`.

Example of a config file is as follows:

```
manifest:
 product_id:"00000000000000000000000000000000"
updates:
  -data:mybootloader.s37
   block_size: 0
```

This file can be used with the standalone bootloader configurations of the Gecko Bootloader.

To create an unsigned GBL file from a Secure Engine, upgrade **mySecureElement.seuv2**, and execute `commander gbl4 create mySecureElement.gbl –config configfile.yaml`.

Example of a config file is as follows:

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
manifest:
 product_id:“00000000000000000000000000000000”
se_upgrade:
 se_file:mysecureElement.seuv2
```

The Secure Engine images, .seuv2, are provided by Silicon Labs and can be found through Simplicity Studio. See the *Gecko Bootloader Operation - Secure Engine Upgrade* section on the *Gecko Bootloader Operation Secure Engine Upgrade* page.

The command can also be used to create a single upgrade image, suitable for use with application bootloader configurations of the Gecko Bootloader: `commander gbl4 create myupgrade.gbl –config configfile.yaml`.

Example of a config file is as follows:

```
manifest:
 product_id:“00000000000000000000000000000000”
updates:
 -data:myapp.s37
 -data:mybootloader.s37

se_upgrade:
 se_file: mysecureElement.seuv2
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Gecko Bootloader Security Features

Gecko Bootloader Security Features

About Bootloader Image Security

Secure Boot and Secure Firmware Upgrade, discussed in the following sections, enables Gecko Bootloader to provide authenticity and integrity checks on the Application image, which provides a sufficient level of security for many applications. However, in systems without a hardware root of trust, no process checks the authenticity or integrity of the Gecko Bootloader itself. Its security is provided solely by the device hardware and the robustness of the software running on the device.

The native behavior of Firmware Upgrade will prevent accidental version rollback of Gecko Bootloader under normal usage conditions.

About Application Image Security

The Gecko Bootloader can enforce security on two levels:

- Secure Boot refers to the verification of the authenticity of the application image in main flash on every boot of the device.
- Secure Firmware Upgrade refers to the verification of the authenticity of an upgrade image before performing a bootload, and optionally enforcing that upgrade images are encrypted.

Secure Boot Procedure

When Secure Boot is enabled, the cryptographic signature of the application image in flash is verified on every boot before the application is allowed to run. Secure Boot is not enabled by default in the example configurations provided by Silicon Labs, but enabling it is highly recommended to ensure the validity and integrity of firmware images.

Signature Algorithms

The Gecko Bootloader supports the ECDSA-P256-SHA256 cryptographic signature algorithm. This is the ECDSA (elliptical curve digital signature algorithm) of the SHA-256 digest of the application firmware image, using the NIST P-256 (secp256r1) curve.

Summary of Operation

1. On boot, the bootloader checks the application image for information about whether it is signed.
2. The type of signature and signature location is determined.
3. If the type of signature does not match the requirements of the bootloader, the bootloader enters device firmware upgrade mode and prevents the application from running.
4. According to the chosen signature algorithm, the signature of the contents of flash from the beginning of the application to the location of the signature is compared to the signature at the signature location.
5. If the signatures do not match, the bootloader enters device firmware upgrade mode and prevents the application from running.

Secure Boot using ECDSA-P256-SHA256

For an image to be signed for Secure Boot, the application needs to contain a copy of the **ApplicationProperties_t** struct. This struct contains information about which signature algorithm is used, and where to find the signature.

On every boot, the bootloader calculates the SHA-256 digest of the application image, from the beginning of the application to the start of the signature. The signature of the SHA-256 digest is then verified using ECDSA-P256.

Copyright © 2025 Silicon Laboratories. All rights reserved.

If the signature is valid, the application is allowed to boot. Else, the bootloader is entered, and an application upgrade is attempted if one is available.

Simplicity Commander can be used to generate a key pair and write the public key to the device. See the Simplicity Commander User Guide for more information.

Secure Boot with Application Rollback Protection

On every boot, the application version in the **ApplicationData_t** struct is stored at the end of the bootloader area in flash, which is used to prevent applications from being downgraded. The application version can remain the same for upgrades.

If application rollback prevention component is installed in the bootloader project, then before applying the upgrade image, the bootloader will check the application version in the application properties structure that resides inside the signed/encrypted GBL file and will only apply the OTA image to application area if the application version in ApplicationData structure is equal or higher than the highest application version last seen.

The application rollback prevention feature can be enabled in the **Bootloader Core** component by selecting the **Enable application rollback protection** option. The **Minimum application version allowed** option can be used to configure the minimum application version that should be allowed to boot.

The application versions are stored in the SE OTP area.

Secure Boot Using a Certificate

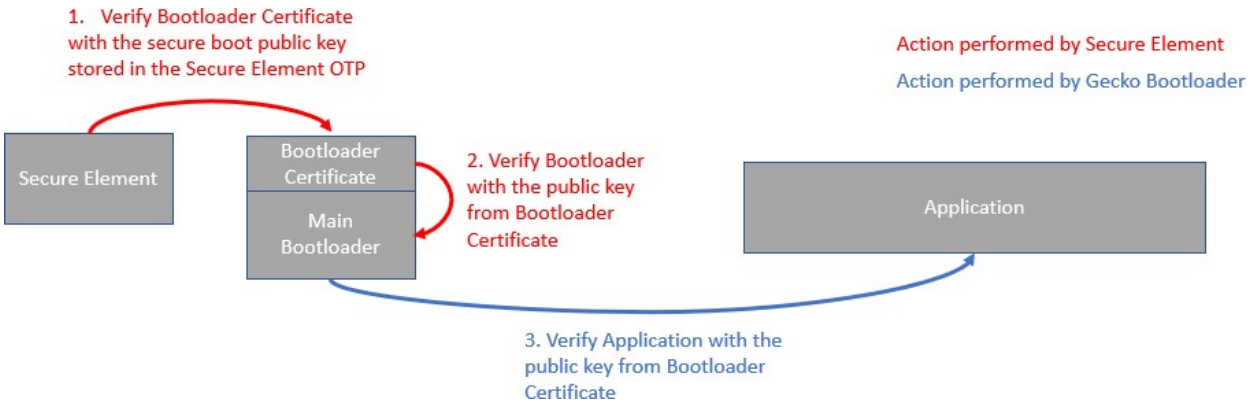
On Series 3 devices, a certificate-based secure boot operation is supported. The Certificate contains:

- Struct version: The version of the certificate structure.
- Public key: ECDSA-P256 public key, X and Y coordinates concatenated, used to validate the image.
- Certificate version: The version of the running certificate.
- Signature: ECDSA-P256 signature, used for the authentication of the public key and the certificate version.

The definition of the certificate struct can be found in `api/application_properties.h`.

To utilize certificate-based secure boot, configure Secure Engine to authenticate the bootloader image by configuring the certificate- based secure boot option in the Secure Engine OTP. Configure the Gecko Bootloader to enable certificate-based secure boot in the **Bootloader Core** component by selecting the **Enable certificate support** option. The Gecko Bootloader certificate must be signed by the private key pair of the public key stored in the Secure Engine OTP. For more information on the key storage, see the *Key Storage* section.

The certificate-based secure boot procedure is illustrated in the following figure.



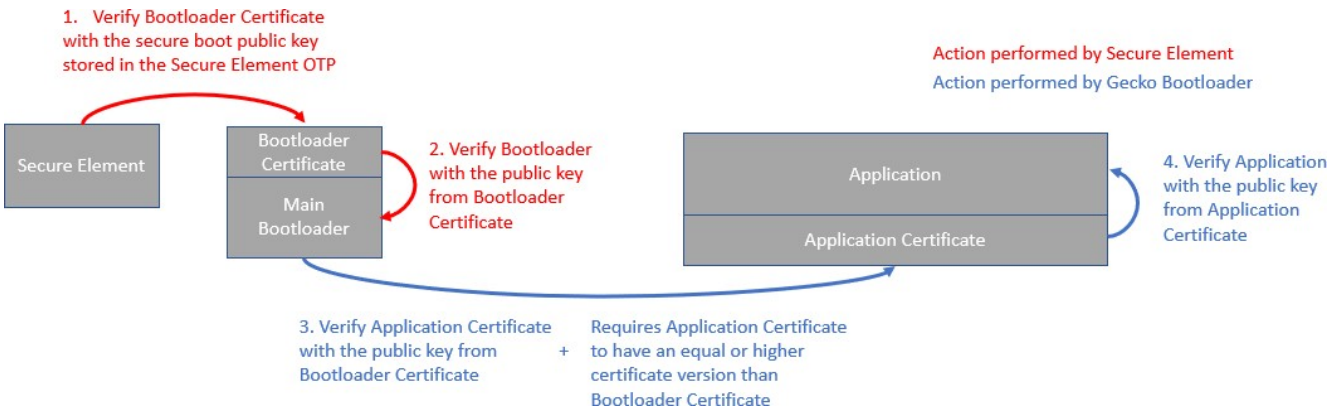
Once the certificate-based secure boot option on Secure Engine is turned on, Secure Engine verifies the Gecko Bootloader certificate. The public key stored in the certificate is used to validate the signature of the Gecko Bootloader. Secure Engine will not accept bootloader images without a certificate.

If only the secure boot option is enabled (not certificate-based) on Secure Engine, and Secure Engine identifies a certificate, the certificate will be used to validate the bootloader image. If the certificate version from the bootloader

Copyright © 2025 Silicon Laboratories. All rights reserved.

image is higher than 0 and it gets verified once, Secure Engine will never again accept direct signed bootloader images without a certificate.

The Gecko Bootloader will authenticate the direct signed application using the public key stored in the Gecko Bootloader certificate. If the application contains a certificate, Gecko Bootloader will authenticate it. The procedure is illustrated in the following figure.



After authentication of the application certificate, Gecko Bootloader verifies the signature of the application using the public key from the application certificate. In addition, Gecko Bootloader compares the Gecko Bootloader certificate version against the application certificate version. All application images with certificate version lower than the certificate version of the Gecko Bootloader will be rejected. Gecko Bootloader can be configured to only allow applications with certificates to boot by configuring the **Bootloader Core** component by selecting the **Reject direct signed images** option.

The **ApplicationProperties_t** struct contains the certificate struct **ApplicationCertificate_t**. The certificate struct can be injected to images that contain an **ApplicationProperties_t** with **ApplicationCertificate_t**. To inject a certificate to an image, issue the following command from Simplicity Commander:

```
commander convert <image file> --secureboot --keyfile <signing key> --certificate <certificate>
--outfile <signed image file with certificate>
```

Secure Firmware Upgrade

The Gecko Bootloader supports a secure firmware upgrade process. This is achieved by using symmetric encryption to encrypt the upgrade image, and asymmetric cryptography to sign the upgrade image. Symmetric encryption provides confidentiality, and asymmetric cryptography provides integrity and authenticity. Note that encryption alone is not enough to provide authenticity.

Encryption Algorithms

The Gecko Bootloader supports the AES-CTR-128 encryption algorithm. The GBL upgrade file is encrypted using 128-bit AES in Counter mode with a random nonce as the initial counter value.

To make use of the OTA decryption key stored in the Secure Engine OTP, the **Use symmetric key stored in Secure Engine storage** option in the **Bootloader Core** component must be selected. Simplicity Commander can be used to generate an OTA decryption key and write the key to the device. For more information on storing the OTA decryption key on Series 3 devices, see Simplicity Commander User Guide.

The Secure Engine OTP key support depends on the SE Manager component, which is enabled by default.

Signature Algorithms

The Gecko Bootloader supports the ECDSA-P256-SHA256 cryptographic signature algorithm. This is the ECDSA signature of the SHA-256 digest of the GBL upgrade file, using the NIST P-256 (secp256r1) curve.

Summary of Operation

Copyright © 2025 Silicon Laboratories. All rights reserved.

Before starting a firmware upgrade process, the application can verify an image in storage by calling into the bootloader verification functions. For bootloaders with Communication Interface, the host device should verify the image before sending it to the NCP or RCP.

During firmware upgrade, the GBL file is parsed, and if encrypted, decrypted on-the-fly. A GBL Signature Tag in the GBL file indicates to the bootloader that the file is signed, and the signature is verified. If signature verification fails, the firmware upgrade process is aborted.

On Series 3 devices, Gecko Bootloader will authenticate the GBL signature tag using the public key stored in the bootloader certificate if the **Enable certificate support** option is selected in the **Bootloader Core** component. A GBL Certificate tag in the GBL file indicates to the bootloader that the GBL certificate tag needs to be authenticated using the public key stored in the bootloader certificate. The certificate version in the GBL certificate tag is compared with the bootloader certificate and only a version equal or higher than the bootloader certificate is accepted. Once the GBL certificate tag is authenticated, the GBL file's signature is verified using the authenticated public key from the GBL certificate tag.

Using Application Image Security Features

This example assumes that a bootloader called **bootloader-uart-xmodem** has been built in Simplicity Studio.The relevant version can be flashed to the device using the Flash Programmer in Simplicity Studio or using Simplicity Commander.

This example provides two ways of signing the upgrade images. The first option uses Simplicity Commander to generate key material and sign data. This is suitable for development. The second option uses an external signer, such as a dedicated Hardware Security Module (HSM) to protect private key material and perform signing operations. Silicon Labs recommends using an HSM to safeguard private keys.

Generating Keys

To use the security features of the Gecko Bootloader, encryption and signing keys need to be generated. These keys must then be written to the device. The encryption key is used with the GBL file for secure firmware upgrade. The signing keys are used both with the GBL file for secure firmware upgrade and to sign the application image for Secure Boot.

Generating a Signing Key Using Simplicity Commander

```
commander util genkey --type ecc-p256 --privkey signing-key --pubkey signing-key.pub
```

This creates an ECDSA-P256 key pair for signing; `signing-key` contains the private key in PEM format and **must be kept secret from third parties**. This key will later be used to sign images and GBL files. `signing-key.pub` contains the public key in PEM format and can be used to verify GBL files using commander gbl4 info.

Generating a Signing Key Using a Hardware Security Module

When using a Hardware Security Module, the private key is kept secret inside the HSM. According to the instructions from your HSM vendor, have it generate an ECDSA-P256 key pair and export the public key in PEM format to the file **signing-key.pub**. Then use Simplicity Commander to convert the key to token format, suitable for writing to the Series 3 device.

```
commander gbl keyconvert --type ecc-p256 signing-key.pub
```

Generating an Encryption Key

```
commander util genkey --type aes-ccm --outfile encryption-key
```

This creates an AES-128 key for encryption in the file **encryption-key**.

Writing Keys to the Device

> **Note**: Refer to the sections, *Writing the AES Decryption Key* and *Writing the Public Key to the Device* in the

Copyright © 2025 Silicon Laboratories. All rights reserved.

> Simplicity Commander User Guide.

Signing an Application Image for Secure Boot

If the bootloader enforces Secure Boot, the application needs to be signed to pass verification. On every boot, a SHA-256 digest of the application is calculated. The signature is verified using ECDSA-P256, with the same public key as for the GBL file signing. Signature verification failure prevents the application from booting.

Application images should contain an **ApplicationProperties_t** struct declaring the application version, capabilities, and other metadata. If **ApplicationProperties_t** is missing, the application image cannot be signed. For more details on adding **ApplicationProperties_t**, see the *Application Properties* section on the *Application Interface* page.

Using Simplicity Commander

Signing the application can be done with the command:

```
commander convert myapp.s37 --secureboot --keyfile signing-key --outfile myapp-signed.s37
```

Using a Hardware Security Module

The application can be prepared for signing by issuing the command:

```
commander convert myapp.s37 --secureboot --extsign --outfile myapp-for-signing.s37
```

Using an HSM, sign the output file **myapp-for-signing.s37**, and supply the resulting DER-formatted signature file **signature.der** back to Simplicity Commander:

```
commander convert myapp-for-signing.s37 --secureboot --signature signature.der --verify signing-key.pub
--outfile myapp-signed.s37
```

Creating a Signed and Encrypted GBL Upgrade Image File from an Application

To create a GBL file from an application, use `commander gbl4 create`.

Note that, as of this writing, secure application images can only be constructed through Simplicity Commander, not through the configuration options available through Simplicity Studio.

Application images should contain an **ApplicationProperties_t** struct declaring the application version, capabilities, and other metadata. If **ApplicationProperties_t** is missing, the application image cannot be signed. For more details on adding **ApplicationProperties_t**, see the *Application Properties* section on the *Application Interface* page.

Using Simplicity Commander to Sign

For an application called **myapp.s37**, use:

```
commander gbl4 create myapp.gbl –-config configfile.yaml
```

Example of a config file is as follows:

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
manifest:
 product_id: "[16 B ID] (default from app or 00000000000000000000000000000000)"
 bundle_version: "[version] (default 0x00000000)"
 min_version: "[version] (default 0x00000000)"
 hash_function: "[none | sha256] (default sha256)"
 certificate: "[path to file] (optional)"
 signing_key: "[path to file] (optional)"
se_upgrade:
 se_file: "[path to file] (optional)"
updates:
 - data: <path to file>
  app_type: "[uint32] (default from app or 0x00000000)"
  app_capabilities: "[uint32] (default from app or 0x00000000)"
  version: "[uint32] (default from app or 0x00000000)"
  compression_scheme: "[none | lz4 | lzma] (default none)"
  encryption_scheme: "[none | aes-ctr-128] (default none)"
  block_size: "[size in bytes] (default 0/disabled)"
```

This single command performs three actions:

- Creates a GBL file
- Encrypts the GBL file
- Signs the GBL file

If Secure Boot is also desired, the application must be signed using `commander convert --secureboot` prior to creating the GBL.

System Security Considerations

The Gecko bootloader security features can be used to create a secure device, but do not create a secure system by themselves. This section goes over considerations that need to be taken when designing a secure system where the Gecko Bootloader is a component.

Key Storage

On Series 3 devices, the decryption key and the sign key used by the Gecko Bootloader in the Secure Engine OTP. The decryption key can be provisioned in the Secure Engine OTP using Simplicity Commander or using the Secure Engine Mailbox interface. Once a key value has been programmed into the Secure Engine OTP, it cannot be changed.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Application Interface

Application Interface

The bootloader has an application interface exposed through a function table in the bootloader. The application interface provides APIs to use bootloader functions for storing and retrieving upgrade images and verifying their integrity. APIs to reboot into the bootloader are also provided. For details see the Gecko Bootloader API Reference, at https://docs.silabs.com/gecko-platform/5.1.2/platform-bootloading-overview/folder.

If you are not using a protocol stack from Silicon Labs, the **api/btl_interface.h** header provides the bootloader application interface API. If you are using a protocol stack from Silicon Labs, the recommended bootloader interface API for the specific protocol stack should be used instead. The following files provide the implementation of the bootloader interface:

api/btl_interface.c (common interface)

api/btl_interface_storage.c (interface to storage functionality)

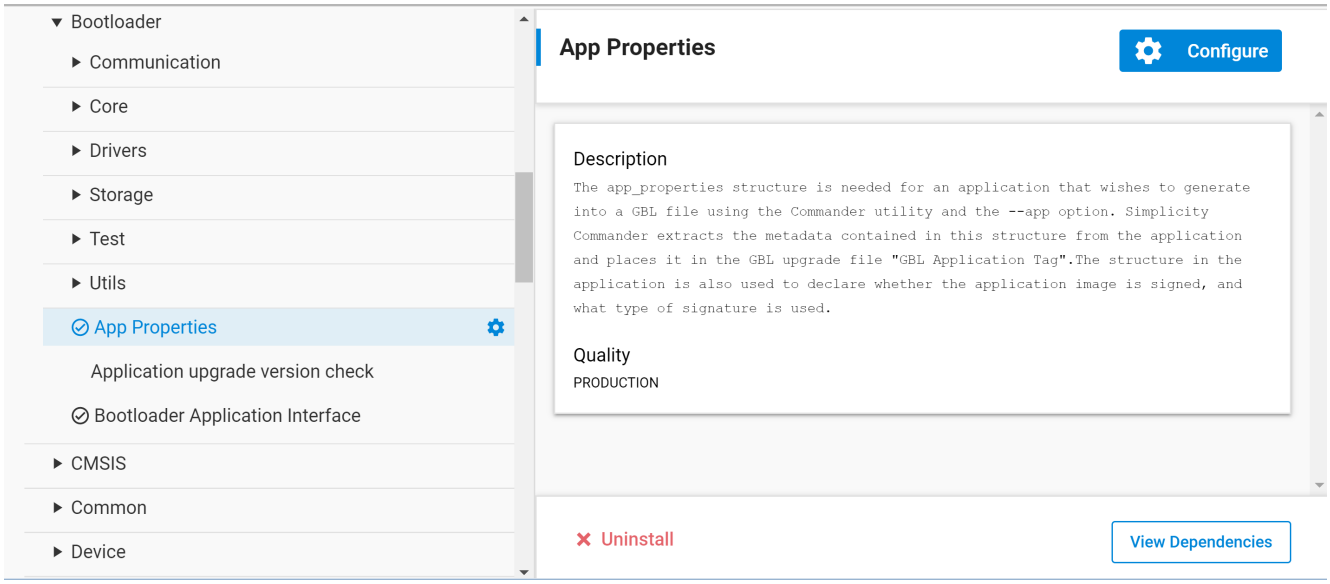
The application interface consists of functions that can be included in the customer application, and that communicate with the bootloader through the **MainBootloaderTable_t**. This table contains function pointers into the bootloader. The 10th word of the bootloader contains a pointer to this structure, allowing any application to easily locate it. Using the wrapper functions provided in the Bootloader Interface API is preferred over accessing the bootloader table directly. Modules include:

- **Application Storage Interface**: Application interface for interfacing with bootloader storage. The Storage Interface is only available on bootloaders that support the storage interface.
- **Common Application Interface**: Generic application interface available on all versions of the bootloader, independently of which components are present.

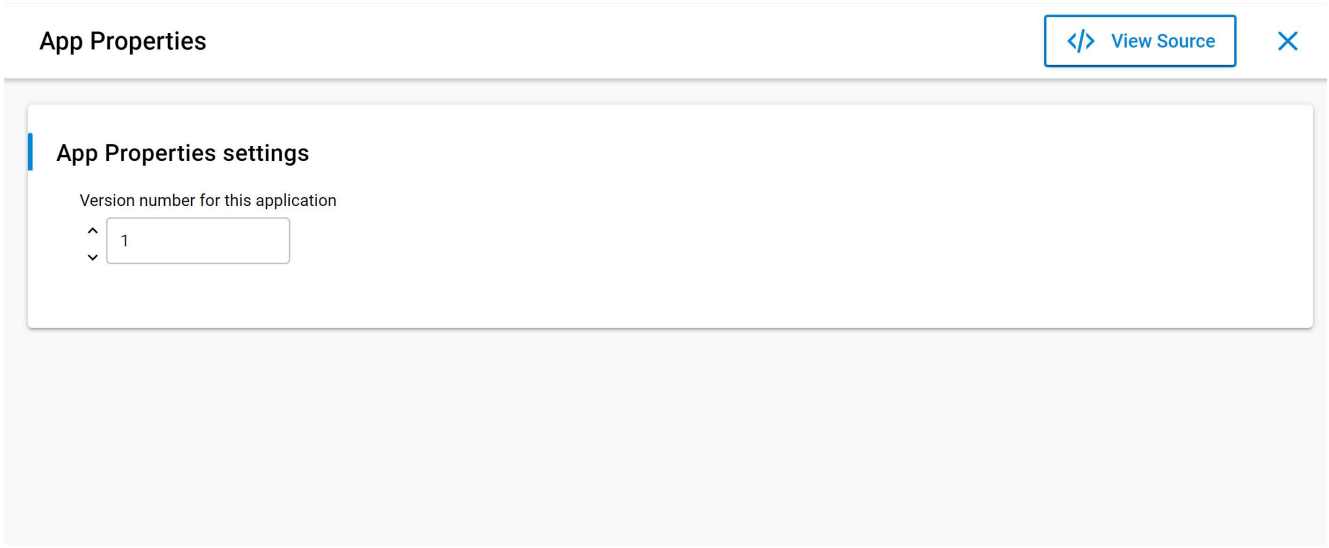
Application Properties

Application images should contain an **ApplicationProperties_t** struct declaring the application version, capabilities, and other metadata. The Simplicity Commander extracts the metadata contained in this structure from the application and places it in the UPDATE_MEMORY_SECTION tag in the GBL upgrade file. If the structure is not present in the application, Simplicity Commander will raise an error. The **ApplicationProperties_t** struct is added to the application on installing the **bootloader_interface** component to the application. The **bootloader_interface** component installs **bootloader_app_properties** component which adds an instance of **ApplicationProperties_t** named **sl_app_properties** to the project. The component adds a source file named **app_properties.c** and a configuration file named **app_properties_config.h**. This configuration file allows users to configure the application version via Simplicity Studio's Component Editor. To open the Component Editor, locate the **App Properties** component under **Platform > Bootloader** as shown below.

Copyright © 2025 Silicon Laboratories. All rights reserved.



Click **Configure** to open the Component Editor.



The application type is automatically populated based on the wireless stack used in building the project. The value of the application type is automatically set during the project generation step and can be found in **autogen/sl_application_type.h** file.

The contents of the **GBL Application Tag** can be extracted from a GBL file by a running application using the Application Storage interface. Note that the **GBL Application Tag** will only be added if the GBL file contains an application image, not if the GBL file only contains a bootloader upgrade or metadata.

The structure in the application is also used to declare whether the application image is signed, and what type of signature is used. This information is added by Simplicity Commander when signing the image using `commander convert (--secureboot, --extsign or -- signature)`. For the bootloader to locate the **ApplicationProperties_t** struct, if not already done by the linker, Simplicity Commander modifies word 13 of the application to insert a pointer to the **ApplicationProperties_t** struct when signing the application image for Secure Boot.

Error Codes

Copyright © 2025 Silicon Laboratories. All rights reserved.

Most Gecko bootloader APIs return error codes. The following table lists the groups of error codes that may be returned. The full list of error codes within each group can be found in *api/btl_errorcode.h* in the platform/bootloader directory of the SDK, as well as in the API Reference.

ID	Description
0x0	OK
0x01xx	Initialization error
0x02xx	Image verification error
0x04xx	Storage error
0x05xx	Bootload error
0x06xx	Security error
0x07xx	Communication error
0x09xx	XMODEM parser error
0x10xx	GBL file parser error
0x11xx	SPI slave driver error
0x12xx	UART driver error
0x13xx	Compression error
0x14xx	RTOS error

Copyright © 2025 Silicon Laboratories. All rights reserved.

Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher

Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher

Note: This section replaces *UG489: Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher*. Further updates to this user guide will be provided here.

This document describes the high-level implementation of the Silicon Labs Gecko Bootloader for EFM32 and EFR32 Series 1 and Series 2 microcontrollers, SoCs (System on Chips) and NCPs (Network Co-Processors), and provides information on different aspects of configuring the Gecko Bootloader. If you are not familiar with the basic principles of performing a firmware upgrade or want more information about upgrade image files, refer to *UG103.6: Bootloader Fundamentals*. For more information on using the Gecko Bootloader with different wireless stacks, see the following:

- Using the Gecko Bootloader with EmberZNet
- *UG435.06: Bootloading and OTA with Silicon Labs Connect v3.x*
- Using the Gecko Bootloader with Silicon Labs Bluetooth Applications

For more information on Series 2 device security, see:

- Series 2 Secure Boot with RTSL
- Series 2 Secure Debug
- *AN1222: Production Programming of Series 2 Devices*

For more information on security using Series 2 devices with Secure Vault, see:

- Anti-Tamper Protection Configuration and Use
- Authenticating Silicon Labs Devices using Device Certificates
- Secure Key Storage

Copyright © 2025 Silicon Laboratories. All rights reserved.

Overview

Overview

The Silicon Labs Gecko Bootloader is a common bootloader for all the newer MCUs and wireless MCUs from Silicon Labs. The Gecko Bootloader can be configured to perform a variety of functions, from device initialization to firmware upgrades. Key features of the bootloader are:

- Useable across Silicon Labs Gecko microcontroller and wireless microcontroller families
- In-field upgradeable
- Configurable
- Enhanced security features, including:
  - **Secure Boot**: When Secure Boot is enabled, the bootloader enforces cryptographic signature verification of the application image on every boot, using asymmetric cryptography. This ensures that the application was created and signed by a trusted party.
  - **Signed upgrade image file**: The Gecko Bootloader supports enforcing cryptographic signature verification of the upgrade image file. This allows the bootloader and application to verify that the application or bootloader upgrade comes from a trusted source before starting the upgrade process, ensuring that the image file was created and signed by a trusted party.
  - **Encrypted upgrade image file**: The image file can also be encrypted to prevent eavesdroppers from acquiring the plaintext firmware image.

The Gecko Bootloader uses a proprietary format for its upgrade images, called GBL (Gecko Bootloader file). These files have the file extension ".gbl". See section Gecko Bootloader File Format for more details.

On Series 1 devices, the Gecko Bootloader has a two-stage design, first stage and main stage, where a minimal first stage bootloader is used to upgrade the main bootloader. The first stage bootloader only contains functionality to read from and write to fixed addresses in internal flash. To perform a main bootloader upgrade, the running main bootloader verifies the integrity and authenticity of the bootloader upgrade image file. The running main bootloader then writes the upgrade image to a fixed location in internal flash and issues a reboot into the first stage bootloader. The first stage bootloader verifies the integrity of the main bootloader firmware upgrade image, by computing a CRC32 checksum before copying the upgrade image to the main bootloader location.

On Series 2 devices, the Gecko bootloader consists only of the main stage bootloader. The main bootloader is upgradable through the Secure Engine. The Secure Engine may be hardware-based, or virtual (software). If hardware-based, the implementation may be either with or without Secure Vault. Throughout this document, the following conventions will be used.

- HSE - Hardware Secure Engine, either with or without Secure Vault if not specified
- VSE - Virtual Secure Engine
- SE - Secure Engine (either HSE or VSE, in general)

The Secure Engine provides functionality to install an image to address 0x0 in internal flash, by copying from a configurable location in internal flash. This makes it possible to have a 2-stage design, where the main bootloader is not present. However, the presence of a main bootloader is assumed throughout this document.

To perform a main bootloader upgrade, the running main bootloader verifies the integrity and authenticity of the bootloader upgrade image file. The running main bootloader then writes the upgrade image to the upgrade location in flash and requests that the Secure Engine installs it. On some devices, the Secure Engine is also capable of verifying the authenticity of the main bootloader update image against a root of trust. The Secure Engine itself is upgradable using the same mechanism. See Gecko Bootloader Operation - Secure Engine Upgrade for more details.

The main bootloader consists of a common core, drivers, and a set of components that give the bootloader specific capabilities. The common bootloader core is now provided as a full-source delivery, as opposed to previously being a

Copyright © 2025 Silicon Laboratories. All rights reserved.

combination of compiled libraries and source code while the components continue to be delivered as source code. The common bootloader core contains functionality to parse GBL files and flash their contents to the device.

The Gecko Bootloader can be configured to perform firmware upgrades in standalone mode (also called a standalone bootloader) or in application mode (also called an application bootloader), depending on the component configuration. Components can be installed in and configured through the Simplicity Studio IDE.

A standalone bootloader uses a communications channel to get a firmware upgrade image. NCP (network co-processor) devices always use standalone bootloaders. Standalone bootloaders perform firmware image upgrades in a single-stage process that allows the application image to be placed into flash memory, overwriting the existing application image, without the participation of the application itself. In general, the only time that the application interacts with a standalone bootloader is when it requests to reboot into the bootloader. Once the bootloader is running, it receives packets containing the firmware upgrade image over-the-air using Bluetooth or by a physical connection such as UART or SPI. To function as a standalone bootloader with a physical connection, a component providing a communication interface such as UART or SPI must be configured.

An application bootloader relies on the application to acquire the firmware upgrade image. The application bootloader performs a firmware image upgrade by writing the firmware upgrade image to a region of flash memory referred to as the download space. The application transfers the firmware upgrade image to the download space in any way that is convenient (UART, over-the-air, Ethernet, USB, and so on). The download space is either an external memory device such as an EEPROM or dataflash or a section of the device's internal flash. The Gecko Bootloader can partition the download space into multiple storage slots and store multiple firmware upgrade images simultaneously. To function as an application bootloader, a component providing a bootloader storage implementation has to be configured.

Silicon Labs provides example bootloaders that come with a preconfigured set of installed components for configuration in either standalone or application mode. See section Configuring the Gecko Bootloader. The Silicon Labs Gecko SDK Suite also includes precompiled bootloader images for several different EFR32 devices. As of this writing, the images shown in the following table are provided.

> **Note**: The bootloader security features are not enabled in these precompiled images.

Table Prebuilt Bootloader Images

Use	Wireless Stack	Image Name	Mode	Interface
SoC	EmberZNet PRO	SPI Flash Storage Bootloader	Application	SPI Serial Flash
SoC	Bluetooth	Bluetooth In-Place OTA DFU Bootloader	Application	OTA/internal flash
NCP	EmberZNet PRO	UART XMODEM Bootloader	Standalone	UART (EZSP)
NCP	Bluetooth	BGAPI UART DFU Bootloader	Standalone	UART (BGAPI)

Note that on devices with a dedicated bootloader area (EFR32xG12 and later Series 1 devices), if the device is configured to boot to the bootloader area (that is, if bit 1 of the Config Lock Word 0 CLW0[1] is set), an image always must be present in the bootloader area. The device is factory-programmed with a dummy bootloader that simply jumps directly to the application in main flash. This means that when flashing a bootloader to a device with a dedicated bootloader area, this dummy bootloader is replaced. If later during development using the bootloader is no longer desired, CLW0[1] must be cleared or the dummy bootloader needs to be re-flashed. Platform-specific prebuilt dummy bootloader images are located in *./platform/bootloader/util/bin/*. Note that since the dummy bootloader only consists of a few instructions and doesn't pad out the remainder of the bootloader area, only the first flash page (where the first-stage bootloader resides) is overwritten, so the main stage bootloader would likely remain intact after programming the dummy bootloader. If desired, the rest of the flash pages in the bootloader area can then be erased separately.

On devices that do not have a dedicated bootloader area (EFR32xG1 and EFR32 Series 2), a dummy bootloader is not needed.

The following sections provide an overview of the Gecko Bootloader common core, drivers, and components. For details, including details on error codes and conditions, see the Gecko Bootloader API Reference, shipped with the SDK in the platform/bootloader/documentation folder.

Copyright © 2025 Silicon Laboratories. All rights reserved.

The bootloader area can be fully erased using the `commander device pageerase --region @bootloader` command with Simplicity Commander. In this state, the device will not boot until CLW0[1] is cleared or the dummy bootloader is flashed. For more information on how to use Simplicity Commander with Gecko bootloader, see section Simplicity Commander and the Gecko Bootloader.

Core

The bootloader core contains the bootloader's main functions. It also contains functionality to write to the internal flash, an image parser to parse and act upon the contents of GBL upgrade files, and functionality to boot the application in main flash.

The image parser can also optionally support the legacy Ember Bootloader (EBL) file format, but none of the security features offered by the Gecko Bootloader are supported if support for legacy EBL files is enabled.

A version of the GBL image parser without support for encrypted upgrade images is also available. This version can be used in flash space constrained bootloader applications where encryption of the upgrade image is not required.

Shared Memory

To exchange information between the bootloader and application, a section of SRAM is used. The contents of SRAM are preserved through a software reset, making the SRAM suitable as a communication channel between bootloader and application.

The shared memory has a size of 4 bytes, and is located at the first address of SRAM, 0x20000000. It is used to store a single word containing the reason for a reset. The structure of the reset cause word is defined in the Reset Information part of the Application Interface, in the file **btl_reset_info.h**, as 16 bits containing the reason, and 16 bits of signature indicating if the word is valid or not. If the signature reads 0xF00F, the reset reason is valid.

All 16-bit reset reasons used by Silicon Labs have the most significant bit set to zero. If custom reset reasons are desired, it is recommended to set the most significant bit to avoid conflicting definitions.

In addition to the reset causes defined in the Reset Information documentation, the bootloader will enter firmware upgrade mode if the shared memory contains the value 0x00000001. This value is supported for compatibility with certain legacy Bluetooth applications.

Drivers

Different applications of firmware upgrade require different hardware drivers for use by the other components of the bootloader.

Driver modules include:

- Delay: Simple delay routines for use with components that require small delays or timeouts.
- SPI: Simple, blocking SPI master implementation for communication with external devices such as SPI flashes.
- SPI Slave: Flexible SPI Slave driver implementation for use in communication components implementing SPI protocols. This driver supports both blocking and non-blocking operation, with DMA (Direct Memory Access) backing the background transfers to support non- blocking operation.
- UART: Flexible serial UART driver implementation for use in communication components implementing UART protocols. This driver supports both blocking and non-blocking operation, with DMA backing the background transfers to support non-blocking operation. Additionally, support for hardware flow control (RTS/CTS) is included.

Components

All parts of the bootloader that are either optional or that may be exchanged for different configurations are implemented as components. Each component may have a configuration header file, and one or more implementations. Components include:

- Communication
  - UART: XMODEM
  - UART: BGAPI
  - SPI: EZSP

Copyright © 2025 Silicon Laboratories. All rights reserved.

    - Bluetooth: AppLoader
- Compression
- Debug
- GPIO Activation
- Security
- Storage
    - Internal flash
    - External SPI flash

Communication

The Communication components provide an interface for implementing communication with a host device, such as a computer or a micro- controller. Several components implement the communication interface, using different transports and protocols.

- BGAPI UART DFU: By enabling the BGAPI communication component, the bootloader communication interface implements the UART DFU protocol using BGAPI commands. See *AN1053: Bluetooth® Device Firmware Update over UART for EFR32xG1 and BGM11x Series Products* for more information about this legacy bootloader.
- Bluetooth:AppLoader: By enabling the Bluetooth AppLoader communication component, the bootloader communication interface implements over-the-air device firmware upgrade functionality using Bluetooth. See Using the Gecko Bootloader with the Silicon Labs Bluetooth Applications for more information.
- EZSP-SPI: By enabling the EZSP-SPI communication component, the bootloader communication interface implements the EZSP protocol over SPI. This component makes the bootloader compatible with the legacy ezsp-spi-bootloader that was previously released with the EmberZNet wireless stack. See *AN760: Using the Ember Standalone Bootloader* for more information about legacy Ember standalone bootloaders.
- UART XMODEM: By enabling the UART XMODEM communication component, the bootloader communication interface implements the XMODEM-CRC protocol over UART. This component makes the bootloader compatible with the legacy serial-uart-bootloader that was previously released with the EmberZNet wireless stack. See *AN760: Using the Ember Standalone Bootloader* for more information about legacy Ember standalone bootloaders.

Compression

The Compression components provide capability for the bootloader GBL file parser to handle compressed GBL upgrade images. Each compression component provides support for one (de)compression algorithm. At the time of writing, decompression of data compressed with the LZ4 and LZMA algorithms is supported, through the *GBL Compression (LZ4)* and *GBL Compression (LZMA)* components.

Debug

This component provides the bootloader with support for debugging output. If the component is configured to enable debug prints, short debug messages will be printed over Serial Wire Output (SWO), which can be accessed in multiple ways, including using Simplicity Commander, and by connecting to port 4900 of the Wireless Starter Kit TCP/IP interface.

To turn on debug prints, enable the Debug component and select **Debug prints**. Select **Debug asserts** to enable assertions in the source code.

On Series 1 devices, also select the GPIO peripheral in the Pin Tool.

GPIO Activation

This component provides functionality to enter firmware upgrade mode automatically after reset if a GPIO pin is active during boot. The GPIO pin location and polarity are configurable.

- GPIO: By enabling the GPIO activation component, the firmware upgrade mode can be activated by the push buttons.
- EZSP GPIO: The EZSP communication protocol over SPI can be used together with this component. By enabling the EZSP GPIO component, the firmware upgrade mode can be entered by activating the *nWake* pin.

Security

Security components provide implementations of cryptographic operations as well as functionality to compute checksums and to read cryptographic keys from manufacturing tokens.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Modules include:

- AES: AES decryption functionality
- CRC16: CRC16 functionality
- CRC32: CRC32 functionality
- ECDSA: ECDSA signature verification functionality
- SHA-256: SHA-256 digest functionality

Storage

These components provide the bootloader with multiple storage options for SoCs. All storage implementations have to provide an API to access image files to be upgraded. This API is based on the concept of dividing the download space into storage slots, where each slot has a predefined size and location in memory and can be used to store a single upgrade image. Some storage implementations also support a raw storage API to access the underlying storage medium. This can be used by applications to store other data in parts of the storage medium that are not used for storing firmware upgrade images. Implementations include:

- **Internal Flash**: The internal flash storage implementation uses the internal flash of the device for upgrade image storage. Note that this storage area is only a download space and is separate from the portion of internal flash used to hold the active application code.
- **SPI Flash**: Two components are available for SPI Flash storage implementation.
    1. **SPI Flash Storage**: The SPI flash storage implementation supports a variety of SPI flash parts. The subset of devices supported can be configured in this component in Gecko Bootloader. (The default configuration if no devices are selected is to include drivers for all supported parts.) Including support for multiple devices requires more flash space in the bootloader. The SPI flash storage implementation does not support any write protection functionality. Supported SPI flash parts are shown in Table 1.2.
    2. **SPI Flash Storage SFDP**: The SPI Flash storage SFDP implementation supports all the SPI flash parts that support SFDP (Serial Flash Description Parameter) Standard from JEDEC. The SPI Flash type is detected automatically by querying the SFDP Parameter table present within the flash memory. All the properties of the SPI Flash are accessed from this parameter table. This component is not configurable since the flash is automatically detected. Any flash that supports the SFDP standard can be used with the Gecko Bootloader. However, performance delays might occur when compared to using the **SPI Flash Storage** component.

> **Note**: Low power devices are recommended for battery-operated applications. Use of the other listed devices will decrease battery life due to higher quiescent current, but this can be mitigated with external shutdown FET circuitry, if desired.

Table Supported Serial Dataflash/EEPROM External Memory Parts

Manufacturer Part Number	Size (kB)	Quiescent Current (µA Typical)*
Macronix MX25R8035F (low power)	1024	0.007
Macronix MX25R6435SF (low power)	8192	0.007
Macronix MX25R3235F (low power)	4096	0.007
Spansion S25FL208K	1024	15
Winbond W25X20BVSNIG (W25X20CVSNJG for high-temperature support)	256	1
Winbond W25Q80BVSNIG (W25Q80BVSNJG for high-temperature support)	1024	1
Macronix MX25L2006EM1I-12G (MX25L2006EM1R-12G for high-temperature support)	256	2
Macronix MX25L4006E	512	2
Macronix MX25L8006EM1I-12G (MX25L8006EM1R-12G for high-temperature support)	1024	2

Copyright © 2025 Silicon Laboratories. All rights reserved.

Manufacturer Part Number	Size (kB)	Quiescent Current (µA Typical)*
Macronix MX25L1606E	2048	2
Macronix MX25U1635E (2V)	2048	2
Atmel/Adesto AT25DF041A	512	15
Atmel/Adesto AT25DF081A	1024	5
Atmel/Adesto AT25SF041	512	2
Micron (Numonyx) M25P20	256	1
Micron (Numonyx) M25P40	512	1
Micron (Numonyx) M25P80	1024	1
Micron (Numonyx) M25P16	2048	1
ISSI IS25LQ025B	32	8
ISSI IS25LQ512B	64	8
ISSI IS25LQ010B	126	8
ISSI IS25LQ020B	256	8
ISSI IS25LQ040B	512	8

* Quiescent current values are as of December 2017; check the latest part specifications for any changes.

Compatibility

1. Silicon Labs recommends building the Gecko Bootloader and the application using the same GSDK. This implies that, if the Gecko Bootloader has been built using GSDK v4.0, it is recommended that the application also be built using GSDK v4.0.
2. Backward compatibility is supported wherein the Gecko Bootloader is built using a previous SDK version and the application is built using a newer SDK version.
3. In general, Silicon Labs does not recommend building the Gecko Bootloader using a newer SDK version with the application built using an older SDK version.

The below table attempts to explain the above-mentioned guidelines:

SDK version X < SDK version Y	Application built using SDK version X	Application built using SDK version Y
Gecko Bootloader built using SDK version X	Recommended	Compatible
Gecko Bootloader built using SDK version Y	Not recommended	Recommended

Copyright © 2025 Silicon Laboratories. All rights reserved.

Gecko Bootloader File Format

Gecko Bootloader File Format

The GBL file format is used by the Gecko Bootloader. File formats described in this section are generated by Simplicity Commander commands. For more information, see Simplicity Commander Reference Guide.

File Structures

The GBL file format is composed of several tags that indicate a format of the subsequent data and the length of the entire tag. The format of a tag is as follows:

Tag ID	Tag Length	Tag Payload
4 bytes	4 bytes	Variable (according to tag length)

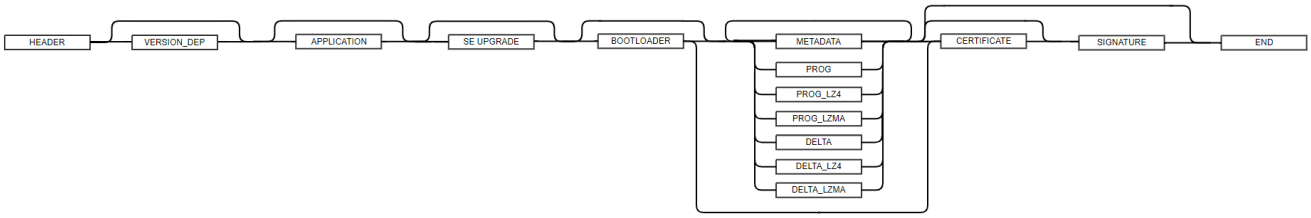
Plaintext Tag Description

Tag Name	ID	Description
GBL Header Tag	0x03A617EB	This must be the first tag in the file. The header tag contains the version number of the GBL file specification, and flags indicating the type of GBL file - whether it is signed or encrypted.
GBL Version Dependency Tag	0x76A617EB	This optional tag contains encoded version dependencies that the software currently running on the device must satisfy before an upgrade can be attempted. Only available on Series 2 devices.
GBL Application Info Tag	0xF40A0AF4	This tag contains information about the application update image that is contained in this GBL file
GBL SE Upgrade Tag	0x5EA617EB	This tag contains a complete encrypted Secure Element update image. Only applicable on Series 2 devices.
GBL Bootloader Tag	0xF50909F5	This tag contains a complete bootloader update image.
GBL Program Data Tag	0xFE0101FE / 0xFD0303FD	This tag contains information about what application data to program at a specific address into the main flash memory.
GBL Delta Tag	0xF80A0AF8UL	This tag contains the information about the delta patch that should be used to create the new app.
GBL Program LZ4 Compressed Data Tag	0xFD0505FD	This tag contains LZ4 compressed information about what application data to program at a specific address into the main flash memory.
GBL Delta LZ4 Compressed Data Tag	0xF80B0BF8UL	This tag contains LZ4 compressed information about the delta patch that should be used to create the new app.
GBL Program LZMA Compressed Data Tag	0xFD0707FD	This tag contains LZMA compressed information about what application data to program at a specific address into the main flash memory.
GBL Delta LZMA Compressed Data Tag	0xF80C0CF8UL	This tag contains LZMA compressed information about the delta patch that should be used to create the new app.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Tag Name	ID	Description
GBL Metadata Tag	0xF60808F6	This tag contains metadata that the bootloader does not parse but can be returned to the application through a callback.
GBL Certificate Tag	0xF30B0BF3	This tag contains a certificate that will be used to verify the authenticity of the GBL file.
GBL Signature Tag	0xF70A0AF7	This tag contains the ECDSA-P256 signature of all preceding data in the file.
GBL End Tag	0xFC0404FC	This tag indicates the end of the GBL file. It contains a 32-bit CRC for the entire file as an integrity check. The CRC is a non-cryptographic check. This must be the last tag.

The allowed sequence of GBL tags in a GBL file is shown in the following figure.

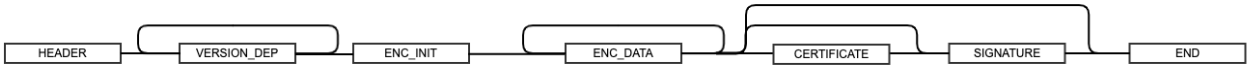


Encrypted Tag Descriptions

The encrypted GBL file format is like the unencrypted version. It introduces several new tags.

Tag Name	ID	Description
GBL Header Tag	0x03A617EB	The GBL header is the same as for a plaintext GBL file, but the flag indicating that the GBL file is encrypted must be set.
GBL Encryption Init Header	0xFA0606FA	This contains information about the image encryption such as the Nonce and the amount of encrypted data.
GBL Encrypted Program Data	0xF90707F9	This contains an encrypted payload containing a plaintext GBL tag, one of Application Info, Bootloader, Metadata or Program Data. The data is encrypted using AES-CTR-128.

The allowed sequence of GBL tags in an encrypted GBL file is shown in the following figure.



GBL Tag Data Structures and Definitions

GBL Tag Header

```
typedef struct {

uint32_t  tagId;  // Tag ID representing the type of tag (GBL Header Tag, GBL Bootloader Tag, etc.).

uint32_t  length; // Length of the subsequent tag data in bytes.
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
} GblTagHeader_t;
```

GBL Header Tag

```
typedef struct {
GblTagHeader_t header;  // Tag header.
uint32_t      version; // Version of the GBL file format specification. E.g. 0×03000000.
uint32_t      type;    // Flags indicating whether the GBL file is encrypted and/or signed.
// See definitions below.
} GblHeader_t;
```

```
// GBL types
#define GBL_TYPE_NONE               0x00000000UL
#define GBL_TYPE_ENCRYPTION_AESCCM  0x00000001UL
#define GBL_TYPE_SIGNATURE_ECDSA    0x00000100UL
```

GBL Version Dependency Tag

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
typedef struct {
uint8_t  imageType;  // Type of image (application, bootloader, SE)
uint8_t  statement;  // Encoded dependency statement (ex. appVersion > (0).1.2.3)
uint16_t reserved;   // Reserved
uint32_t version;    // The version number used in the statement (ex. (0).1.2.3)
} VersionDependency_t;
// Image types
#define GBL_VERSION_DEPENDENCY_TYPE_APPLICATION          0x01U
#define GBL_VERSION_DEPENDENCY_TYPE_BOOTLOADER           0x02U
#define GBL_VERSION_DEPENDENCY_TYPE_SE                   0x03U
// Operator encoding
#define GBL_VERSION_DEPENDENCY_OPERATOR_MASK             0x0FU
#define GBL_VERSION_DEPENDENCY_OPERATOR_SHIFT            0x00U
#define GBL_VERSION_DEPENDENCY_OPERATOR_TYPE_MASK        0x0EU
#define GBL_VERSION_DEPENDENCY_OPERATOR_NEGATOR_BIT_MASK 0x01U
#define GBL_VERSION_DEPENDENCY_OPERATOR_LT               0x00U
#define GBL_VERSION_DEPENDENCY_OPERATOR_LEQ              0x02U
#define GBL_VERSION_DEPENDENCY_OPERATOR_EQ               0x04U
#define GBL_VERSION_DEPENDENCY_OPERATOR_GEQ              0x06U
#define GBL_VERSION_DEPENDENCY_OPERATOR_GT               0x08U
// Connective encoding
#define GBL_VERSION_DEPENDENCY_CONNECTIVE_MASK           0xF0U
#define GBL_VERSION_DEPENDENCY_CONNECTIVE_SHIFT          0x04U
#define GBL_VERSION_DEPENDENCY_CONNECTIVE_TYPE_MASK      0x0EU
#define GBL_VERSION_DEPENDENCY_CONNECTIVE_NEGATOR_BIT_MASK 0x01U
#define GBL_VERSION_DEPENDENCY_CONNECTIVE_AND            0x00U
// SE version mask for ignoring the compatibility byte when comparing versions
#define GBL_VERSION_DEPENDENCY_SE_VERSION_MASK           0x00FFFFFFUL
```

GBL Application Info Tag

```
typedef struct {
GblTagHeader_t    header;  // Tag header.
ApplicationData_t appInfo; // Application information structure. See definition below.
} GblApplication_t;
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
typedef struct ApplicationData {
uint32_t type;          // Bitfield representing the type of application.
// See definitions below for possible values.
uint32_t version;       // Version number for this application (customer-defined).
uint32_t capabilities;  // Bitfield representing the capabilities of this application.
uint8_t  productId[16]; // Unique ID (UUID or GUID) for the product this application is built for.
} ApplicationData_t;
```

```
// Application types
#define APPLICATION_TYPE_ZIGBEE         (1UL << 0UL)
#define APPLICATION_TYPE_THREAD         (1UL << 1UL)
#define APPLICATION_TYPE_FLEX           (1UL << 2UL)
#define APPLICATION_TYPE_BLUETOOTH      (1UL << 3UL)
#define APPLICATION_TYPE_MCU            (1UL << 4UL)
#define APPLICATION_TYPE_BLUETOOTH_APP  (1UL << 5UL)
#define APPLICATION_TYPE_BOOTLOADER     (1UL << 6UL)
#define APPLICATION_TYPE_ZWAVE          (1UL << 7UL)
```

GBL SE Upgrade Tag

```
typedef struct {
GblTagHeader_t header;  // Tag header.
uint32_t      blobSize; // Size of the SE upgrade data blob.
uint32_t      version;  // Version of the SE image.
uint8_t       data[];   // Array of data containing the SE upgrade blob.
} GblSeUpgrade_t;
```

GBL Bootloader Tag

```
typedef struct {
GblTagHeader_t header;          // Tag header.
uint32_t      bootloaderVersion; // Version number of the bootloader.
uint32_t      address;          // Base address of the bootloader image.
uint8_t       data[];           // Array of data containing the bootloader upgrade image.
} GblBootloader_t;
```

GBL Program Data Tag

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
typedef struct {
GblTagHeader_t header;          // Tag header.
uint32_t       flashStartAddress; // Address at which to start flashing data.
uint8_t        data[];          // Array of data to be flashed
// (compressed data in the LZ4 and LZMA variants of the tag).
} GblProg_t;
```

GBL Metadata Tag

```
typedef struct {
GblTagHeader_t header;    // Tag header.
uint8_t        metadata[]; // Array containing the metadata.
} GblMetadata_t;
```

GBL Certificate Tag

```
typedef struct {
GblTagHeader_t          header;     // Tag header.
ApplicationCertificate_t certificate; // Certificate used to verify the GBL file. See definition below.
} GblCertificateEcdsaP256_t;
typedef struct ApplicationCertificate {
uint8_t  structVersion;    // Version of the certificate structure.
uint8_t  flags[3];         // Reserved flags.
uint8_t  key[64];          // Public key used to verify the GBL file.
uint32_t version;          // Version number of this certificate.
uint8_t  signature[64];    // The signature of the certificate itself (not the GBL file).
} ApplicationCertificate_t;
```

GBL Signature Tag

```
typedef struct {
GblTagHeader_t header; // Tag header.
uint8_t        r[32];  // The r value of the ECDSA signature.
uint8_t        s[32];  // The s value of the ECDSA signature.
} GblSignatureEcdsaP256_t;
```

GBL End Tag

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
typedef struct {
GblTagHeader_t header; // Tag header.
uint32_t       gblCrc; // CRC32 checksum of the entire GBL file.
} GblEnd_t;
```

GBL Encryption Init Header Tag

```
typedef struct {
GblTagHeader_t header;    // Tag header.
uint32_t       msgLen;    // Length of the ciphertext in bytes.
uint8_t        nonce[12]; // Random AES-CTR nonce.
} GblEncryptionInitAesCcm_t;
```

GBL Encrypted Data Tag

```
typedef struct {
GblTagHeader_t header;            // Tag header.
uint8_t        encryptedGblData[]; // Array containing the ciphertext (encrypted GBL data).
// Note that the corresponding plaintext must contain one or
// more complete GBL tags. Put differently: A single plaintext GBL
// tag cannot be split across multiple encrypted tags.
} GblEncryptionData_t;
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Gecko Bootloader Operation Application Upgrade

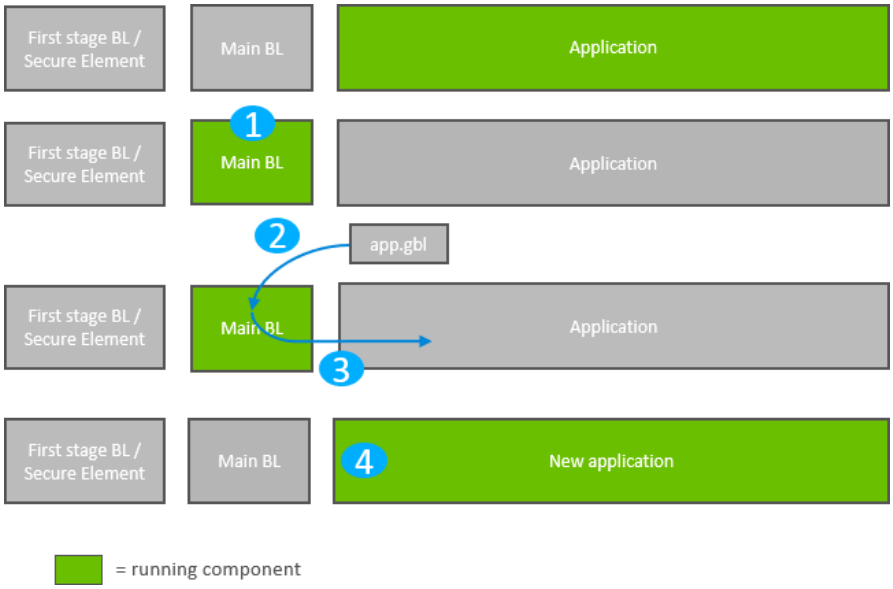
Gecko Bootloader Operation - Application Upgrade

This section summarizes Gecko Bootloader operation for updating application firmware, first if the Gecko Bootloader is configured in standalone mode and then if it is configured in application mode. Section Gecko Bootloader Operation - Bootloader Upgrade provides the same information for updating the bootloader firmware.

The figures that illustrate Gecko Bootloader operation in this section do not provide information about the bootloader memory layouts for different devices. For more details refer to the section "Memory Space for Bootloading" in *UG103.6: Bootloader Fundamentals*.

Standalone Bootloader Operation

Standalone bootloader operation is illustrated in the following figure:



1. The device reboots into the bootloader.
2. A GBL file containing an application image is transmitted from the host to the device. If image encryption is enabled in the main stage bootloader and the image is encrypted, decryption is performed during the process of receiving and parsing the GBL file.
3. The bootloader applies the application upgrade from the GBL upgrade file on-the-fly. If image authentication is enabled in the main stage bootloader and the GBL file contains a signature, the authenticity of the image is verified before completing the process.
4. The device boots into the application. Application upgrade is complete.

Rebooting Into the Bootloader

The Gecko Bootloader supports multiple mechanisms for triggering the bootloader. If the **GPIO activation** component is installed, the host device can keep this pin low/high (depending on configuration) through reset to make the device enter the bootloader. The bootloader can also be entered through software. The `bootloader_rebootAndInstall` API first signals to the bootloader that it should enter firmware upgrade mode by writing a command to the shared memory location at the

Copyright © 2025 Silicon Laboratories. All rights reserved.

bottom of SRAM, and then performs a software reset. If the bootloader finds the correct command in shared memory upon boot, it will enter firmware upgrade mode instead of booting the existing application.

Downloading and Applying a GBL Upgrade File

When the bootloader enters firmware upgrade mode, it enters a receive loop waiting for data from the host device. The specifics of the receive loop depend on the protocol. Received packets are passed to the image parser, a state machine that parses the data and returns a callback containing any data that should be acted upon. The bootloader core implements this callback and flashes the data to internal flash at the address specified. If GBL file authentication or encryption is enabled, the image parser will enforce this, and abort the image upgrade if the authentication fails.

The bootloader prevents a newly uploaded image from being bootable by holding back parts of the application vector table until the GBL file CRC and GBL signature (if required) have been verified.

Booting Into the Application

When an application upgrade is completed, the bootloader triggers a reboot with a message in shared memory at the bottom of SRAM signaling that an application upgrade has been successfully completed. The application can use this reset information to learn that an application upgrade was just performed.

Before jumping to the main application, the bootloader verifies that the application is ready to run. This includes verifying if the Program counter and Stack Pointer are valid. If secure boot is enabled, the bootloader expects a signed application and attempts to validate the signature of the application. In scenarios where secure boot is not enabled, the bootloader attempts to validate if the Application properties pointer points to valid app properties structure in the flash. If a valid app properties struct is found, the bootloader proceeds based on the signature type indicated by the application properties struct or else the bootloader assumes that the Application properties pointer points to the Reset Handler of the application (an app without application properties) and proceeds to boot into the application. In case the verification of the application fails at any stage, the bootloader enters the bootload mode instead of booting into the application.

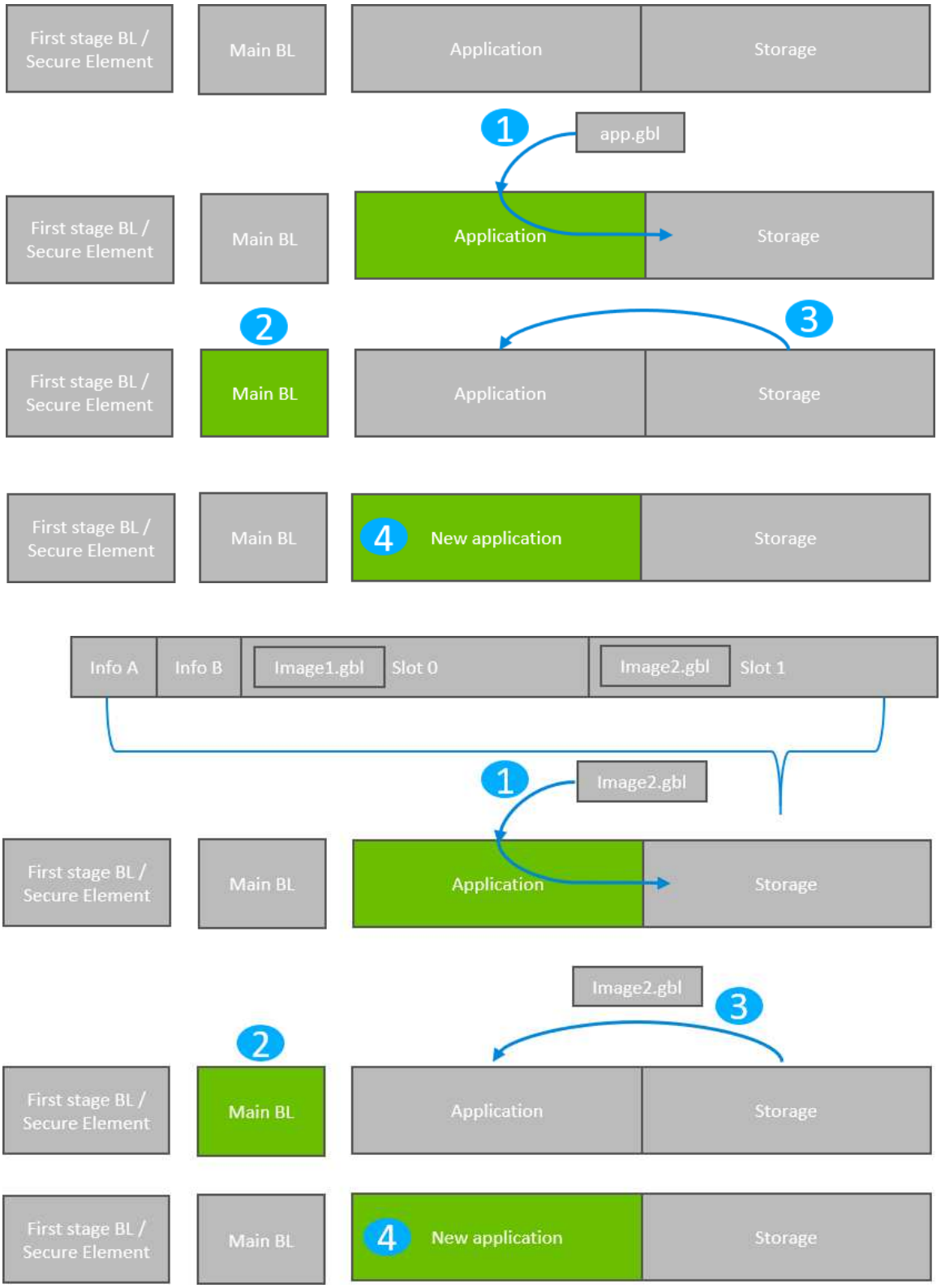
Error Handling

If the application upgrade is interrupted at any time, the device will be without a working application. The bootloader then resets the device, and re-enters firmware upgrade mode. The host device can easily restart the application upgrade process, to try loading the upgrade image again.

Application Bootloader Operation

The following figure illustrates the application bootloader operation both for a single image/single storage slot, and multiple images/ multiple storage slots.

Copyright © 2025 Silicon Laboratories. All rights reserved.



1. A GBL file is downloaded onto the storage medium of the device (internal flash or external dataflash), as described below, and the presence of an upgrade image is indicated.
2. The device reboots into the bootloader, and the bootloader enters firmware upgrade mode.
3. The bootloader applies the application upgrade from the GBL upgrade file.
4. The device boots into the application. Application upgrade is complete.

Downloading and Storing a GBL Upgrade Image File

Copyright © 2025 Silicon Laboratories. All rights reserved.

To prepare for receiving an upgrade image, the application finds an available storage slot, or erases an existing one using `bootloader_eraseStorageSlot`. If the bootloader only supports a single storage slot, a value of 0 should be used for the slot ID.

The application then receives a GBL file using an applicable protocol, such as Ethernet, USB, Zigbee, OpenThread, or Bluetooth, and stores it in the slot by calling `bootloader_writeStorage`.

When download is complete, the application can optionally verify the integrity of the GBL file by calling `bootloader_verifyImage`. This is also done by the bootloader before applying the image but can be done from the application in order to avoid rebooting into the bootloader if the received image was corrupt.

If multiple storage slots are supported, the application should write a bootload list by calling `bootloader_setImageToBootload`. The list is written to the two bootload info pages as shown in Figure The bootload list is a prioritized list of slots indicating the order the bootloader should use when attempting to perform a firmware upgrade. The bootloader attempts to verify the images in these storage slots in sequence and applies the first image to pass verification. If only a single storage slot is supported, the bootloader treats the entire download space as a single storage slot.

Rebooting and Applying a GBL Upgrade File

The bootloader can be entered through software. The `bootloader_rebootAndInstall` API signals to the bootloader that it should enter firmware upgrade mode by writing a command to the shared memory location at the bottom of SRAM, and then performs a software reset. If the bootloader finds the correct command in shared memory upon boot, it enters firmware upgrade mode instead of booting the existing application.

The bootloader iterates over the list of storage slots marked for bootload and attempts to verify the image stored in each. Once it finds a valid GBL upgrade file, firmware upgrade is attempted from this GBL file. If the upgrade fails, the bootloader moves to the next image in the list. If no images pass verification, the bootloader reboots back into the existing application with a message in the shared memory location in SRAM indicating that no good upgrade images were found.

Booting Into the Application

When an application upgrade is completed, the bootloader triggers a reboot with a message in shared memory at the bottom of SRAM signaling that an application upgrade has been successfully completed. The application can use this reset information to learn that an application upgrade was just performed.

Before jumping to the main application, the bootloader verifies that the application is ready to run. This includes verifying if the Program counter and Stack Pointer are valid. If secure boot is enabled, the bootloader expects a signed application and attempts to validate the signature of the application. In scenarios where secure boot is not enabled, the bootloader attempts to validate if the Application properties pointer points to valid app properties structure in the flash. If valid app properties struct is found, the bootloader proceeds based on the signature type indicated by the application properties struct or else the bootloader assumes that the Application properties pointer points to the Reset Handler of the application (an app without application properties) and proceeds to boot into the application. In case the verification of the application fails at any stage, the bootloader enters the bootload mode instead of booting into the application.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Gecko Bootloader Operation Bootloader Upgrade

Gecko Bootloader Operation - Bootloader Upgrade

Bootloader upgrade functionality is provided by the first stage bootloader on Series 1 devices, or the Secure Engine on Series 2 devices. The Secure Engine itself is also upgradable. For more details, see section Gecko Bootloader Operation - Secure Engine Upgrade. On Series 1 devices, the first stage bootloader is not upgradable.

Requirements for upgrading the main bootloader vary depending on the bootloader configuration:

- Application bootloader with storage: Upgrading the main bootloader requires a single GBL file containing both bootloader and application upgrade images.
- Standalone bootloader with communication interface: Upgrading the bootloader requires two GBL files, one with only the bootloader upgrade image, and one with only the application upgrade image.

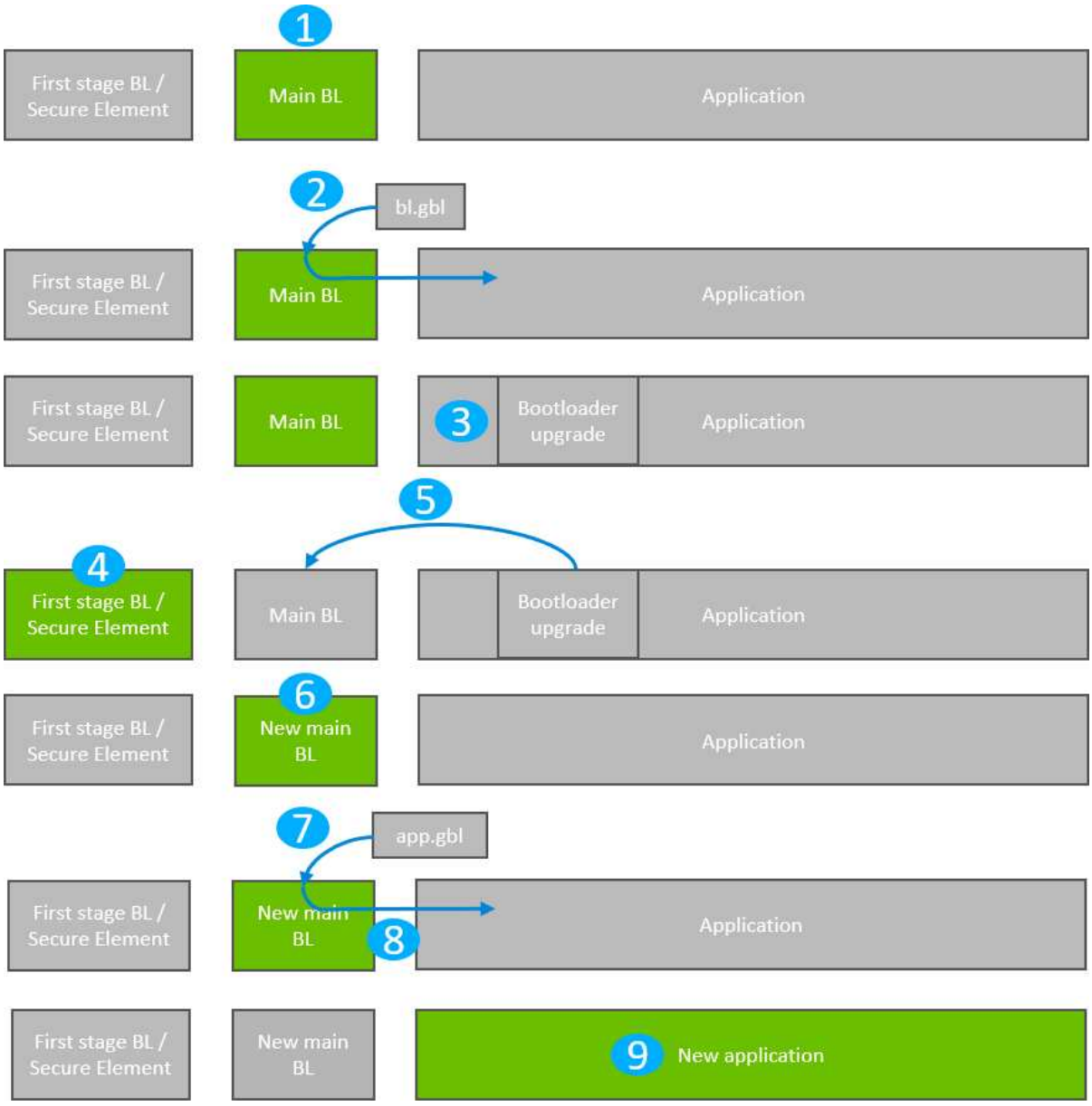
Security of the bootloader upgrade process is provided by signing the GBL file. See section Creating a Signed and Encrypted GBL Upgrade Image File from an Application.

The figures that illustrate Gecko Bootloader operation in this section do not provide information about the bootloader memory layouts for different devices. For more details refer to the “Memory Space for Bootloading” section in *UG103.6: Bootloader Fundamentals*. For convenience, the figures do not distinguish between SE and VSE.

Bootloader Upgrade on Bootloaders with Communication Interface (Standalone Bootloaders)

The process is illustrated in the following figure:

Copyright © 2025 Silicon Laboratories. All rights reserved.



1. The device reboots into the bootloader.
2. A GBL file containing only a bootloader upgrade image is transmitted from the host to the device.
3. The contents of the GBL Bootloader tag are written to the bootloader upgrade location in internal flash, overwriting the existing application.
4. The device reboots into the first stage bootloader / Secure Engine.
5. The first stage bootloader / Secure Engine replaces the main bootloader with the new version found in the bootloader upgrade location.
6. The device boots into the new main bootloader.
7. A GBL file containing only an application image is transmitted from the host to the device.
8. The bootloader applies the application image from the GBL upgrade file on-the-fly.
9. The device boots into the application. Bootloader upgrade is complete.

A bootloader upgrade is started in the same way as an application upgrade.

Downloading and Applying a Bootloader GBL Upgrade File

When the bootloader has entered the receive loop, a GBL upgrade file containing a bootloader upgrade is transmitted to the bootloader. When a packet is received, it is passed to the image parser. The image parser parses the data and returns

Copyright © 2025 Silicon Laboratories. All rights reserved.

bootloader upgrade data in a callback. The bootloader core implements this callback and flashes the data to internal flash at the bootloader upgrade location.

The bootloader prevents a newly uploaded bootloader upgrade image from being interpreted as valid by holding back parts of the bootloader upgrade vector table until the GBL file CRC and GBL signature (if required) have been verified.

When a complete bootloader upgrade image is received, the main bootloader signals the first stage bootloader / Secure Engine that it should enter firmware upgrade mode. On Series 1 devices, this is done by writing a command to the shared memory location at the bottom of SRAM, and then performing a software reset. On Series 2 devices, Secure Engine communication is used to signal that bootloader upgrade is ready to be performed.

On Series 1 devices, the first stage bootloader verifies the CRC on the current main bootloader and verifies the CRC of the bootloader upgrade present in the bootloader upgrade location in internal flash.

- If the CRC in the upgrade image location and the CRC in the current main bootloader are both valid, then the upgrade image will be copied over the main bootloader if the version number of the upgrade is higher than the current main bootloader version.
- If the CRC in the upgrade image location is valid and the CRC in the current main bootloader location is not valid, the upgrade image will be copied over the main bootloader regardless of the version. This is because the version of the main bootloader cannot be relied upon if the main bootloader image is corrupt.
- If the CRC in the upgrade location is not valid, the upgrade will not be applied.

On Series 2 devices, the authenticity of the main bootloader optionally can be verified before applying the bootloader upgrade. See section Setting a Version Number for more information about versioning bootloader images.

Upgrading Bootloaders Without Secure Boot to Bootloaders with Secure Boot

A bootloader without the secure boot feature can be upgraded to a bootloader with the secure boot feature, using the following procedure:

1. Prepare a Gecko Bootloader image with secure boot enabled. The version of the bootloader needs to be higher than the bootloader on the device.
   - Turn on secure boot in Simplicity Studio by going to the **Bootloader Core** component and selecting the **Enable secure boot** option.
   - (Optional) In the **Bootloader Core** component, select the **Require signed firmware upgrade files** option. This means that the Gecko Bootloader will only accept signed GBL files.
2. Generate a public/private Signing Key pair. See section Generating Keys for more information on creating a Signing Key pair.
3. Write the public key generated from the previous step to the device. The public key is stored as a manufacturing token in the device by default. This key can be written by application code running on the device as long as the Lock Bits page is configured to allow flash writes. If the Lock Bits page is locked, it can only be erased by the debugger. Therefore, signing/decryption keys residing in the Lock Bits page cannot be erased from firmware. This means that, for devices in the field, those areas in flash cannot be replaced with new ones. However, the Gecko Bootloader prepared from step 1 can be modified to look for the decryption and signature keys in a different location. Key locations are defined in the bootloader project file `btl_security_tokens.c`.
4. Create a GBL file using the Gecko Bootloader image. The GBL file needs to be signed/unsigned depending on the current configuration of the Gecko Bootloader running on the device. For more details on creating a GBL file, see section Creating a Signed and Encrypted GBL Upgrade Image File from an Application.
5. Upload the GBL file. For more details on the upgrade process, see section Bootloader Upgrade on Bootloaders with Communication Interface (Standalone Bootloaders).

Enabling Secure Boot RTSL on Series 2 Devices

Secure Boot RTSL (Root of Trust and Secure Loader) can be enabled using the following procedure:

1. Prepare a Gecko Bootloader image with secure boot enabled. The version of the Gecko Bootloader needs to be higher than the Gecko Bootloader on the device.
   - Turn on secure boot in Simplicity Studio by going to the **Bootloader Core** component and selecting the **Enable secure boot** option.
   - For EFR32xG21, in the **Bootloader Core** component, disable the **Allow use of public key from manufacturing token storage** option. This means that the Gecko Bootloader will never make use of the public key stored in the last page of the main flash

Copyright © 2025 Silicon Laboratories. All rights reserved.

    - (Optional) In the **Bootloader Core** component, select the **Require signed firmware upgrade files** option. This means that the Gecko Bootloader will only accept signed GBL files.
2. Generate a public/private Signing Key pair. See section Generating Keys for more information on creating a Signing Key pair.
3. Prepare an application that installs the public key generated from step 2 to the Secure Engine One-time Programmable memory. Installing a key in the VSE requires a reset routine. Make sure that the application does not end up in the reset loop. Create an unsigned GBL file from this application and upload it. For more information on installing public keys, see section Creating a Signed and Encrypted GBL Upgrade Image File from an Application.
4. Sign the Gecko Bootloader image generated from step 1 using the private key generated in step 2. See section Signing an Application Image for Secure Boot for more information on signing binaries.
5. Make a custom application that turns on secure boot on the Secure Engine and sign this application binary with the private key generated from step 2. For more details on how to turn on secure boot on the Secure Engine, see Series 2 Secure Boot with RTSL.
6. Create a GBL file using the Gecko Bootloader image from step 4.
7. Create a GBL file using the application from step 5. The GBL file need to be signed if the **Bootloader Core** component option **Require signed firmware upgrade files** was selected in step 1.
8. Upload the GBL file containing the Gecko Bootloader image.
9. Upload the GBL file containing the application.

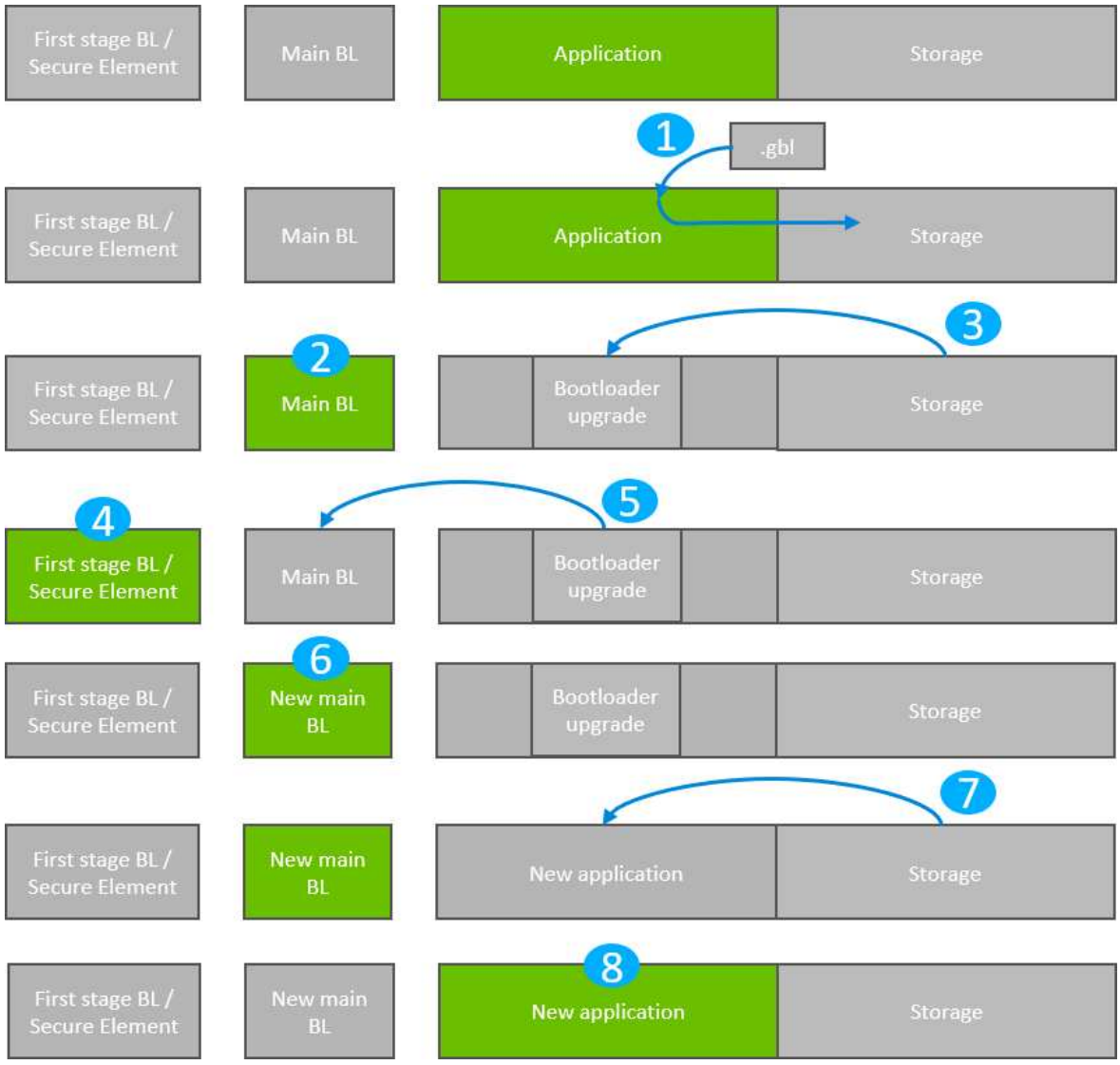
Downloading and Applying an Application GBL Upgrade File

Once the bootloader upgrade is completed, the existing application is rendered invalid, since the bootloader upgrade location overlaps with the application. A GBL upgrade file containing an application upgrade is transmitted to the bootloader. The application upgrade process follows that in section Standalone Bootloader Operation.

Bootloader Upgrade on Application Bootloaders with Storage

The process is illustrated in the following figure.

Copyright © 2025 Silicon Laboratories. All rights reserved.



1. A single GBL file containing both a bootloader upgrade image and an application image is downloaded onto the storage medium of the device (internal flash or external SPI flash).
2. The device reboots into the main bootloader.
3. The main bootloader parses its upgrade image into internal flash at the bootloader upgrade location. Depending on the application size as well as how the “scratch space” is configured for the bootloader on Series 2 devices, the existing application may be overwritten.
4. The main bootloader verifies the integrity of the upgrade image and then resets the device with reset reason BOOTLOADER_RESET_REASON_UPGRADE to apply the upgrade.
5. The device reboots into the first stage bootloader / Secure Engine.
6. The first stage bootloader / Secure Engine replaces the main bootloader with the new version.
7. The device boots into the new main bootloader.
8. The bootloader applies the application image from the GBL upgrade file.
9. The device boots into the application. Bootloader upgrade is complete.

A bootloader upgrade is started in the same way as an Application Upgrade. A single GBL file containing both a bootloader and an application upgrade is written to storage by the application, and the bootloader is entered.

The bootloader iterates over the list of storage slots marked for bootload and attempts to verify the GBL file stored within. Verification returns information about whether the GBL file contains an application, or both a bootloader and an application.

Copyright © 2025 Silicon Laboratories. All rights reserved.

The image parser parses the file. If the GBL file contains a bootloader, the bootloader upgrade data is returned in a callback. The bootloader core implements this callback and flashes the data to internal flash at the bootloader upgrade location.

The bootloader prevents a newly uploaded bootloader upgrade image from being interpreted as valid by holding back parts of the bootloader upgrade vector table until the GBL file CRC and GBL signature (if required) have been verified.

On Series 1 devices, the main bootloader signals the first stage bootloader that it should enter firmware upgrade mode by writing a command to the shared memory location at the bottom of SRAM, and then performing a software reset. On Series 2 devices, Secure Engine communication interface is used to signal the Secure Engine that a bootloader upgrade is ready to be performed.

On Series 1 devices, the first stage bootloader verifies the CRC of the bootloader upgrade present in the bootloader upgrade location in internal flash and copies the bootloader upgrade over the main bootloader if the version number of the upgrade is higher than the version number of the existing main bootloader. On Series 2 devices, the authenticity of the main bootloader optionally can be verified before applying the bootloader upgrade. See section Setting a Version Number for more information about versioning bootloader images.

The new main bootloader is entered, and the images in the list of storage slots marked for bootload are verified. When the image parser parses the slot containing the GBL file with the bootloader + application upgrade, the version number of the bootloader upgrade is equal to the running main bootloader version, so another bootloader upgrade will not be performed. Instead, the application upgrade data are returned in a callback. Bootloading of the new application proceeds as described in section Application Bootloader Operation.

Storage Space Size Configuration

The storage space size must be configured to have enough space to store the upgrade images. Depending on the configuration, the bootloader size can vary. For size requirements of the bootloader, see section Size Requirements for Different Bootloader Configurations for Series 1 Devices.

Upgrading Bootloaders without Secure Boot to Bootloaders with Secure Boot

A bootloader without the secure boot feature can be upgraded to a bootloader with the secure boot feature. The procedure is as follows:

1. Prepare a Gecko Bootloader image with secure boot enabled. The version of the bootloader needs to be higher than the bootloader on the device.
   - Turn on secure boot from the **Bootloader Core** component in Simplicity Studio by selecting the **Enable secure boot** option.
2. Generate a public/private Signing Key pair. See section Generating Keys for more information on creating a Signing Key pair.
3. Write the public key generated from the previous step to the device. The public key is stored as a manufacturing token in the device by default. This key can be written by application code running on the device as long as the Lock Bits page is configured to allow flash writes. If the Lock Bits page is locked, it can only be erased by the debugger. Therefore, signing/decryption keys residing in the Lock Bits page cannot be erased from firmware. This means that, for devices in the field, those areas in flash cannot be replaced with new ones. However, the Gecko Bootloader prepared from step 1 can be modified to look for the decryption and signature keys in a different location. Key locations are defined in the bootloader project file `btl_security_tokens.c` .
4. Prepare a signed application image using the private key generated in step 2. See section Signing an Application Image for Secure Boot for more information on signing an application.
5. Create a GBL file using the Gecko Bootloader image and the signed application image. The GBL file needs to be signed/unsigned depending on the configuration of the Gecko Bootloader running on the device. For more details on creating a GBL file, see section Creating a Signed and Encrypted GBL Upgrade Image File from an Application.
6. Upload the GBL file. For more details on the upgrade process, see section Bootloader Upgrade on Application Bootloaders with Storage.

Enabling Secure Boot RTSL on Series 2 Devices

Secure Boot RTSL can be enabled by using the following procedure:

1. Prepare a Gecko Bootloader image with secure boot enabled. The version of the Gecko Bootloader needs to be higher than the Gecko Bootloader on the device.
   - Turn on secure boot from the **Bootloader Core** component in Simplicity Studio by selecting the **Enable secure boot** option.

Copyright © 2025 Silicon Laboratories. All rights reserved.

- For EFR32xG21, in the **Bootloader Core** component, disable the **Allow use of public key from manufacturing token storage** option. This means that the Gecko Bootloader will never make use of the public key stored in the last page of the main flash.
- (Optional) In the **Bootloader Core** component, select the **Require signed firmware upgrade files** option. This means that the Gecko Bootloader will only accept signed GBL files.

2. Generate a public/private Signing Key pair. See section Generating Keys for more information on creating a Signing Key pair.
3. Prepare an application that installs the public key generated from step 2 to the Secure Engine One-time Programmable memory. Installing a key in VSE requires a reset routine. Make sure that the application does not end up in the reset loop. Create an un- signed GBL file from this application and upload it. For more information on installing public keys, see Series 2 Secure Boot with RTSL. For more details on creating a GBL file, see section Creating a Signed and Encrypted GBL Upgrade Image File from an Application.
4. Sign the Gecko Bootloader image generated from step 1 using the private key generated in step 2. See section Signing an Application Image for Secure Boot for more information on signing binaries.
5. Make a custom application that turns on secure boot on the Secure Engine and sign this application binary with the private key generated from step 2. For more details on how to turn on secure boot on the Secure Engine, see Series 2 Secure Boot with RTSL.
6. Create a GBL file using the Gecko Bootloader image from step 4 and the application from step 5. The GBL file must be signed if the **Bootloader Core** component option **Require signed firmware upgrade files** was selected in step 1. For more details on creating a GBL file, see section Creating a Signed and Encrypted GBL Upgrade Image File from an Application.
7. Upload the GBL file containing the Gecko Bootloader image and the application.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Gecko Bootloader Operation Secure Engine Upgrade

Gecko Bootloader Operation - Secure Engine Upgrade

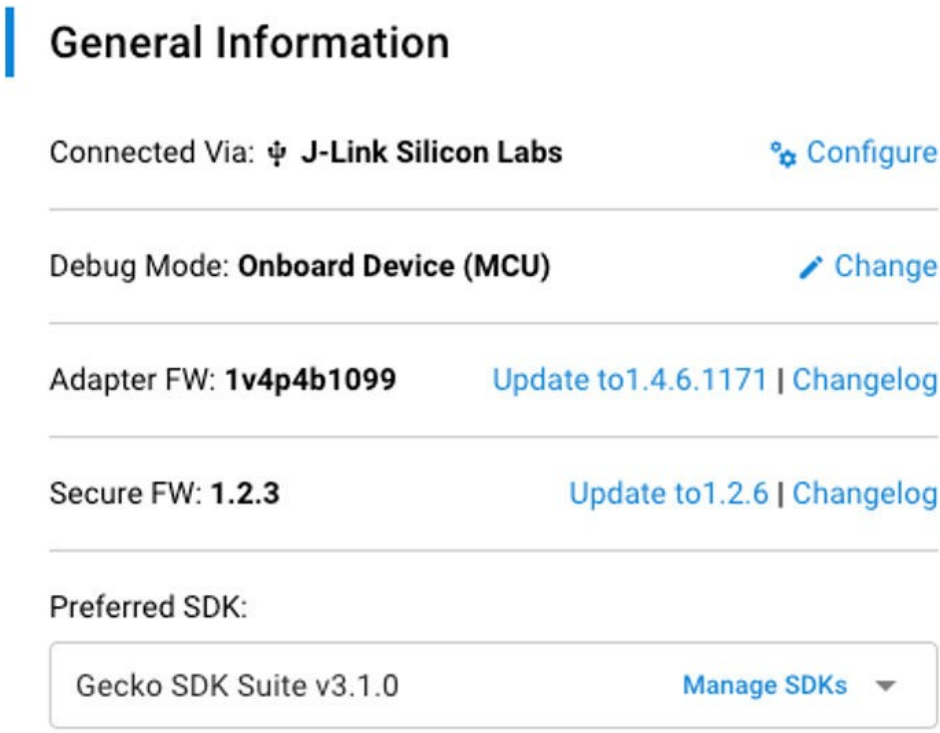
The Secure Engine is upgradable and requirements for upgrading the Secure Engine vary depending on the bootloader configuration:

- **Application bootloader with storage**: Upgrading the Secure Engine requires a single GBL file containing both Secure Engine and application upgrade images.
- **Standalone bootloader with communication interface**: Upgrading the Secure Engine requires two GBL files, one with only the Secure Engine upgrade image, and one with only the application upgrade image and optionally a third image containing only the Main bootloader upgrade.

A bootloader upgrade can also be included in the same GBL file in application mode, or as a third GBL file in standalone mode. The figures that illustrate Gecko Bootloader operation in this section do not provide information about the bootloader memory layouts for different devices. For more details refer to the section “Memory Space for Bootloading” in *UG103.6: Bootloader Fundamentals*.

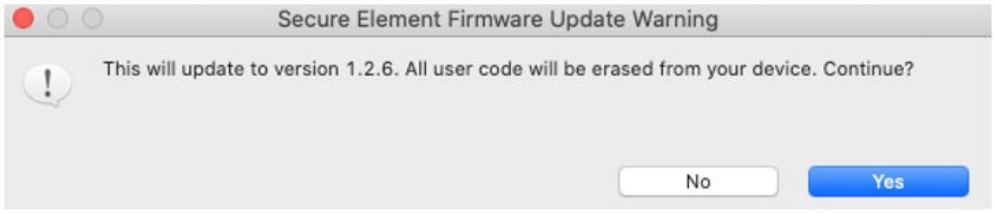
Signed and encrypted Secure Engine upgrade images are provided by Silicon Labs through Simplicity Studio. Upgrade images with the same or lower version number than the running Secure Engine will be ignored.

To download Secure Engine firmware images, connect a Series 2 device and select a preferred SDK. The Secure Firmware **Update to x.x.x** link appears in the Launcher Perspective, as shown in the following figure.



Click **Update to x.x.x** next to Secure FW: x.x.x. A warning dialog box appears. Click **[Yes]** to continue.

Copyright © 2025 Silicon Laboratories. All rights reserved.



The Launcher Perspective is then updated so that the current Secure Firmware version and link version are the same.

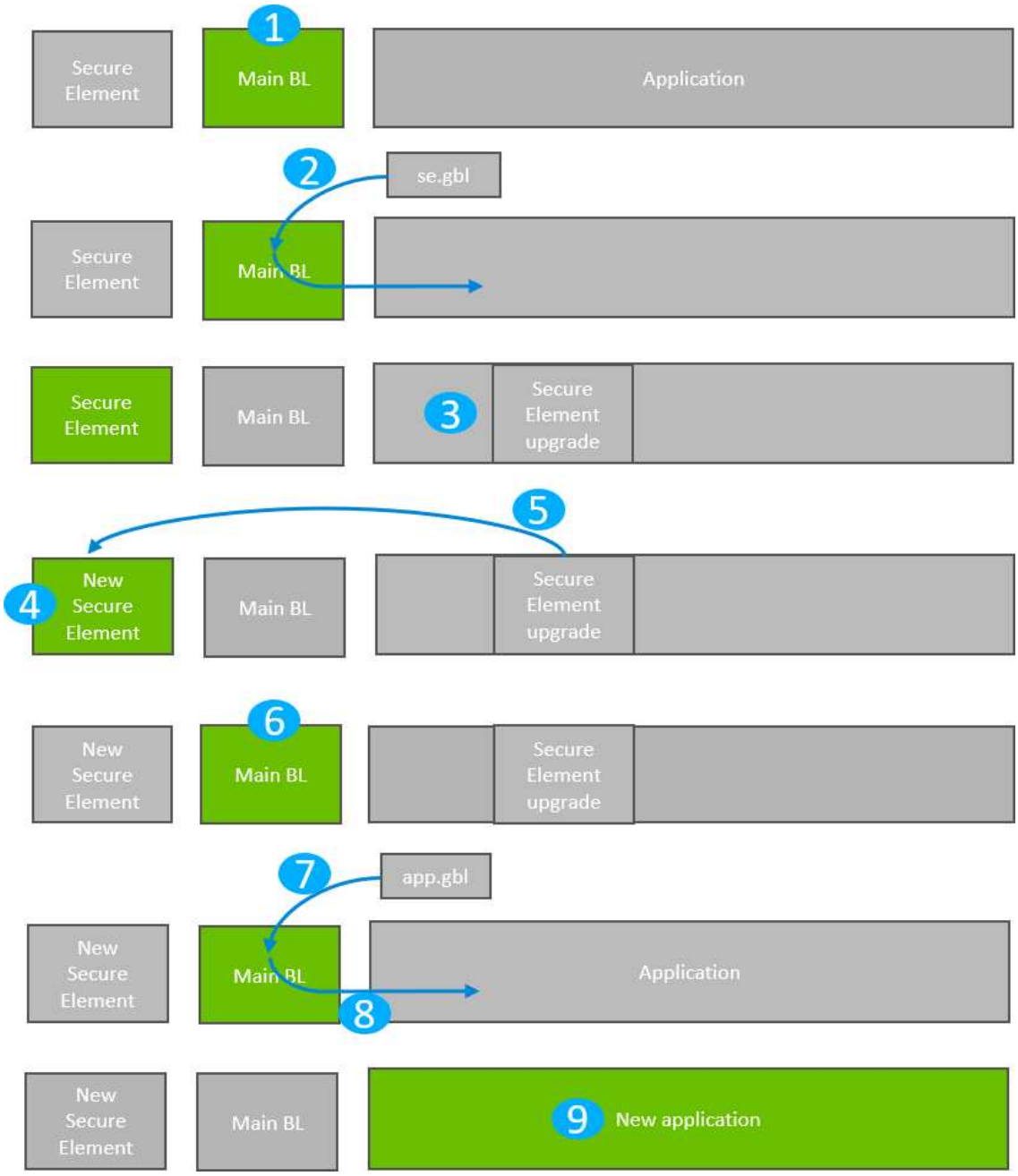


The Secure Engine firmware images can be found in the *util/se_release/public* directory under the Gecko SDK. Simplicity Studio displays the SE firmware version available in the Gecko SDK selected.

Secure Engine Upgrade on Bootloaders with Communication Interface (Standalone Bootloaders)

The process is illustrated in the following figure.

Copyright © 2025 Silicon Laboratories. All rights reserved.



1. The device reboots into the bootloader.
2. A GBL file containing only a Secure Engine upgrade image is transmitted from the host to the device.
3. The contents of the GBL Secure Engine tag are written to the pre-configured upgrade location in internal flash, overwriting the existing application.
4. The device reboots into the Secure Engine.
5. The Secure Engine is replaced by the new version found in the pre-configured upgrade location.
6. The device boots into the main bootloader.
7. A GBL file containing only an application image is transmitted from the host to the device.
8. The bootloader applies the application image from the GBL upgrade on the fly.
9. The device boots into the application. Secure Engine upgrade is complete.

Downloading and Applying a Secure Engine GBL Upgrade File

When the bootloader has entered the receive loop, a GBL upgrade file containing a Secure Engine upgrade is transmitted to the bootloader. When a packet is received, it is passed to the image parser. The image parser parses the data and

Copyright © 2025 Silicon Laboratories. All rights reserved.

returns Secure Engine upgrade data in a callback. The bootloader core implements this callback and flashes the data to internal flash at the pre-configured bootloader upgrade location.

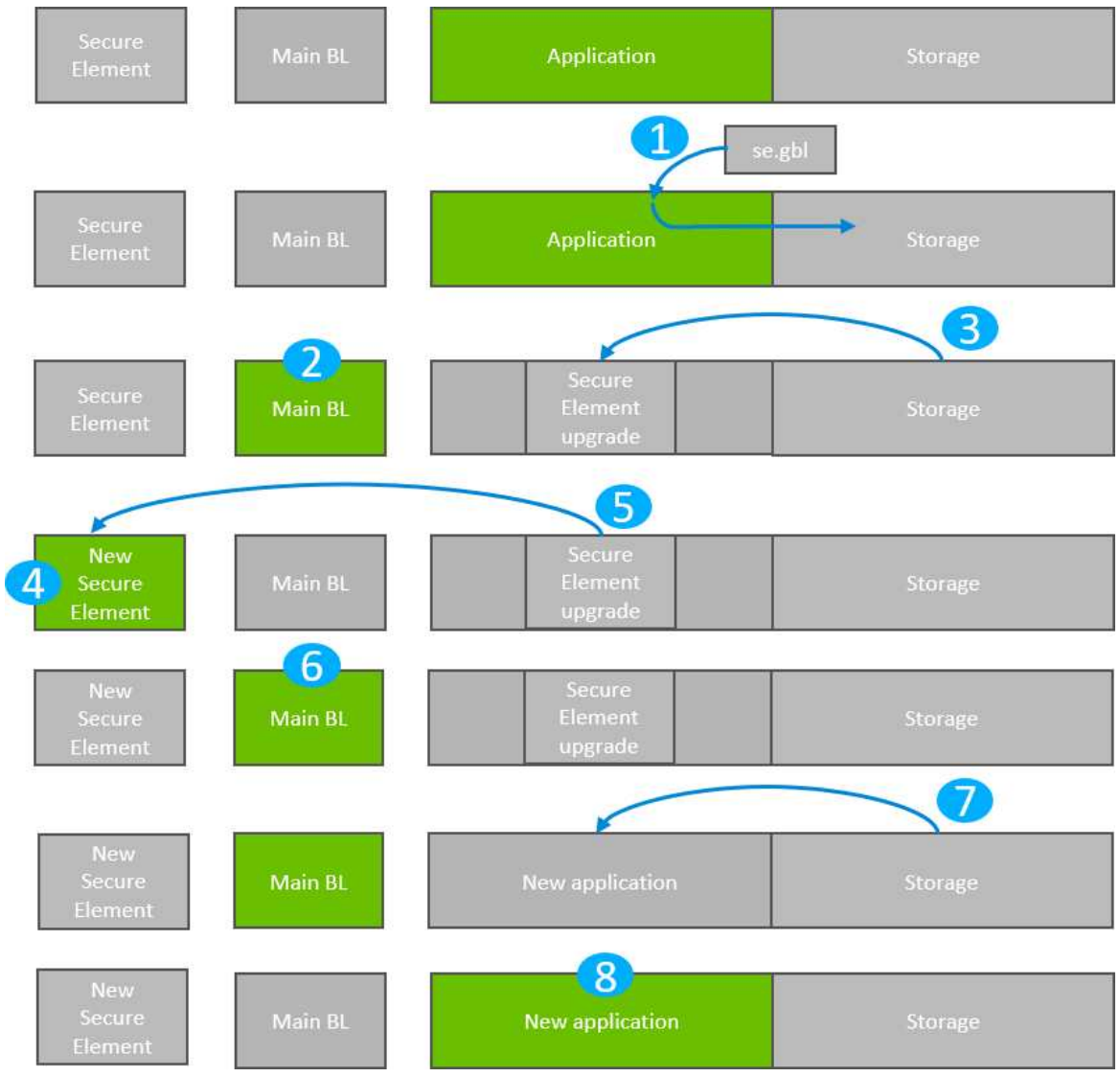
When a complete Secure Engine upgrade image is received, the main bootloader signals the Secure Engine that it should enter firmware upgrade mode. This is done by the Secure Engine communication interface that is used to signal that bootloader upgrade is ready to be performed.

Downloading and Applying an Application GBL Upgrade File

Once the Secure Engine upgrade is completed, the existing application is rendered invalid if the Secure Engine upgrade location overlaps with the application. A GBL upgrade file containing an application upgrade is transmitted to the bootloader. The application upgrade process follows that. For more information, see section Standalone Bootloader Operation.

Secure Engine Upgrade on Application Bootloaders with Storage

The process is illustrated in the following figure.



Copyright © 2025 Silicon Laboratories. All rights reserved.

1. A single GBL file containing both a Secure Engine upgrade image and an application image is downloaded onto the storage medium of the device (internal flash or external SPI flash).
2. The device reboots into the bootloader.
3. a) The main bootloader copies its upgrade image into internal flash at the pre-configured upgrade location.
   b) Alternatively, if the no-staging Secure Engine upgrade option has been enabled, the upgrade image will be fetched directly from the GBL file in storage instead of first copying the image to the pre-configured upgrade location.
4. The device reboots into the Secure Engine.
5. The Secure Engine is replaced by the new version found in the pre-configured upgrade location (or directly from storage, ref. 3b).
6. The device boots into the main bootloader.
7. The bootloader applies the application image from the GBL upgrade file.
8. The device boots into the application. Secure Engine upgrade is complete.

Storage Space Size Configuration

The storage space size must be configured to have enough space to store the upgrade images. The following table shows the reserved SE upgrade image sizes.

Device Family	Reserved Flash for SE Upgrade Image
EFR32xG21	48 kB
EFR32xG22	24 kB
EFR32xG23	96 kB

Depending on the configuration, the bootloader size can vary. For size requirements of the bootloader, see section Size Requirements for Different Bootloader Configurations for Series 1 Devices. The bootloader size for EFR32xG21 devices can be up to 16 kB and for EFR32xG22, EFR32xG23, and EFR32xG24 devices the bootloader size can be up to 24 kB. For more details, see *UG103.6: Bootloader Fundamentals*.

Copyright © 2025 Silicon Laboratories. All rights reserved.

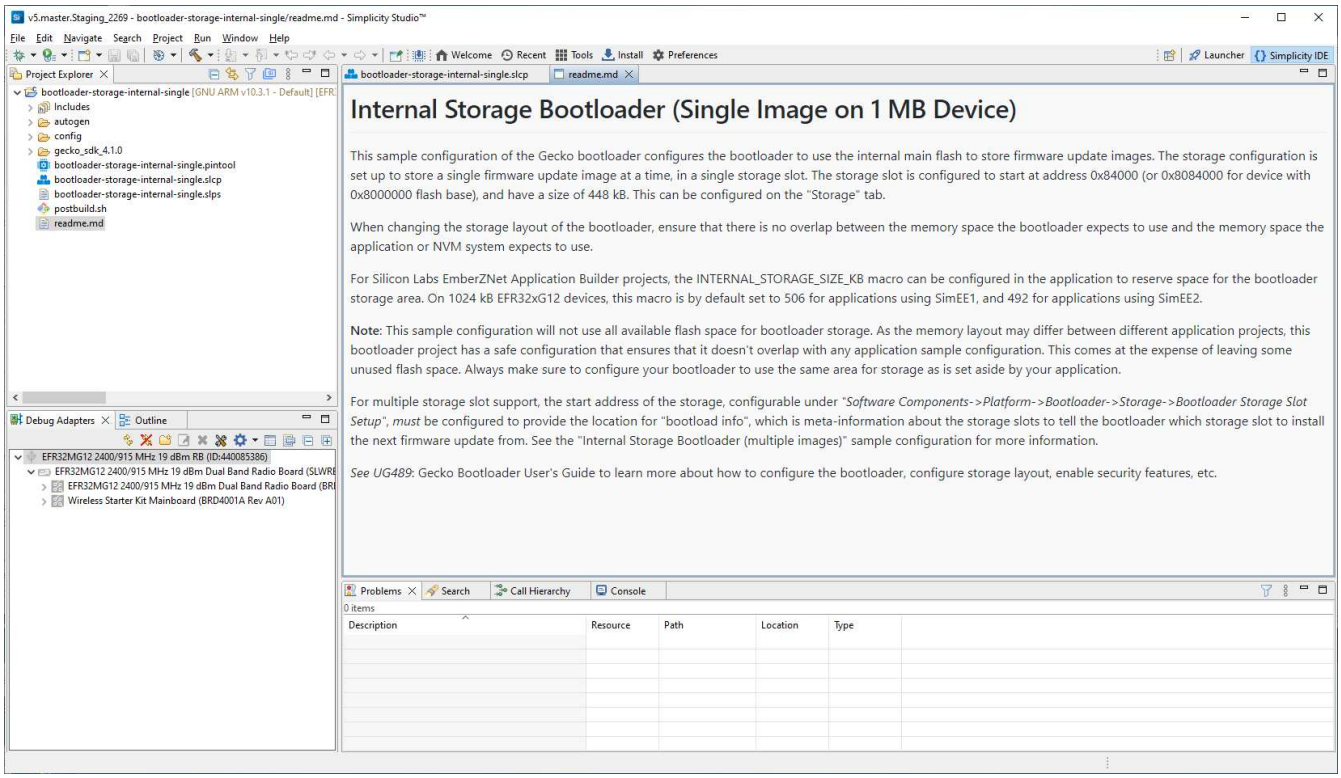
Getting Started with the Gecko Bootloader

Getting Started with the Gecko Bootloader

This section describes how to build a Gecko Bootloader from one of the provided examples. Simplicity Studio 5, used with Gecko SDK Suite (GSDK) v3.x and higher, differs from Simplicity Studio 4, used with GSDK v2.x, in how sample applications are selected and how projects are created. Refer to the documentation provided with your SDK for details. These instructions assume that you have installed Simplicity Studio 5, the GSDK and associated utilities as described in the SDK's quick start guide, and that you are familiar with generating, compiling, and flashing an example application in the relevant version.

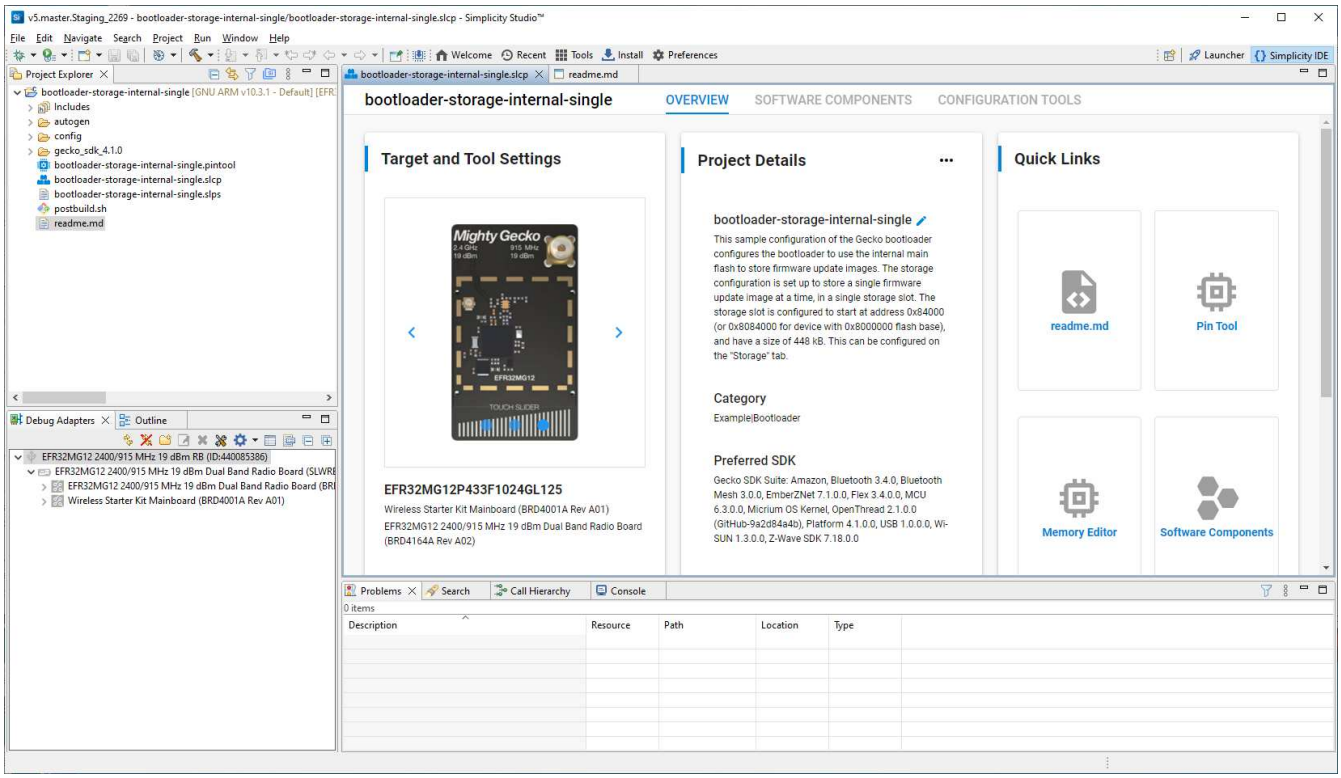
- *QSG169: Bluetooth® SDK v3.x Quick-Start Guide*
- Proprietary Flex SDK v3.x Quick-Start Guide
- *QSG170: Silicon Labs OpenThread SDK Quick-Start Guide*
- *QSG180: Zigbee EmberZNet Quick-Start Guide for SDK 7.0 and Higher*

1. Create a project based on the Gecko Bootloader example of your choice. The project opens with a tab describing the example.

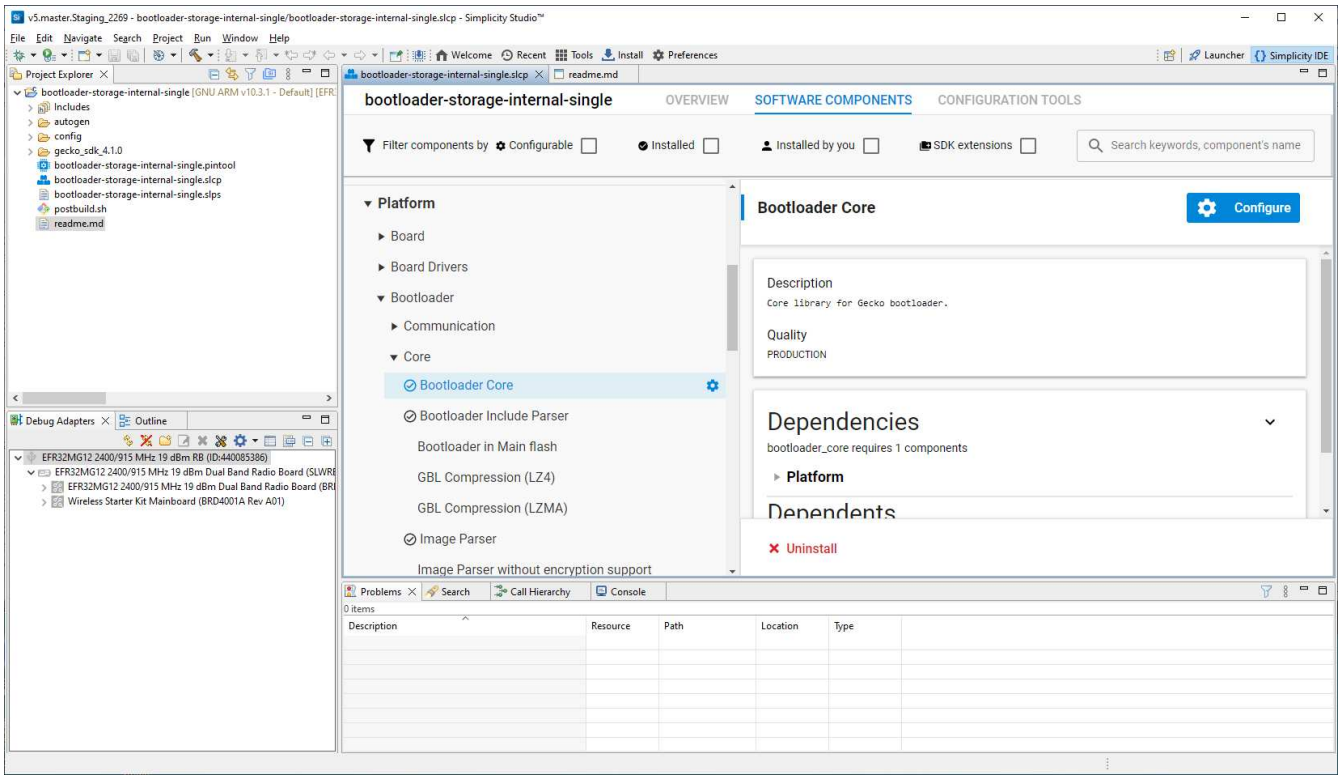


2. Click the project (*.slcp) tab to move to the Project Configurator interface.

Copyright © 2025 Silicon Laboratories. All rights reserved.

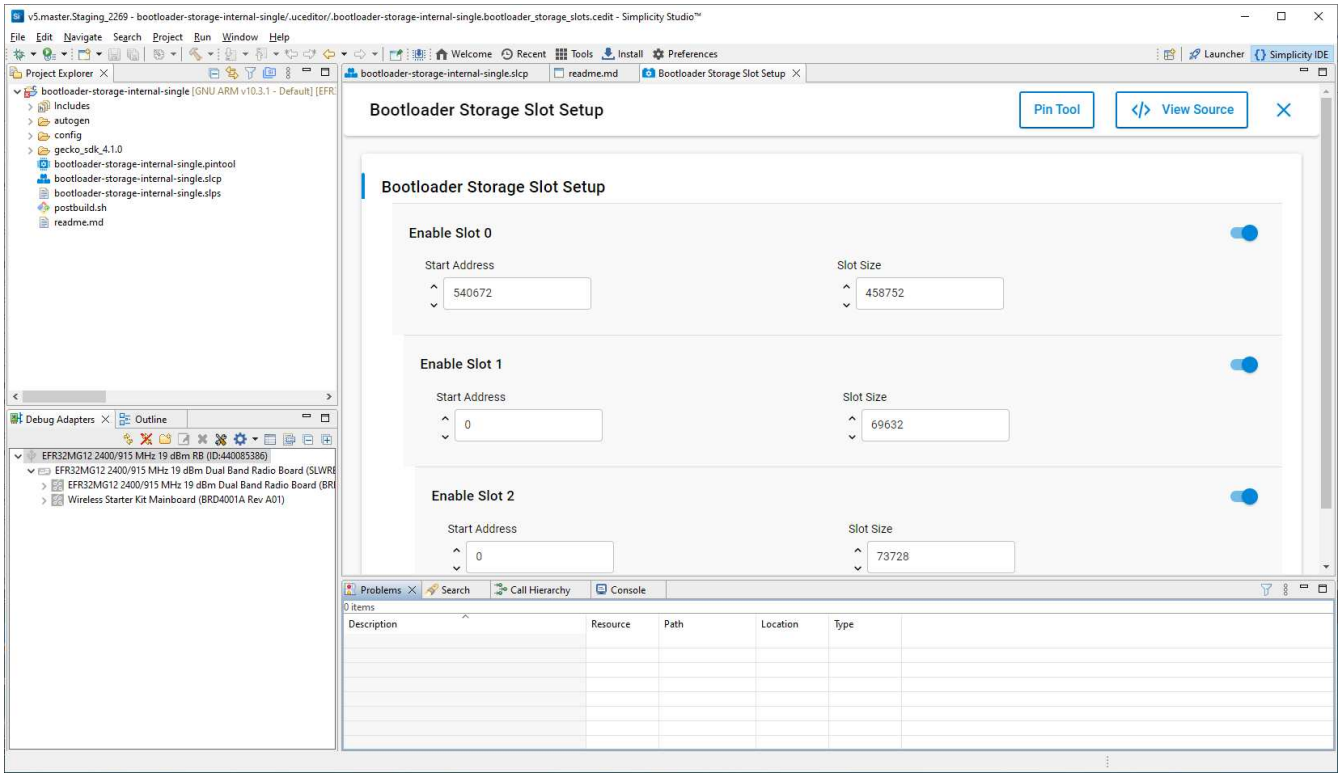


3. The Software Components tab shows the list of available components that can be installed in the project.

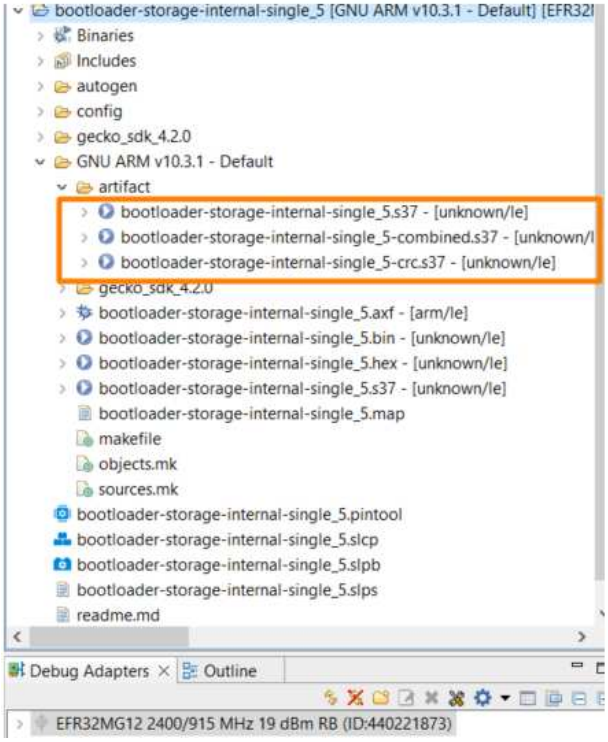


4. The **Storage Slot Setup** component allows you to configure storage slots to be used if a storage component is also installed. The default configuration matches the target part and bootloader type. This component supports a maximum of three storage slots.

Copyright © 2025 Silicon Laboratories. All rights reserved.



5. Click the **Build** (hammer) icon.
6. After the build is complete, the bootloader binaries are available in the **artifact** folder as depicted in the image below.



On Series 1 devices, three bootloader images are generated into the build directory: a main bootloader, a main bootloader with CRC32 checksum, and a combined first stage and main bootloader with CRC32 checksum. The main bootloader image is called **<project- name>.s37**, the main bootloader with CRC32 checksum is called **<projectname>-crc.s37**, while the

Copyright © 2025 Silicon Laboratories. All rights reserved.

combined first stage image + main bootloader image with a CRC32 checksum is called **<projectname>-combined.s37**. The first time a device is programmed, whether during development or manufacturing, the combined image needs to be programmed. For subsequent programming, when a first stage bootloader is already present on the device, it is okay to download an image containing just the main bootloader. In this case, the main bootloader with CRC32 should be used.

The requirement is that any main bootloader image that is programmed via serial wire must contain the CRC32 in the image. Files downloaded via serial wire are “s37” files. Most often, the **<projectname>-combined.s37** file is the one downloaded during production programming. However, it is possible to download only the main bootloader over serial wire, in which case **<projectname>-crc.s37** should be used.

Any main bootloader that is upgraded with the OTA or host method should already contain CRC32 because bootloader-initiated upgrades use GBL files (not “s37” files) and Simplicity Commander adds the CRC32 when it constructs the GBL file. The input files to Simplicity Commander can (and should) use the non-CRC “s37” file.

On Series 2 devices, the combined image is not present, since the first stage bootloader does not exist. The image containing only a main bootloader is the image that must be used to create a GBL file for bootloader upgrade.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Configuring the Gecko Bootloader

Configuring the Gecko Bootloader

Configuring Storage

Gecko Bootloaders configured as application bootloaders must include an API to store and access image files. This API is based on the concept of storage slots, where each slot has a predefined size and location in memory and can be used to store a single upgrade image. Slots are configured in the **Bootloader Storage Slot Setup** component.

When multiple storage slots are configured, a bootload list is used to indicate the order in which the bootloader should access slots to find upgrade images. If multiple storage slots are supported, the application should write the bootload list by calling bootloader_setImageToBootload before rebooting into the bootloader to initiate a firmware upgrade process. The bootloader attempts to verify the images in these storage slots in sequence and applies the first image to pass verification. If only a single storage slot is supported, the bootloader uses this slot implicitly. A maximum of three slots may be configured in the **Bootloader Storage Slot Setup** component.

SPI Flash Storage Configuration

When configuring a Gecko Bootloader to obtain images from SPI flash, modify the following.

The **base address of the storage area** should be configured in the **Common Storage** component. This is the address at which the bootloader places the bootload list, if more than one storage slot is configured. In the default configuration, this address is set to 0. If only a single storage slot is configured, the bootload list is not used, so configuring it may be omitted.

The **location and size of the storage slots** can be configured using the **Bootloader Storage Slot Setup** component (supports a maximum of three configurable storage slots). The addresses input here are absolute addresses (they are not offsets from the base address). If more than a single slot is configured, space must be reserved between the base address as configured in the **Common Storage** component and the first storage slot configured in the **Bootloader Storage Slot Setup** component. Enough space to fit two copies of the bootload list must be reserved. These two copies need to reside on different flash pages, to provide redundancy in case of power loss during writing. Two full flash pages therefore need to be reserved. In the default example application, a SPI flash part with 4 kB flash sectors is used. This means that 8 kB must be reserved before the first storage slot. The following figure illustrates how the storage area can be partitioned, where the numbers in the top row represent the starting addresses.

External storage with two storage slots

0	4 kB	8 kB	260 kB
Bootload list	Bootload list	Storage slot 0 Size = 252 kB	Storage Slot 1 Size = 252 kB

External storage with one storage slot

0
Storage slot 0 Size = 512 kB

Internal Storage Configuration

Copyright © 2025 Silicon Laboratories. All rights reserved.

When configuring a Gecko Bootloader to obtain images from internal flash, modify the following.

The **base address of the storage area** should be configured in the **Common Storage** component. This is the address at which the bootloader will place the prioritized list of storage slots to attempt to bootload from, if more than one storage slot is configured. In the default configuration, only a single storage slot is configured, so this value is set to 0, and isn't used. If more than one storage slot is configured, this value needs to be configured too.

The **location and size of the storage slots** can be configured using the **Bootloader Storage Slot Setup** Component (supports a maximum of three configurable storage slots). The addresses input here are absolute addresses (they are not offsets from the base address). If more than a single slot is configured, enough space must be reserved between the base address as configured in the **Common Storage** component and the first storage slot configured in the **Bootloader Storage Slot Setup** component. Enough space to fit two copies of the bootload list must be reserved. These two copies need to reside on different flash pages, to provide redundancy in case of power loss during writing. Two full flash pages therefore need to be reserved. The following figure illustrates how the storage area can be partitioned.

Internal storage with one storage slot

0	512 kB
Application	Storage Slot 0 Size = 512 kB

Internal storage with two storage slots

0	340 kB	342 kB	344 kB	648 kB
Application	Bootload list	Bootload list	Storage slot 0 Size = 340 kB	Storage slot 1 Size = 340 kB

> **Note**: The storage area partitioning in the example for two storage slots above does not take any NVM system into account. If using an NVM system like SimEE or PS Store, take care to place and size the storage area in such a way that bootloader storage does not overlap with NVM.

Compressed Upgrade Images

The Gecko Bootloader optionally supports compressed GBL files. In a compressed GBL file, only the application upgrade data is compressed, any metadata and bootloader upgrade data (if present) stay uncompressed. This means that a compressed GBL file is identical to a normal (uncompressed) GBL file, except that the GBL Programming Tag containing the application upgrade image (as described in *UG103.6: Bootloader Fundamentals*) has been replaced by a GBL LZ4 Compressed Programming Tag or GBL LZMA Compressed Programming Tag. Signature and encryption operations on a compressed GBL work identically to on an uncompressed GBL.

To be able to use compressed upgrade images, a decompressor for the relevant compression algorithm must be added to the Gecko bootloader. The following table shows which compression algorithms are supported by the Gecko Bootloader, and which Bootloader component should be added to enable the feature. The table also shows how much space the decompressor takes up in the bootloader, and how big of a size reduction to expect for the compressed application upgrade image. Be aware of the bootloader size requirement. The bootloader space might be too small to fit the decompressors, depending on the device and enabled components.

Compression Algorithm	Component	Bootloader Size Requirement	Application Upgrade Size Reduction (typical)
LZ4	GBL Compression (LZ4)	< 1 kB	~ 10%

Copyright © 2025 Silicon Laboratories. All rights reserved.

Compression Algorithm	Component	Bootloader Size Requirement	Application Upgrade Size Reduction (typical)
LZMA	Bootloader Compression (LZMA)	~5 kB flash, 18 kB RAM	~ 30%

It is important to note that the compressed GBL file stays compressed while being transferred to the device, and while it is stored in the upgrade area. It is decompressed by the bootloader when the upgrade is applied. This means that the running application in main flash will be identical to one that was installed using an uncompressed (normal) GBL file.

Compressed GBL files can only be decompressed by the bootloader when running standalone, not through the Application Interface. This means that upgrade image verification performed by the application prior to reboot will not attempt to decompress the application upgrade, it will only verify the signature of the compressed payload. After rebooting into the bootloader, it will decompress the image as part of the upgrade process.

> **Note**: The above means that Bluetooth in-place application upgrades cannot be compressed, as they are processed by the Bluetooth Supervisor or AppLoader using functionality in the bootloader through the Application Interface. Supervisor/stack and AppLoader updates can be compressed, but the user application cannot.

LZMA Compression Settings

LZMA decompression is only supported for images compressed with certain compression settings. Simplicity Commander automatically uses these settings when using the `commander gbl create --compress lzma` command.

- Probability model counters: lp + lc <= 2. Simplicity Commander uses lp=1, lc=1.
- Dictionary size no greater than 8 kB. Simplicity Commander uses 8 kB.

Together, these settings cause the decompressor to require 18 kB of RAM for decompression – 10 kB for the counters and 8 kB for the dictionary.

The GBL LZMA Compressed Programming Tag contains a full LZMA file, containing the LZMA header, raw stream, and end mark. The Gecko bootloader only supports decompressing payloads that contain the end mark as the last 8 bytes of the compressed stream.

Bootloader Example Configurations

The following sections describe the key configuration options for the example bootloader applications.

> **Note**: Security features are disabled for all example configurations. In development, Silicon Labs strongly recommends enabling security features to prevent unauthorized parties from uploading untrusted program code. See section Using Application Image Security Features to learn how to configure the security features of the Gecko Bootloader.

UART XMODEM Bootloader

Standalone bootloader for EFM32 and EFR32 devices running the EmberZNet PRO and Silicon Labs Connect protocol stacks, using XMODEM-CRC over UART.

In this configuration, the **UART XMODEM** communication component, **XMODEM Parser** component, and **Bootloader UART Driver** component are installed. For the example application to run on a custom board, the GPIO ports and pins used for UART need to be configured in the **Bootloader UART Driver** component. Here, Hardware Flow Control can be enabled or disabled, and the baud rate and pinout can be configured.

The **GPIO activation** component is also installed by default, allowing bootloader entry into firmware upgrade mode by activating a GPIO through reset. The GPIO pin used can be configured here. This component can be uninstalled if this

Copyright © 2025 Silicon Laboratories. All rights reserved.

functionality is not desired.

BGAPI UART DFU Bootloader

Standalone bootloader for the Bluetooth protocol stack, using the BGAPI protocol for UART DFU. This bootloader should be used for all NCP-mode Bluetooth applications.

In this configuration, the **BGAPI UART DFU** communication component and **Bootloader UART Driver** component are installed. For the example application to run on a custom board, the GPIO ports and pins used for UART need to be configured in the **Bootloader UART Driver** component. Here, Hardware Flow Control can be enabled or disabled, and the baud rate and pinout can be configured.

The GPIO activation component is also installed by default, allowing bootloader entry by activating a GPIO through reset. The GPIO pin used can be configured here. This component can be uninstalled if this functionality is not desired.

EZSP SPI Bootloader

Standalone bootloader for EmberZNet PRO and Silicon Labs Connect protocol stacks using EZSP for SPI.

In this configuration, the **EZSP-SPI** communication component, **XMODEM Parser** component, and **Bootloader SPI Peripheral Driver** component are installed. For the example application to run on a custom board, the GPIO ports and pins used for SPI and EZSP signaling need to be configured in the **Bootloader SPI Peripheral Driver** and **EZSP-SPI** components, respectively.

The **EZSP GPIO activation** component is also installed by default, allowing bootloader entry into firmware upgrade mode by activating a GPIO through reset. The GPIO pin used can be configured here. This component can be uninstalled if this functionality is not desired.

Bluetooth AppLoader OTA DFU Bootloader

Standalone bootloader for the Bluetooth protocol stack, using Bluetooth for Over-The-Air DFU.

In this configuration, the **Bluetooth AppLoader** communication component is installed.

SPI Flash Storage Bootloader

Application bootloader for all wireless protocol stacks, using an external SPI flash to store upgrade images received over the air by the application.

In this configuration, the **SPI Flash Storage** and **Common Storage** components, as well as the **Bootloader SPI Controller Driver** component, are installed. For the example application to run on a custom board, the GPIO ports and pins used for SPI communication with the external flash need to be configured in the **Bootloader SPI Controller Driver** component, and the type of SPI flash needs to be configured in the **SPI Flash Storage** component. The base address of the storage area can be configured in the **Common Storage** component. The location and size of the storage slots themselves can be configured using the **Bootloader Storage Slot Setup** component (supports up to three configurable storage slots).

The SPI Flash storage bootloader provides customizable callback functions, which can be found in *bootloader-callback-stubs.c*. The callback functions *storage_customInit* and *storage_customShutdown*, for example, can be used for custom hardware setups. Those callback functions can be interfaced from the applications using the bootloader interface functions such as *bootloader_init* and *bootloader_deinit* from *api/btl_interface.h*.

Currently, Silicon Labs supports the following SPI Flash storage bootloaders:

1. SPI Flash storage bootloader (multiple images)
2. SPI Flash storage bootloader (single image)
3. SPI Flash storage bootloader (single image with slot size of 1MB). **Note: This bootloader should only be used for systems with an external SPI Flash size >= 1 MB**.

SPI Flash Storage Bootloader SFDP

Application bootloader for all wireless protocol stacks, using an external SPI flash to store upgrade images received over the air by the application.

In this configuration, the **SPI Flash Storage SFDP** and **Common Storage** components, as well as the **Bootloader SPI Controller Driver** component, are installed. For the example application to run on a custom board, the GPIO ports and pins

Copyright © 2025 Silicon Laboratories. All rights reserved.

used for SPI communication with the external flash need to be configured in the **Bootloader SPI Controller Driver** component. The type of SPI flash is detected at runtime. The base address of the storage area can be configured in the **Common Storage** component. The location and size of the storage slots themselves can be configured using the **Bootloader Storage Slot Setup** component (supports up to three configurable storage slots).

The SPI Flash storage bootloader SFDP provides customizable callback functions, which can be found in *bootloader-callback-stubs.c*. The callback functions *storage_customInit* and *storage_customShutdown*, for example, can be used for custom hardware setups. Those callback functions can be interfaced from the applications using the bootloader interface functions such as *bootloader_init* and *bootloader_deinit* from *api/btl_interface.h*.

Internal Storage Bootloader

Application bootloader for all wireless protocol stacks, using internal flash to store upgrade images received over the air by the application.

Multiple examples are provided, including configurations for 512 kB flash memory devices like EFR32xG13, 1024 kB flash memory devices like EFR32xG12, and 2048 kB flash memory devices like EFM32GG11. **The storage layout should be modified before running the bootloader on any other devices**. In this configuration, the internal flash and common storage components are installed. The base address of the storage area is configured in the **Common Storage** component. The location and size of the storage slots can be configured using the **Bootloader Storage Slot Setup** component (provides up to three configurable storage slots). Default example applications are provided with configurations for both single storage slot and multiple storage slots.

The default storage slot configurations provided by the Gecko Bootloader **must** be configured to match the use-case-specific application configurations.

Table: Internal Storage Bootloader Default Storage Configurations

Sample Applications	Storage Offset	Storage Size	Additional Notes
Internal Storage Bootloader (single image on 256 kB device)	0x21800 (137216)	88064	This configuration is available for devices where flash base address is 0x00
'	0x8021800 (134354944)	88064	This configuration is only available for devices where the flash base address is 0x08000000
Internal Storage Bootloader (single image on 352 kB device)	0x28000 (163840)	147456	This configuration is available for devices where flash base address is 0x00
'	0x8028000 (134381568)	147456	This configuration is only available for devices where the flash base address is 0x08000000
Internal Storage Bootloader (single image on 512 kB device)	0x44000 (278528)	196608	This configuration is available for devices where flash base address is 0x00
'	0x8044000 (134496256)	196608	This configuration is only available for devices where the flash base address is 0x08000000
Internal Storage Bootloader (single image on 1 MB device)	0x84000 (540672)	458752	This configuration is available for devices where flash base address is 0x00
'	0x8084000 (134758400)	458752	This configuration is only available for devices where the flash base address is 0x08000000

Copyright © 2025 Silicon Laboratories. All rights reserved.

Sample Applications	Storage Offset	Storage Size	Additional Notes
Internal Storage Bootloader (single image on 2 MB device)	0x10000 (1048576)	1011712	This configuration is available for devices where flash base address is 0x00
'	0x8100000 (135266304)	1011712	This configuration is only available for devices where the flash base address is 0x08000000
Internal Storage Bootloader (multiple images on 1 MB device)	0x5A000 (368640)	337920	This configuration is available for devices where flash base address is 0x00
'	0xAC800 (706560)	337920	This configuration is available for devices where flash base address is 0x00
Internal Storage Bootloader (single image on 1536kB device)	0xC0000 (786432)	737280	This configuration is available for devices where flash base address is 0x00
'	0x80C0000 (135004160)	737280	This configuration is only available for devices where the flash base address is 0x08000000
Internal Storage Bootloader (single image on 1920kB device)	0x80E8000 (135168000)	966656	This example application is available for device families with flash size of 1920kB or higher (for example xG25 device family)

Image Acquisition Application Example Configuration

These examples illustrate applications that acquire and store a GBL upload image for an application bootloader. For the running bootloader to accept an application upgrade, the new application version must be higher than the existing version. The version number can be set in Appbuilder by defining the macro “APP_PROPERTIES_VERSION” in the **Additional Macros** field on the **Other** tab.

Setting a Version Number

To distinguish between different builds of the Gecko Bootloader, it is useful to set a version number. To perform a bootloader upgrade, not only must the running bootloader pass its integrity checks (see section Downloading and Applying a Bootloader GBL Upgrade File), but the bootloader upgrade image must also have a higher version number than the running bootloader image. A version number can be set using Simplicity Studio by configuring the **Bootloader Version Main Customer** option of the **Bootloader Core** component. This macro will be picked up by the config file **btl_config.h**, where it is combined with the version number of the Gecko Bootloader files provided by Silicon Labs.

Placing Bootloader in Main Flash

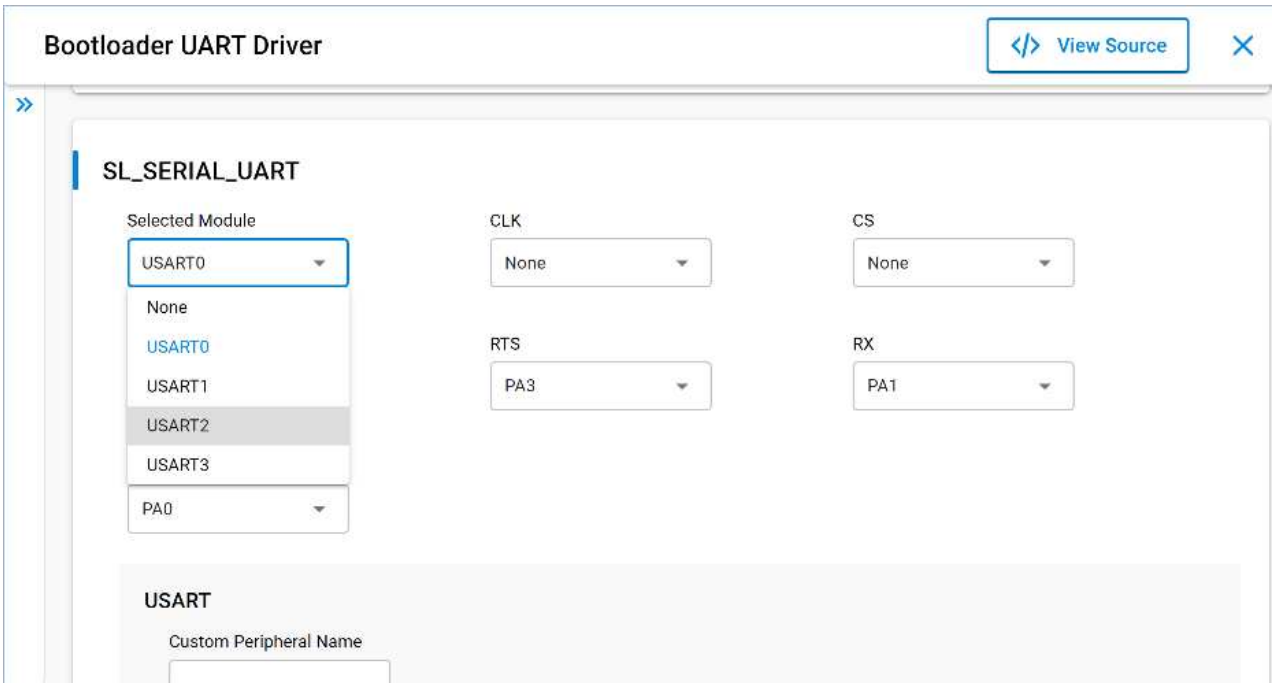
For device families xG13 and xG14, the entire main stage bootloader might not fit into the bootloader flash if the user installs some extra components. In such scenarios, the main stage bootloader can be placed in the main flash by installing the Bootloader in Main Flash core component. This component internally provides a MACRO named MAIN_BOOTLOADER_IN_MAIN_FLASH which is used to set the appropriate address required to place the main bootloader in main flash. The MACRO is used to generate the linker file from the common linker template with the appropriate addresses to place the main stage bootloader in the main flash.

Hardware Configuration

The Gecko Bootloader uses the Pin Tool for configuration of pinout and other hardware-related settings. When Pin Tool configuration is available for a bootloader component, the relevant settings are shown in the Component Editor for that

Copyright © 2025 Silicon Laboratories. All rights reserved.

component.



The standalone Pin Tool User Interface can also be used to configure settings for Gecko Bootloader if desired.

While the Pin Tool provides configuration for many different peripherals, the Gecko Bootloader uses only the following Pin Tool modules:

- SERIAL is used by the UART Driver component to configure baud rate, flow control mode and pinout.
- VCOM is used by the UART Driver component to enable the serial interface if necessary (only on Silicon Labs Wireless Starter Kits).
- EXTFLASH is used by the SPI Driver to configure frequency and pinout.
- SPINCP is used by the SPI Slave Driver to configure pinout.
- BTL_BUTTON is used by the GPIO Activation component.
- CMU HFXO frequency setting is used by the delay driver to calibrate timing if the core is running from the HFXO.

Other settings, like CMU oscillator configuration or DCDC configuration, are not taken into consideration by the default bootloader code. If using these configuration settings is desired, the required code must be added in btl_main.c.

> **Note**: While the delay driver uses the HFXO frequency setting from Pin Tool, the HFXO enable setting is not used to initialize the HFXO on startup. This setting is only used when calling the bootloader through the Application Interface (see section Gecko Bootloader and TrustZone), and the application has switched to the HFXO prior to calling the Bootloader Application Interface API.

Size Requirements for Different Bootloader Configurations for Series 1 Devices

Enabling different configuration options for the Gecko Bootloader changes the size of the resulting image. The following table shows a list of example bootloader configurations and the resulting approximate size of the main bootloader. Note that any size above 14 kB will be too large to fit in the bootloader area of flash of EFR32xG13, and that any size above 16 kB will be too large to fit in the bootloader area of flash on EFR32xG14 and EFM32TG11. This table was last updated for Gecko SDK Suite 4.0 (Gecko Bootloader 2.0), and no guarantees are made that a configuration of a specific size with one SDK version will maintain that size in future releases.

Table: Bootloader Size Requirements

Copyright © 2025 Silicon Laboratories. All rights reserved.

Base Configuration	Enabled Options	Size (kB)
XMODEM UART	Default configuration	12.8
'	Secure boot, signed and encrypted upgrade	12.8
'	Secure boot, signed and encrypted upgrade, LZ4 compression	13.7
'	Secure boot, signed and encrypted upgrade, LZMA compression	17.9
Internal storage, single slot	Default configuration	11.5
'	Secure boot, signed and encrypted upgrade	11.5
'	Secure boot, signed and encrypted upgrade, LZ4 compression	12.4
'	Secure boot, signed and encrypted upgrade, LZMA compression	16.6
Internal storage, multiple slots	Default configuration	12.1
'	Secure boot, signed and encrypted upgrade	12.1
'	Secure boot, signed and encrypted upgrade, LZ4 compression	13
'	Secure boot, signed and encrypted upgrade, LZMA compression	17.1
SPI flash, single slot	Default configuration	12.6
'	Secure boot, signed and encrypted upgrade	12.6
'	Secure boot, signed and encrypted upgrade, LZ4 compression	13.5
'	Secure boot, signed and encrypted upgrade, LZMA compression	17.7
SPI flash, multiple slots	Default configuration	13.1
'	Secure boot, signed and encrypted upgrade	13.1
'	Secure boot, signed and encrypted upgrade, LZ4 compression	14
'	Secure boot, signed and encrypted upgrade, LZMA compression	18
SPI flash using SFDP, single slot	Default configuration	12.8
'	Secure boot, signed and encrypted upgrade	12.8
'	Secure boot, signed and encrypted upgrade, LZ4 compression	13.8
'	Secure boot, signed and encrypted upgrade, LZMA compression	18
BGAPI UART	Default configuration	11.8
'	Secure boot, signed and encrypted upgrade	11.8
'	Secure boot, signed and encrypted upgrade, LZ4 compression	12.7
'	Secure boot, signed and encrypted upgrade, LZMA compression	16.9
EZSP SPI	Default configuration	13.1
'	Secure boot, signed and encrypted upgrade	13.1
'	Secure boot, signed and encrypted upgrade, LZ4 compression	13.9

Copyright © 2025 Silicon Laboratories. All rights reserved.

Base Configuration	Enabled Options	Size (kB)
'	Secure boot, signed and encrypted upgrade, LZMA compression	18.1

Copyright © 2025 Silicon Laboratories. All rights reserved.

Simplicity Commander and the Gecko Bootloader

Simplicity Commander and the Gecko Bootloader

Simplicity Commander is a single, all-purpose tool to be used in a production environment. It is invoked using a simple CLI (Command Line Interface that is also scriptable. You can use Simplicity Commander to perform these essential tasks:

- Generating key files for signing and encryption
- Signing application images for Secure Boot
- Creating GBL images (encrypted or unencrypted, signed, or unsigned)
- Parsing GBL images

Simplicity Commander is used throughout the examples in the following sections. For more information on executing the commands to complete these tasks, see *UG162: Simplicity Commander Reference Guide*.

> **Note**: Simplicity Commander also offers a GUI (Graphical User Interface) that can be used in the lab for typical tasks such as flashing device images. The functions described in this User Guide are performed from the CLI.

Creating GBL Files Using Simplicity Commander

To create an unsigned GBL file from an application **myapp.s37**, execute `commander gbl create myapp.gbl --app myapp.s37`.

To create an unsigned GBL file from a main bootloader upgrade **mybootloader.s37**, execute `commander gbl create mybootloader.gbl --bootloader mybootloader.s37`. This file can be used with the standalone bootloader configurations of the Gecko Bootloader.

To create an unsigned GBL file from a Secure Engine, upgrade **mySecureElement.seu**, and execute `commander gbl create mySecureElement.gbl --seupgrade mySecureElement.seu`. The Secure Engine images, .seu, are provided by Silicon Labs and can be found through Simplicity Studio. See section Gecko Bootloader Operation - Secure Engine Upgrade.

These commands can also be combined to create a single upgrade image, suitable for use with application bootloader configurations of the Gecko Bootloader: `commander gbl create myupgrade.gbl --app myapp.s37 --bootloader mybootloader.s37 --seupgrade mySecureElement.seu`.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Gecko Bootloader Security Features

Gecko Bootloader Security Features

About Bootloader Image Security

Secure Boot and Secure Firmware Upgrade, discussed in the following sections, enable Gecko Bootloader to provide authenticity and integrity checks on the Application image, which provides a sufficient level of security for many applications. However, in systems without a hardware root of trust, no process checks the authenticity or integrity of the Gecko Bootloader itself. Its security is provided solely by the device hardware and the robustness of the software running on the device.

The native behavior of Firmware Upgrade will prevent accidental version rollback of Gecko Bootloader under normal usage conditions. However, without a hardware root of trust, intentional downgrade attacks may be feasible. If a higher level of security assurance is required by the application, using a Series 2 device with hardware root of trust and anti-rollback protection is recommended. For more information, refer to Series 2 Secure Boot with RTSL.

About Application Image Security

The Gecko Bootloader can enforce security on two levels:

- Secure Boot refers to the verification of the authenticity of the application image in main flash on every boot of the device.
- Secure Firmware Upgrade refers to the verification of the authenticity of an upgrade image before performing a bootload, and optionally enforcing that upgrade images are encrypted.

Secure Boot Procedure

When Secure Boot is enabled, the cryptographic signature of the application image in flash is verified on every boot before the application is allowed to run. Secure Boot is not enabled by default in the example configurations provided by Silicon Labs, but enabling it is highly recommended to ensure the validity and integrity of firmware images.

Signature Algorithms

The Gecko Bootloader supports the ECDSA-P256-SHA256 cryptographic signature algorithm. This is the ECDSA (elliptical curve digital signature algorithm) of the SHA-256 digest of the application firmware image, using the NIST P-256 (secp256r1) curve.

Summary of Operation

1. On boot, the bootloader checks the application image for information about whether it is signed.
2. The type of signature and signature location is determined.
3. If the type of signature does not match the requirements of the bootloader, the bootloader enters device firmware upgrade mode and prevents the application from running.
4. According to the chosen signature algorithm, the signature of the contents of flash from the beginning of the application to the location of the signature is compared to the signature at the signature location.
5. If the signatures do not match, the bootloader enters device firmware upgrade mode and prevents the application from running.

Secure Boot using ECDSA-P256-SHA256

For an image to be signed for Secure Boot, the application needs to contain a copy of the **ApplicationProperties_t** struct. This struct contains information about which signature algorithm is used, and where to find the signature.

On every boot, the bootloader calculates the SHA-256 digest of the application image, from the beginning of the application to the start of the signature. The signature of the SHA-256 digest is then verified using ECDSA-P256.

Copyright © 2025 Silicon Laboratories. All rights reserved.

If the signature is valid, the application is allowed to boot. Else, the bootloader is entered, and an application upgrade is attempted if one is available.

The public key used for signature verification is stored as a manufacturing token on Series 1 and EFR32xG22 devices. OnEFR32xG21 devices, the public key can either be stored as a manufacturing token or stored in the Secure Engine One-time Programmable memory (OTP). Simplicity Commander can be used to generate a key pair and write the public key to the device. See *UG162: Simplicity Commander Reference Guide* for more information.

Secure Boot with Application Rollback Protection

On every boot, the application version in the **ApplicationData_t** struct is stored at the end of the bootloader area in flash, which is used to prevent applications from being downgraded. The application version can remain the same for upgrades. The bootloader will only allow applications to increment its version 6 times by default. The bootloader can be upgraded after that to reset the stored application versions and to allow application upgrades again. The allowed number of upgrades can be increased by modifying linkerfiles to reserve an extra 4 bytes for each version and by increasing SL_GBL_APPLICATION_VERSION_STORAGE_CAPACITY from *btl_bootload.c*. **For Series 2 devices, this option is not applicable on devices with Secure Engine configured to perform full page lock**. If the bootloader area in flash is locked by Secure Engine, the bootloader will not be able to store the application versions in flash, and it would continue without performing that function.

When the application upgrade version check option is enabled in the **Bootloader Core** component, the bootloader checks where App properties points into flash. If app properties struct is present, then it will check compatibility of the application properties struct and check if the upgrade version is greater than the installed app; otherwise, it will check if product ID is correct before applying the upgrade image.

If app properties struct is absent at the app properties location, it is assumed that the previous upgrade was interrupted, and allows the upgrade to continue.

If application rollback prevention component is installed in the bootloader project, then before applying the upgrade image, the bootloader will check the application version in the application properties structure that resides inside the signed/encrypted GBL file and will only apply the OTA image to application area if the application version in ApplicationData structure is equal or higher than the highest application version last seen.

The application rollback prevention feature can be enabled in the **Bootloader Core** component by selecting the **Enable application rollback protection** option. The **Minimum application version allowed** option can be used to configure the minimum application version that should be allowed to boot.

The application versions are stored in the bootloader area in flash. To properly protect the stored application versions, it is recommended to lock the bootloader flash by selecting the **Prevent bootloader write/erase** option in the **Bootloader Core** component.

Secure Boot Using a Certificate

On Series 2 devices, a certificate-based secure boot operation is supported. The Certificate contains:

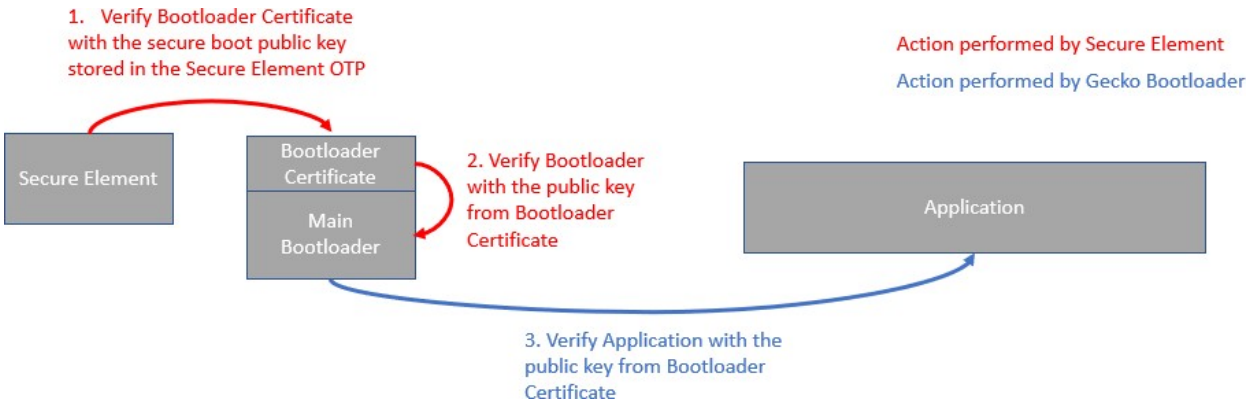
- Struct version: The version of the certificate structure.
- Public key: ECDSA-P256 public key, X and Y coordinates concatenated, used to validate the image.
- Certificate version: The version of the running certificate.
- Signature: ECDSA-P256 signature, used for the authentication of the public key and the certificate version.

The definition of the certificate struct can be found in api/application_properties.h.

To utilize certificate-based secure boot, configure Secure Engine to authenticate the bootloader image by configuring the certificate- based secure boot option in the Secure Engine OTP. Configure the Gecko Bootloader to enable certificate-based secure boot in the **Bootloader Core** component by selecting the **Enable certificate support** option. The Gecko Bootloader certificate must be signed by the private key pair of the public key stored in the Secure Engine OTP. For more information on the key storage, see section Key Storage.

The certificate-based secure boot procedure is illustrated in the following figure.

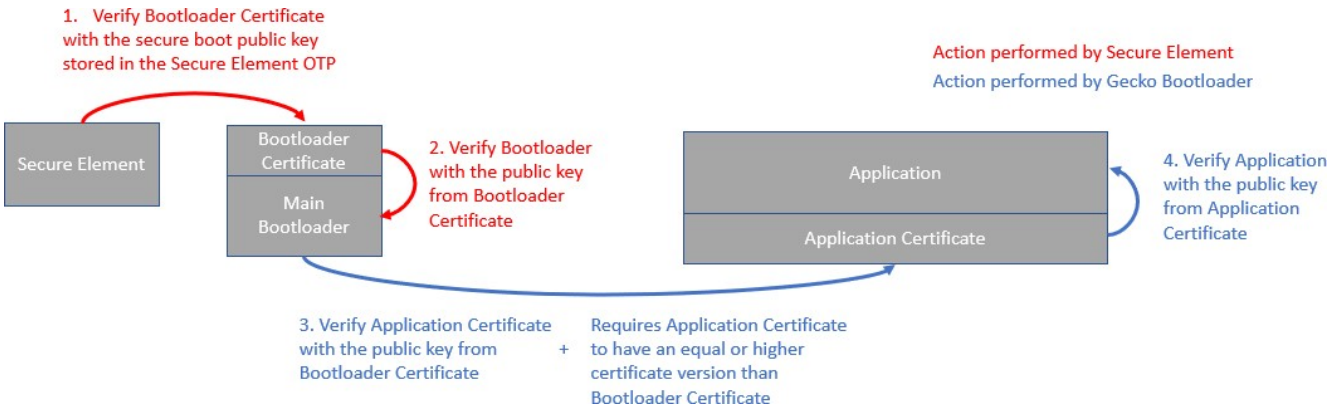
Copyright © 2025 Silicon Laboratories. All rights reserved.



Once the certificate-based secure boot option on Secure Engine is turned on, Secure Engine verifies the Gecko Bootloader certificate. The public key stored in the certificate is used to validate the signature of the Gecko Bootloader. Secure Engine will not accept bootloader images without a certificate.

If only the secure boot option is enabled (not certificate-based) on Secure Engine, and Secure Engine identifies a certificate, the certificate will be used to validate the bootloader image. If the certificate version from the bootloader image is higher than 0 and it gets verified once, Secure Engine will never again accept direct signed bootloader images without a certificate.

The Gecko Bootloader will authenticate the direct signed application using the public key stored in the Gecko Bootloader certificate. If the application contains a certificate, Gecko Bootloader will authenticate it. The procedure is illustrated in the following figure.



After authentication of the application certificate, Gecko Bootloader verifies the signature of the application using the public key from the application certificate. In addition, Gecko Bootloader compares the Gecko Bootloader certificate version against the application certificate version. All application images with a certificate version lower than the certificate version of the Gecko Bootloader will be rejected. Gecko Bootloader can be configured to only allow applications with certificate to boot by configuring the **Bootloader Core** component by selecting the **Reject direct signed images** option.

The **ApplicationProperties_t** struct contains the certificate struct **ApplicationCertificate_t**. The certificate struct can be injected to images that contain an **ApplicationProperties_t** with **ApplicationCertificate_t**. To inject a certificate to an image, issue the following command from Simplicity Commander:

```
commander convert <image file> --secureboot --keyfile <signing key> --certificate <certificate>--outfile <signed image file with certificate>
```

Secure Firmware Upgrade

The Gecko Bootloader supports a secure firmware upgrade process. This is achieved by using symmetric encryption to encrypt the upgrade image, and asymmetric cryptography to sign the upgrade image. Symmetric encryption provides

Copyright © 2025 Silicon Laboratories. All rights reserved.

confidentiality and asymmetric cryptography provides integrity and authenticity. Note that encryption alone is not enough to provide authenticity.

Encryption Algorithms

The Gecko Bootloader supports the AES-CTR-128 encryption algorithm. The GBL upgrade file is encrypted using 128-bit AES in Counter mode with a random nonce as the initial counter value. Through configuration, the GBL file's decryption duration can be improved by increasing the counter's block size. The block size is configurable with options 1, 2, 4, or 8 blocks. The configuration is available as part of the AES CTR Stream Block Config component.

The AES-128 key used for decryption, more commonly called the OTA decryption key, is stored as a manufacturing token on Series 1 and EFR32xG22 devices. On EFR32xG21 devices, the OTA decryption key can either be stored as a manufacturing token or stored in the Secure Engine OTP. To make use of the OTA decryption key stored in the Secure Engine OTP, the **Use symmetric key stored in Secure Engine storage** option in the **Bootloader Core** component must be selected. Simplicity Commander can be used to generate an OTA decryption key and write the key to the device. See *UG162: Simplicity Commander Reference Guide* for more information. For more information on storing the OTA decryption key on Series 2 devices, see *AN1222: Production Programming of Series 2 Devices*.

The Secure Engine OTP key support depends on the SE Manager component, which is enabled by default.

Signature Algorithms

The Gecko Bootloader supports the ECDSA-P256-SHA256 cryptographic signature algorithm. This is the ECDSA signature of the SHA-256 digest of the GBL upgrade file, using the NIST P-256 (secp256r1) curve.

Summary of Operation

Before starting a firmware upgrade process, the application can verify an image in storage by calling into the bootloader verification functions. For bootloaders with Communication Interface, the host device should verify the image before sending it to the NCP or RCP.

During firmware upgrade, the GBL file is parsed, and if encrypted, decrypted on-the-fly. A GBL Signature Tag in the GBL file indicates to the bootloader that the file is signed, and the signature is verified. If signature verification fails, the firmware upgrade process is aborted.

On Series 2 devices, Gecko Bootloader will authenticate the GBL signature tag using the public key stored in the bootloader certificate if the **Enable certificate support** option is selected in the **Bootloader Core** component. A GBL Certificate tag in the GBL file indicates to the bootloader that the GBL certificate tag needs to be authenticated using the public key stored in the bootloader certificate. The certificate version in the GBL certificate tag is compared with the bootloader certificate and only a version equal or higher than the bootloader certificate is accepted. Once the GBL certificate tag is authenticated, the GBL file's signature is verified using the authenticated public key from the GBL certificate tag. On EFRxG21 devices, the GBL certificate tag can also be signed by the private key pair of the public key stored in the Secure Engine OTP.

Using Application Image Security Features

This example assumes that a bootloader called **bootloader-uart-xmodem** has been built in Simplicity Studio. For a Series 1 device, three files of interest have been generated in the output directory:

- **bootloader-uart-xmodem.s37** – This file contains the main bootloader. Can be used for bootloader upgrade.
- **bootloader-uart-xmodem-crc.s37** - This file contains the main bootloader with CRC32 checksum.
- **bootloader-uart-xmodem-combined.s37** – This file contains the first stage and main bootloader with a CRC32 checksum in a single image. Can be used for manufacturing and initial deployment of the bootloader.

The relevant version can be flashed to the EFR32 using the Flash Programmer in Simplicity Studio or using Simplicity Commander. For Series 2 devices, only the main bootloader binary is generated.

This example provides two ways of signing the upgrade images. The first option uses Simplicity Commander to generate key material and sign data. This is suitable for development. The second option uses an external signer, such as a dedicated Hardware Security Module (HSM) to protect private key material and perform signing operations. Silicon Labs recommends using an HSM to safeguard private keys.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Generating Keys

To use the security features of the Gecko Bootloader, encryption and signing keys need to be generated. These keys must then be written to the EFR32 device. The encryption key is used with the GBL file for secure firmware upgrade. The signing keys are used both with the GBL file for secure firmware upgrade and to sign the application image for Secure Boot.

Generating a Signing Key Using Simplicity Commander

```
commander util genkey --type ecc-p256 --privkey signing-key --pubkey signing-key.pub --tokenfile signing-key- tokens.txt
```

This creates an ECDSA-P256 key pair for signing; **signing-key** contains the private key in PEM format and **must be kept secret from third parties**. This key will later be used to sign images and GBL files. **signing-key.pub** contains the public key in PEM format and can be used to verify GBL files using commander gbl parse. **signing-key-tokens.txt** contains the public key in token format, suitable for writing to the EFR32 device.

Generating a Signing Key Using a Hardware Security Module

When using a Hardware Security Module, the private key is kept secret inside the HSM. According to the instructions from your HSM vendor, have it generate an ECDSA-P256 key pair and export the public key in PEM format to the file **signing-key.pub**. Then use Simplicity Commander to convert the key to token format, suitable for writing to the EFR32 device.

```
commander gbl keyconvert --type ecc-p256 signing-key.pub --outfile signing-key-tokens.txt
```

Generating an Encryption Key

```
commander util genkey --type aes-ccm --outfile encryption-key
```

This creates an AES-128 key for encryption in the file **encryption-key**. The file has token format, making it suitable to write to the EFR32 device using `commander flash --tokenfile` .

Writing Keys to the Device

To write the two token files containing the encryption key and public key as manufacturing tokens to the device, issue the following command:

```
commander flash --tokengroup znet --tokenfile encryption-key --tokenfile signing-key-tokens.txt
```

Signing an Application Image for Secure Boot

If the bootloader enforces Secure Boot, the application needs to be signed to pass verification. On every boot, a SHA-256 digest of the application is calculated. The signature is verified using ECDSA-P256, with the same public key as for the GBL file signing. Signature verification failure prevents the application from booting.

Application images should contain an **ApplicationProperties_t** struct declaring the application version, capabilities, and other metadata. If **ApplicationProperties_t** is missing, the application image cannot be signed. For more details on adding **ApplicationProperties_t**, see section Application Properties.

Using Simplicity Commander

Signing the application can be done with the command:

```
commander convert myapp.s37 --secureboot --keyfile signing-key --outfile myapp-signed.s37
```

Using a Hardware Security Module

The application can be prepared for signing by issuing the command:

```
commander convert myapp.s37 --secureboot --extsign --outfile myapp-for-signing.s37
```

Creating a Signed and Encrypted GBL Upgrade Image File from an Application

To create a GBL file from an application, use `commander gbl create` .

Copyright © 2025 Silicon Laboratories. All rights reserved.

Note that, as of this writing, secure application images can only be constructed through Simplicity Commander, not through the configuration options available through Simplicity Studio.

Application images should contain an **ApplicationProperties_t** struct declaring the application version, capabilities, and other metadata. If **ApplicationProperties_t** is missing, the application image cannot be signed. For more details on adding **ApplicationProperties_t**, see section Application Properties.

Using Simplicity Commander to Sign

For an application called **myapp.s37**, use:

```
commander gbl create myapp.gbl --app myapp.s37 --sign signing-key --encrypt  encryption-key
```

This single command performs three actions:

- Creates a GBL file
- Encrypts the GBL file
- Signs the GBL file

If Secure Boot is also desired, the application must be signed using `commander convert --secureboot` prior to creating the GBL.

Using a Hardware Security Module to Sign

For an application called **myapp-signed.s37**, which has previously been signed for Secure Boot, use:

```
commander gbl create myapp-for-signing.gbl --app myapp-signed.s37 --extsign --encrypt  encryption-key
```

This command performs the following actions:

- Creates a partial GBL file
- Encrypts the partial GBL file
- Prepares the partial GBL file for signing by an external signer

Using an HSM, sign the output file **myapp-for-signing.gbl.extsign**, and supply the resulting DER-formatted signature file **signature.der** back to Simplicity Commander:

```
commander gbl sign myapp-for-signing.gbl.extsign --signature signature.der --verify signing-key.pub --outfile myapp.gbl
```

The GBL file is not valid until the signature has been applied using gbl sign.

System Security Considerations

The Gecko bootloader security features can be used to create a secure device, but do not create a secure system by themselves. This section goes over considerations that need to be taken when designing a secure system where the Gecko Bootloader is a component.

Key Storage

On Series 1 devices, the decryption and public sign keys used by the Gecko bootloader are stored in the Lockbits page in flash. To prevent a rogue application from being able to change or wipe the keys, the Lockbits page should be write protected after the keys have been written in manufacturing.

On Series 2 devices, the decryption key and the sign key used by the Gecko Bootloader can either be stored in the topmost page of the main flash or in the Secure Engine OTP. The decryption key can be provisioned in the Secure Engine OTP using Simplicity Commander or using the Secure Engine Mailbox interface. The keys are not accessible from the user applications on EFR32xG22 devices, which means that the keys used by Gecko Bootloader need to be stored in the topmost page of the main flash. Once a key value has been programmed into the Secure Engine OTP, it cannot be changed. For more details, refer to Series 2 Secure Boot with RTSL and Simplicity Commander Reference Guide.

The default behavior of the Gecko Bootloader is to use the sign key stored in the Secure Engine OTP to perform the signature verification. If the sign key is not provisioned, the Gecko Bootloader will try to use the public sign key stored in

Copyright © 2025 Silicon Laboratories. All rights reserved.

the topmost page of the main flash. This fallback mechanism can be disabled by disabling the “Allow use of public key from manufacturing token storage” component option from the **Bootloader Core** component in Simplicity Studio.

Write-Protecting the Bootloader

By default, the region in flash used by the Gecko bootloader is readable and writeable. The region needs to stay readable for the Application Interface to be able to interact with the bootloader. Immediately after reset, the region also needs to be writable to be able to perform bootloader upgrades.

Series 1

It is possible to write-protect the bootloader region on EFR32xG12 and newer. This is done by setting the write-once MSC_BOOTLOADERCTRL_BLWDIS bit on every bootup on Series 1 devices. The pages are write-protected until the next reboot. This is done by the Gecko bootloader main stage if the “Prevent bootloader write/erase” component option is enabled in the **Bootloader Core** component in Simplicity Studio.

On EFR32xG1, where the bootloader resides in main flash rather than the information block, the BLWDIS option does not exist. On EFR32xG1, the first flash page containing the first stage bootloader can be write-protected with a Page Lock word, using commander `device protect --write --range 0x0:+0x800`. If bootloader upgrades are to be supported, the pages containing the main bootloader need to stay writeable.

> **Note**: Setting MSC_BOOTLOADERCTRL_BLWDIS will also prevent debuggers from flashing a new bootloader. This means that debug tools that do not completely halt and reset the target device before re-flashing might fail to flash the new bootloader, as the flash is write-protected. If you are unsure whether your debug tools will handle this gracefully, Silicon Labs recommends keeping this setting disabled during development, and enabling it before going into production. If your debug tools halt and reset the target device before flashing, this is not an issue, and Silicon Labs recommends enabling this setting early in the development cycle.

Series 2

On Series 2 devices, the write-once MSC_PAGELOCKx register is used to write-protect the pages used by the bootloader if the “Prevent bootloader write/erase” component option is enabled in the **Bootloader Core** component in Simplicity Studio. The pages are write-protected until the next reboot.

Write-Protecting the Application

On Series 2 devices, it is also possible to write-protect the application. When the component option “Prevent write/erase of verified application” is enabled, the bootloader will write-protect the pages used by the application after successfully verifying the application signature, before allowing the application to start. This option is only available when Secure Boot of the application is enabled. The MSC_PAGE- LOCKx register is used to protect the pages, and the write protection lasts until the next reboot.

Debug Access

Series 1

To prevent debugger access to the Series 1 devices, the Debug Lock word needs to be written. Device recovery after enabling the Debug Lock is possible but will erase the main flash and the Lockbits page. This will erase the main application and the key material stored in the Lockbits page. The Userdata page and bootloader area will survive Debug Unlock, so secrets should not be stored in these locations. To debug lock the device, issue `commander device lock --debug enable`. The AAP Lock can be used to permanently lock the debug port. This also prevents Debug Unlock. To AAP lock the device, please see the reference manual for your device for the address location of the AAP lock word, and use `commander flash --patch` to write the appropriate value to this address.

Series 2

On Series 2 devices, debugger access can be prevented through Secure Engine. Debugger access can be re-enabled after enabling the Debug Lock by issuing device erase. To debug-unlock the device, issue commander device lock --debug

Copyright © 2025 Silicon Laboratories. All rights reserved.

disable from Simplicity Commander. This also erases the main flash, which results in the top page of main flash that stores the encryption and signing keys being erased. To permanently lock the debug port, the device erase disable command can first be issued through Secure Engine. Thereafter, the debug lock command can be issued to the Secure Engine. For more details, refer to Series2 Secure Debug.

Disabling Debug Access from Software

The software APIs `DBG_DisableDebugAccess(dbgLockModeAllowErase)` and `DBG_DisableDebugAccess (dbgLockModePermanent)` from the EMLIB DBG module can be used from the application to lock the debug interface.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Gecko Bootloader and Trustzone

Gecko Bootloader and TrustZone

With GSDK v4.2.2, Gecko Bootloader comes with TrustZone support to build TrustZone aware solutions. This section outlines the major points that should be considered when using a TrustZone aware bootloader.

Gecko Bootloader Operation

A TrustZone aware application can have the following configurations:

- Secure app only
- Non-secure app only
- Secure and Non-secure pair

With TrustZone enabled, the application bootloaders are entirely configured as Secure applications whereas the communication bootloaders are split into Secure and Non-secure application pairs. The communication interfaces are typically in the Non-secure part of the application and the GBL parser, and other functionalities are in the Secure part of the application. TrustZone aware applications are generated and built using pre-defined workspace applications. Refer to Series 2 TrustZone and Software Project Generation and Configuration with SLC-CLI for more details regarding TrustZone workspaces and the SLC CLI tool.

For the communication bootloaders to function properly, a combined bootloader binary image (containing both the Secure and Non-secure binary images) needs to be programmed onto the device. A combined bootloader binary can be obtained by either using the post-build steps defined for the workspace or using the Commander tool. Refer to *UG162: Simplicity Commander Reference Guide* for more details on how to use the Commander tool to combine multiple binaries into a single binary. Application bootloaders need to be built with the **Bootloader TrustZone Secure** component when building a TrustZone aware solution. The **Bootloader TrustZone Secure** component is required to set up the necessary configuration for the application to function correctly within a TrustZone aware solution.

To create a bootloader upgrade file, the combined bootloader binary (Secure and Non-secure) should be used. Refer to Series 2 TrustZone for more details on upgrading an existing application to TrustZone. Special care must be taken while building the TrustZone aware communication bootloaders for the xG21 family as the bootloader size increases from 16kB (non TrustZone aware solution) to 24kB (TrustZone aware solution). Similarly, the Bootloader – SoC Bluetooth Apploader OTA DFU application's size increases from 72kB to 80kB with TrustZone enabled.

Gecko Bootloader Configuration

The following table provides three of the configuration options available as part of the Bootloader Interface with TrustZone enabled.

Configuration Option	Description
BOOTLOADER_DISABLE_OLD_BOOTLOADER_MITIGATION	Disables multi-tiered fallback logic. The fault handling logic as well as the USART auto-detection logic will be disabled. Default value is set to 0.
BOOTLOADER_DISABLE_NVM3_FAULT_HANDLING	Disables peripheral access fault handling. The fault handling triggered by an erroneous access of peripherals will be disabled. This should be disabled if all the peripherals that are in use by the bootloader have been properly configured by the Manually override security state of peripherals option. Default value is set to 0.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Configuration Option	Description
BOOTLOADER_MANUAL_OVERRIDE_SECURITY_STATE	Manually override the security state of peripherals in use by the bootloader. This option can be used to manually set a peripheral's access state before calling into the bootloader. Default value is set to 0.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Gecko Bootloader and Delta DFU Upgrades

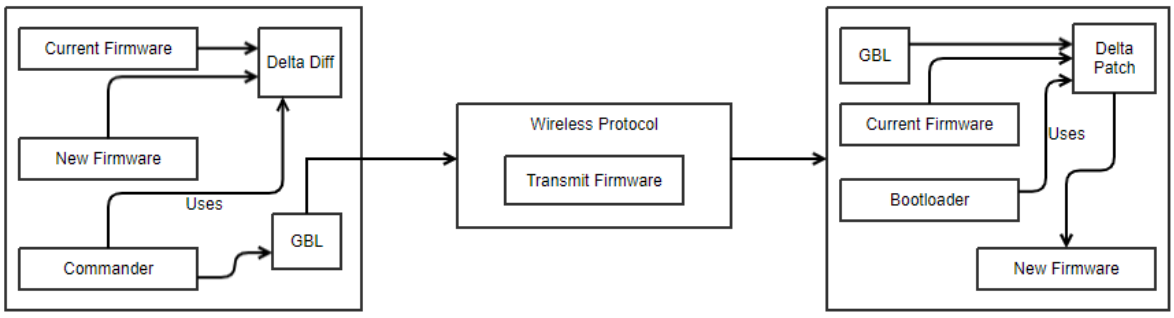
Gecko Bootloader and Delta DFU Upgrades

With GSDK v2024.6.1, Gecko Bootloader support comes with support for delta DFU upgrades. This section provides an introduction to Delta DFU upgrades and describes the major points that should be considered while using the Delta DFU component.

Introduction to Delta DFU

The main function of the image delta DFU is to calculate the difference between the current firmware stored on the device and the new firmware to be updated. This difference is used to create a patch file that can be applied to the current firmware, resulting in the new firmware. The patch file is created using the Simplicity Commander tool, transferred to the device with any wireless protocol as a GBL, and applied on top of the current firmware by the Gecko Bootloader. The patch file creation and application are both implemented in a library that is shared between these tools.

The following diagram illustrates the complete process.



Gecko Bootloader Operation

The Delta DFU upgrades can be enabled by including the Delta DFU component in the Gecko Bootloader. Simplicity Commander supports the creation of a Delta DFU GBL file.

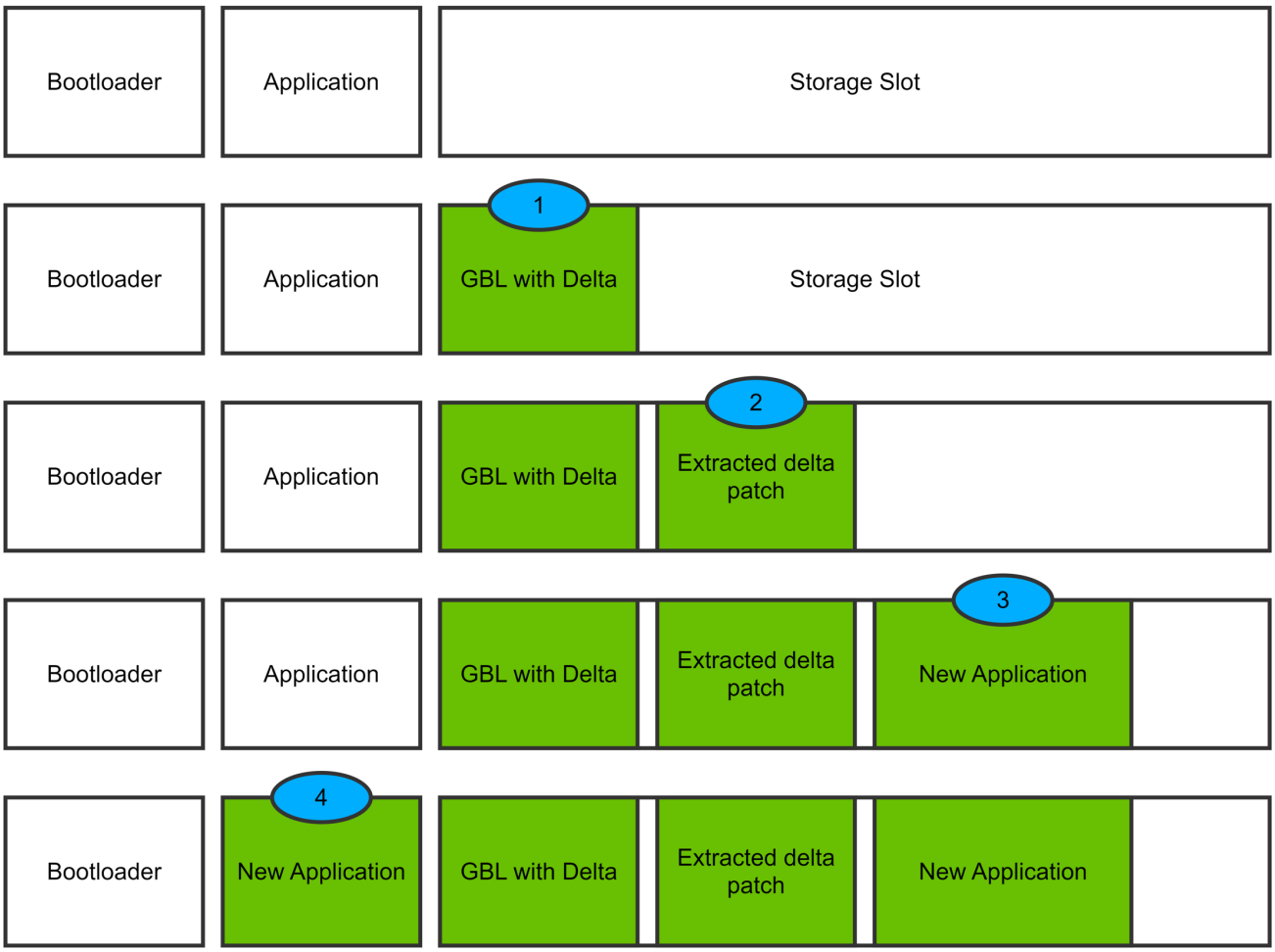
With Delta DFU enabled, the application bootloaders are equipped to accept Delta DFU GBL files. These GBL files contain a patch file that is extracted into the slot. The Gecko Bootloader creates the new firmware using this patch file and the currently running firmware within the slot. The slot size should have enough number of pages to hold the GBL file, patch file, and the new firmware binary in the slot. Each file starts on a new page.

The Delta DFU upgrades are triggered in the same way as in an application bootloader. Once the bootloader finds a GBL that contains Delta Patch, it extracts the patch file and then re-constructs the new firmware file within the slot. If the new firmware is created correctly, then the bootloader copies this new firmware to the application area, completing the upgrade.

To create the GBL file, Commander needs two input files, one of the firmware that is current and one that is the upgrade. For Delta DFU upgrades to work, it is essential that the firmware on the device and the firmware used to create the GBL are the same. Also, the version dependency tag is included in Delta DFU and should be correctly set by the user.

Delta DFU operation is illustrated in the following figure:

Copyright © 2025 Silicon Laboratories. All rights reserved.



Copyright © 2025 Silicon Laboratories. All rights reserved.

Application Interface

Application Interface

The bootloader has an application interface exposed through a function table in the bootloader. The application interface provides APIs to use bootloader functions for storing and retrieving upgrade images and verifying their integrity. APIs to reboot into the bootloader are also provided. For details see the Gecko Bootloader API Reference, shipped with the SDK in the platform/bootloader/documentation folder.

If you are not using a protocol stack from Silicon Labs, the **api/btl_interface.h** header provides the bootloader application interface API. If you are using a protocol stack from Silicon Labs, the recommended bootloader interface API for the specific protocol stack should be used instead. The following files provide the implementation of the bootloader interface:

api/btl_interface.c (common interface)

api/btl_interface_storage.c (interface to storage functionality)

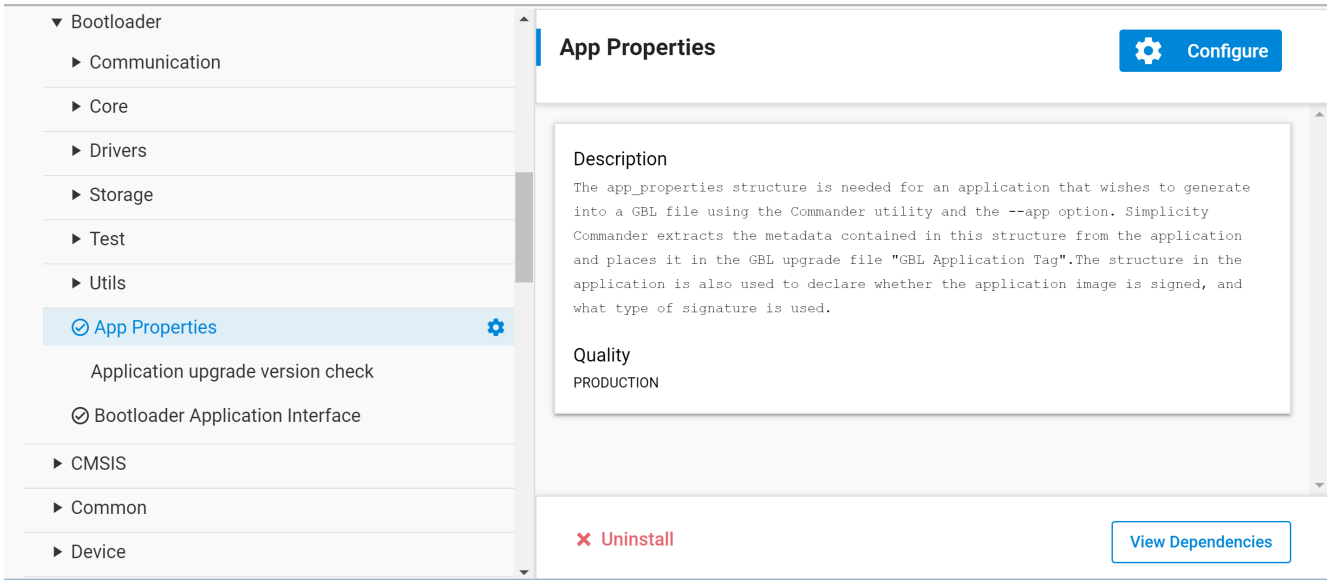
The application interface consists of functions that can be included into the customer application, and that communicate with the bootloader through the **MainBootloaderTable_t**. This table contains function pointers into the bootloader. The 10th word of the bootloader contains a pointer to this structure, allowing any application to easily locate it. Using the wrapper functions provided in the Bootloader Interface API is preferred over accessing the bootloader table directly. Modules include:

- **Application Parser Interface**: Application interface for interfacing with the bootloader image parser.
- **Application Storage Interface**: Application interface for interfacing with the bootloader storage. The Storage Interface is only available on bootloaders that support the storage interface.
- **Common Application Interface**: Generic application interface available on all versions of the bootloader, independently of which components are present.

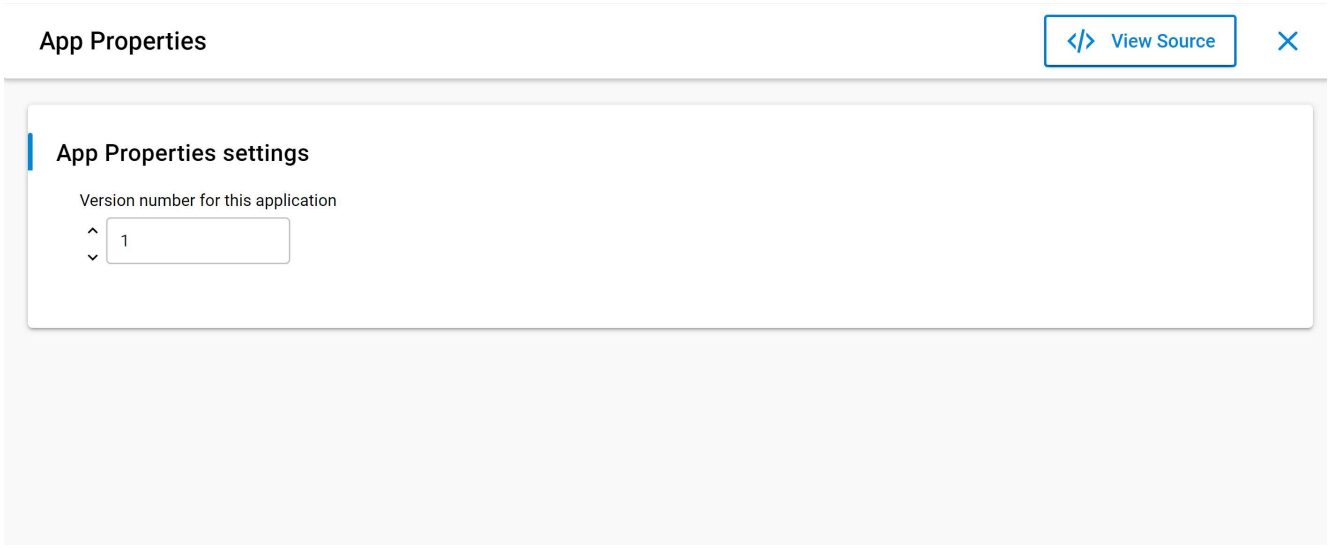
Application Properties

Application images should contain an **ApplicationProperties_t** struct declaring the application version, capabilities, and other metadata. Simplicity Commander extracts the metadata contained in this structure from the application and places it in the GBL upgrade file **GBL Application Tag**. If the structure is not present in the application, Simplicity Commander will raise an error. The **ApplicationProperties_t** struct is added to the application on installing the **bootloader_interface** component to the application. The **bootloader_interface** component installs **bootloader_app_properties** component which adds an instance of **ApplicationProperties_t** named **sl_app_properties** to the project. The component adds a source file named **app_properties.c** and a configuration file named **app_properties_config.h**. This configuration file allows users to configure the application version via Simplicity Studio's Component Editor. To open the Component Editor, locate the **App Properties** component under **Platform > Bootloader** as shown below.

Copyright © 2025 Silicon Laboratories. All rights reserved.



Click **Configure** to open the Component Editor.



The application type is automatically populated based on the wireless stack used in building the project. The value of the application type is automatically set during the project generation step and can be found in **autogen/sl_application_type.h** file.

The contents of the **GBL Application Tag** can be extracted from a GBL file by a running application using the Application Storage interface. Note that the **GBL Application Tag** will only be added if the GBL file contains an application image, not if the GBL file only contains a bootloader upgrade or metadata.

The structure in the application is also used to declare whether the application image is signed, and what type of signature is used. This information is added by Simplicity Commander when signing the image using `commander convert (--secureboot, --extsign or -- signature)`. For the bootloader to locate the **ApplicationProperties_t** struct, if not already done by the linker, Simplicity Commander modifies word 13 of the application to insert a pointer to the **ApplicationProperties_t** struct when signing the application image for Secure Boot.

Error Codes

Copyright © 2025 Silicon Laboratories. All rights reserved.

Most Gecko bootloader APIs return error codes. The following table lists the groups of error codes that may be returned. The full list of error codes within each group can be found in *api/btl_errorcode.h* in the platform/bootloader directory of the SDK, as well as in the API Reference.

ID	Description
0x0	OK
0x01xx	Initialization error
0x02xx	Image verification error
0x04xx	Storage error
0x05xx	Bootload error
0x06xx	Security error
0x07xx	Communication error
0x09xx	XMODEM parser error
0x10xx	GBL file parser error
0x11xx	SPI slave driver error
0x12xx	UART driver error
0x13xx	Compression error

Copyright © 2025 Silicon Laboratories. All rights reserved.

Series 2 Secure Boot with RTSL

Series 2 Secure Boot with RTSL

NOTE: This section replaces *AN1218: Series 2 Secure Boot with RTSL*. Further updates to this application note will be provided here.

This application note describes the design of with RTSL (Root of Trust and Secure Loader) on devices. It also provides examples of how to implement the process. For more information on using the Gecko Bootloader with devices, see the following:

- Bootloader Fundamentals
- *UG266: Silicon Labs Gecko Bootloader User's Guide for GSDK 3.2 and Lower*
- Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher

Key Points

- Compares the process in and devices
- Describes the with RTSL components and process
- Provides examples of configuring devices for the process
- Recovers secure boot failure devices

Copyright © 2025 Silicon Laboratories. All rights reserved.

Series 2 Device Security Features

Series 2 Device Security Features

Protecting IoT devices against security threats is central to a quality product. Silicon Labs offers several security options to help developers build secure devices, secure application software, and secure paths of communication to manage those devices. Silicon Labs' security offerings were significantly enhanced by the introduction of the Series 2 products that included a Secure Engine. The Secure Engine is a tamper-resistant component used to securely store sensitive data and keys and to execute cryptographic functions and secure services.

On Series 1 devices, the security features are implemented by the TRNG (if available) and CRYPTO peripherals.

On Series 2 devices, the security features are implemented by the Secure Engine and CRYPTOACC (if available). The Secure Engine may be hardware-based, or virtual (software-based). Throughout this document, the following abbreviations are used:

- HSE - Hardware Secure Engine
- VSE - Virtual Secure Engine
- SE - Secure Engine (either HSE or VSE)

Additional security features are provided by Secure Vault. Three levels of Secure Vault feature support are available, depending on the part and SE implementation, as reflected in the following table:

Level (1)	SE Support	Part (2)
Secure Vault High (SVH)	HSE only (HSE-SVH)	Refer to IoT Endpoint Security Fundamentals for details on supporting devices.
Secure Vault Mid (SVM)	HSE (HSE-SVM)	"
"	VSE (VSE-SVM)	"
Secure Vault Base (SVB)	N/A	"

Notes:

1. The features of different Secure Vault levels can be found in Security.
2. IoT Endpoint Security Fundamentals.

Secure Vault Mid consists of two core security functions:

- **Secure Boot**: Process where the initial boot phase is executed from an immutable memory (such as ROM) and where code is authenticated before being authorized for execution.
- **Secure Debug access control**: The ability to lock access to the debug ports for operational security, and to securely unlock them when access is required by an authorized entity.

Secure Vault High offers additional security options:

- **Secure Key Storage**: Protects cryptographic keys by "wrapping" or encrypting the keys using a root key known only to the HSE-SVH.
- **Anti-Tamper protection**: A configurable module to protect the device against tamper attacks.
- **Device authentication**: Functionality that uses a secure device identity certificate along with digital signatures to verify the source or target of device communications.

A Secure Engine Manager and other tools allow users to configure and control their devices both in-house during testing and manufacturing, and after the device is in the field.

User Assistance

Copyright © 2025 Silicon Laboratories. All rights reserved.

In support of these products, Silicon Labs offers whitepapers, webinars, and documentation. The following table summarizes the key security documents:

Document	Summary	Applicability
Series 2 Secure Debug	How to lock and unlock Series 2 debug access, including background information about the SE	Secure Vault Mid and High
Series 2 Secure Boot with RTSL (this document)	Describes the secure boot process on Series 2 devices using SE	Secure Vault Mid and High
AN1222: Production Programming of Series 2 Devices	How to program, provision, and configure security information using SE during device production	Secure Vault Mid and High
Anti-Tamper Protection Configuration and Use	How to program, provision, and configure the anti-tamper module	Secure Vault High
Authenticating Silicon Labs Devices using Device Certificates	How to authenticate a device using secure device certificates and signatures, at any time during the life of the product	Secure Vault High
Secure Key Storage	How to securely "wrap" keys so they can be stored in non-volatile storage.	Secure Vault High

Key Reference

Public/Private keypairs along with other keys are used throughout Silicon Labs security implementations. Because terminology can sometimes be confusing, the following table lists the key names, their applicability, and the documentation where they are used.

Key Name	Customer Programmed	Purpose
Public Sign key (Sign Key Public)	Yes	Secure Boot binary authentication and/or OTA upgrade payload authentication
Public Command key (Command Key Public)	Yes	Secure Debug Unlock or Disable Tamper command authentication
OTA Decryption key (GBL Decryption key) aka AES-128 Key	Yes	Decrypting GBL payloads used for firmware upgrades
Attestation key aka Private Device Key	No	Device authentication for secure identity

SE Firmware

Silicon Labs strongly recommends installing the latest SE firmware on Series 2 devices to support the required security features. Refer to AN1222: Production Programming of Series 2 Devices for the procedure to upgrade the SE firmware and IoT Endpoint Security Fundamentals for the latest SE Firmware shipped with Series 2 devices and modules.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Secure Boot Process

Secure Boot Process

Introduction

The purpose of Secure Boot is to protect the integrity of the behavior of the system. Because the behavior of the system is defined by the firmware running on it, Secure Boot acts to ensure the authenticity and integrity of the firmware. Secure Boot is a foundational component of platform security, and without it, other security aspects such as secure storage, secure transport, secure identity, and data confidentiality can often be subverted through the injection of malicious code.

Secure Boot works as a process by which each piece of firmware is validated for authenticity and integrity before it is allowed to run. Each authenticated module can also validate additional modules before executing them, forming a chain of trust. If any module fails its security check, it is not allowed to run, and program control will typically stall in the validating module. In most lightweight IoT systems, the behavior of a Secure Boot failure is to cause the device to stop working until an authentically signed image can be loaded onto it. Whereas this may seem extreme, it is a better outcome than a smart light bulb being repurposed to mine crypto-currency, or a smart speaker being repurposed as a surveillance device on the end user's private conversations.

The first link in the chain of trust is the root of trust. This is often the weakest link in the Secure Boot chain because the root of trust itself is not checked for authenticity or integrity. The security strength of the root of trust lies in its immutability. The strongest roots of trust have their firmware origin in ROM and use a Public Sign Key that is also located in ROM.

Wireless SoC Series 1 and Series 2 devices both use a two-stage boot design consisting of a non-upgradable first stage root of trust followed by an upgradable second stage. In Series 1 devices, the root of trust (also called the first-stage bootloader) is in flash rather than ROM, and the upgradable portion (the main bootloader) is checked for integrity using a CRC32 checksum but is not checked for authenticity using a Public Sign Key. In Series 2 devices, the root of trust is in ROM, and the upgradable portion is checked both for integrity and authenticity.

The Secure Boot with RTSL is implemented by Root code executed by the Hardware Secure Engine (HSE) or the Cortex-M33 operating in Root Mode (VSE). For more information about SE, see section *Secure Engine Subsystem* in Series 2 Secure Debug.

Silicon Labs provides Custom Part Manufacturing Service (CPMS) to customize the users' security features and settings.

This application note uses the following abbreviations:

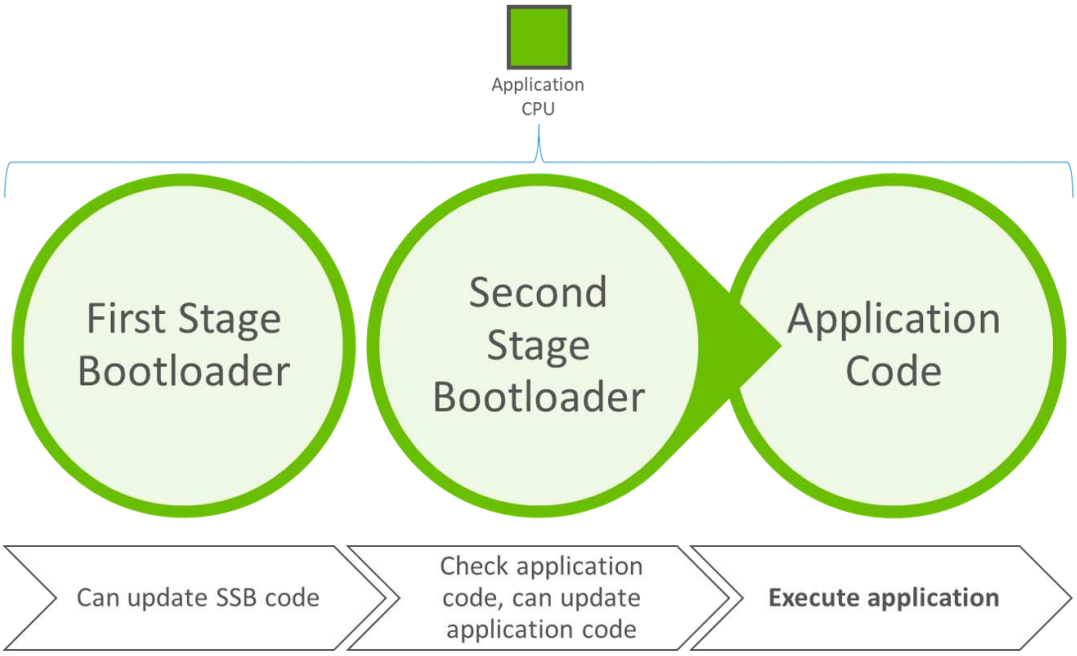
- FSB - First Stage Bootloader
- SSB - Second Stage Bootloader
- GBL - Gecko Bootloader
- RTSL - Root of Trust and Secure Loader
- HSM - Hardware Security Module
- OTP - One-Time Programmable
- WSTK - Wireless Starter Kit
- GSDK - Gecko Software Development Kit. For more info refer GSDK.
- ECDSA-P256-SHA256 - Elliptic Curve Digital Signature Algorithm aka ECSDA using a P-256 curve and a SHA256 hash
- PEM (.pem) - Privacy Enhanced Mail
- DER (.der) - Distinguished Encoding Rules

ECDSA-P256-SHA256 Secure Boot in Series 1 Devices

The Secure Boot process for Series 1 (SVB) devices originates in flash, typically with the execution of the first stage of GBL. The first stage of GBL checks to see if an upgrade is pending for the second stage of GBL. If so, it processes the upgrade of the second stage and then executes it. Otherwise, it just executes the second stage. If Secure Boot is enabled, the second stage of GBL checks the integrity and authenticity of the application image before executing it. If the integrity

Copyright © 2025 Silicon Laboratories. All rights reserved.

check fails, program control remains in the SSB. The following figure illustrates the Secure Boot process on Series 1 devices.



See UG266/Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher for more information to generate and download signed firmware images using Simplicity Commander.

ECDSA-P256-SHA256 Secure Boot in Series 2 Devices

For Series 2 devices, the Secure Engine (SE) implements the FSB to authenticate and upgrade the SSB. The GBL implements the SSB (aka Main Bootloader in UG266/Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher to authenticate and upgrade the application firmware.

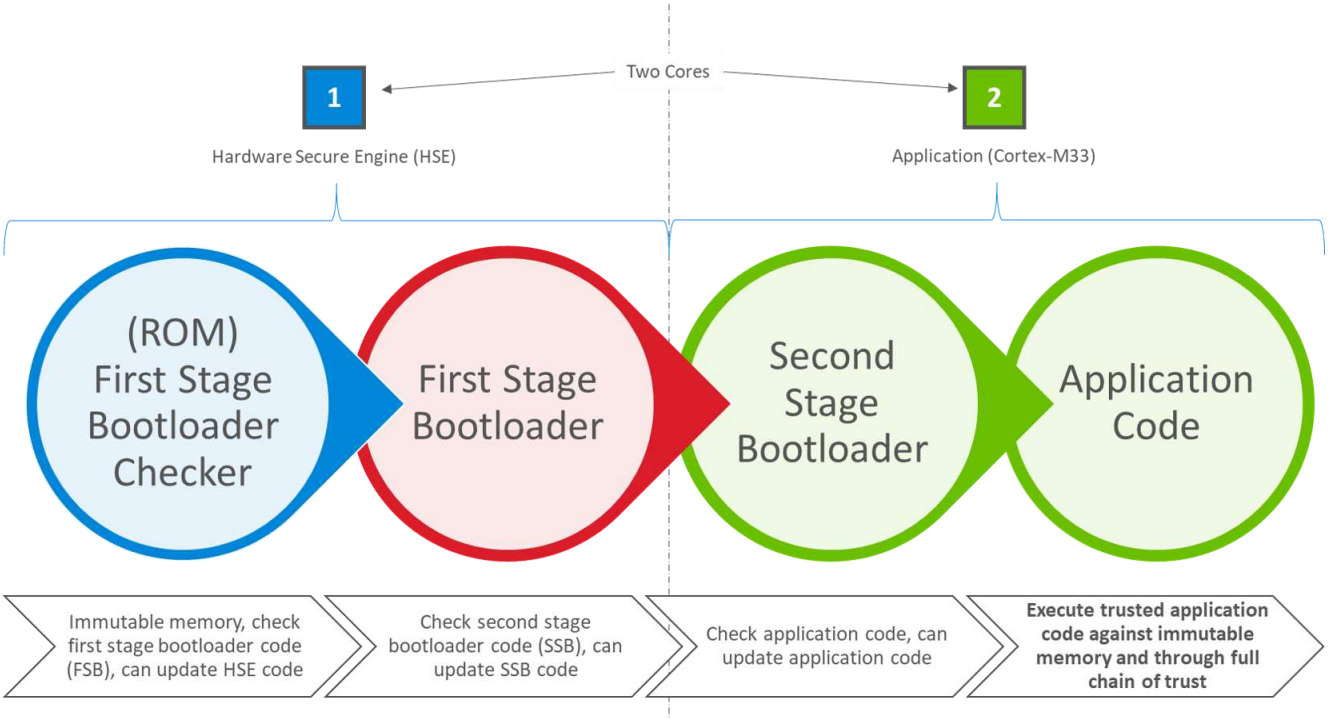
Refer to the "Gecko Bootloader Security Features" section in UG266/Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher and ECDSA-P256-SHA256 Secure Boot example for more information about the ECDSA-P256-SHA256 secure boot process in Series 2 devices.

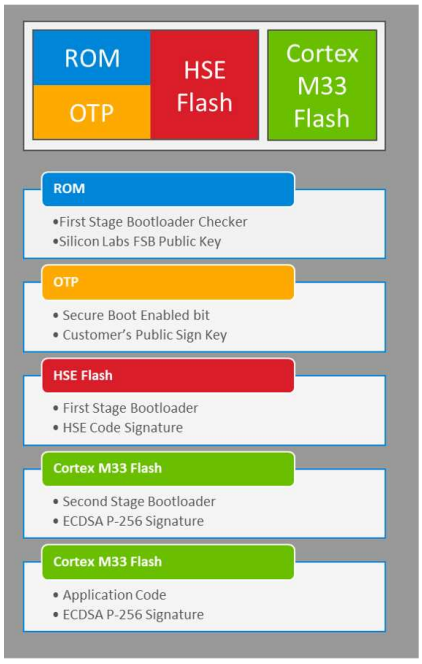
> **Note**: It is possible to have a 2-stage design to skip the SSB between FSB and application. However, the application cannot be upgraded if discarding the SSB, and this application note assumes the SSB is present.

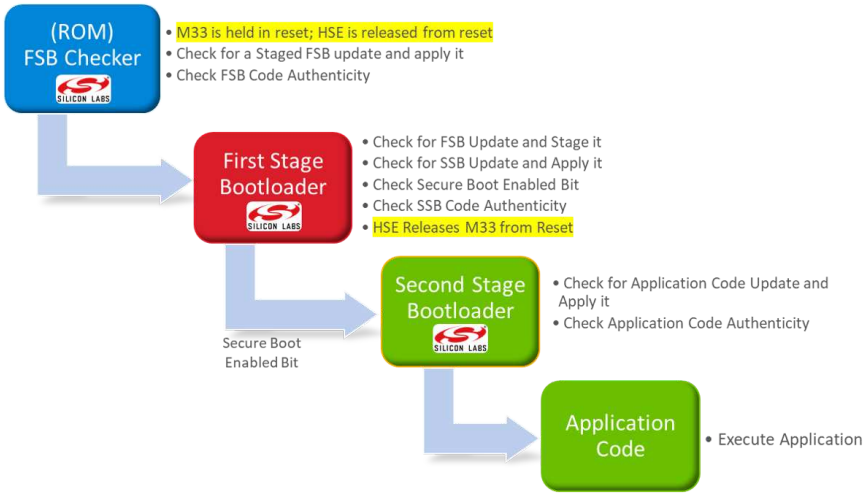
HSE

In HSE-SVM and HSE-SVH devices, the Secure Boot process originates in ROM contained in the security co-processor (HSE). The following figures illustrate the Secure Boot process and flow on Series 2HSE devices.

Copyright © 2025 Silicon Laboratories. All rights reserved.





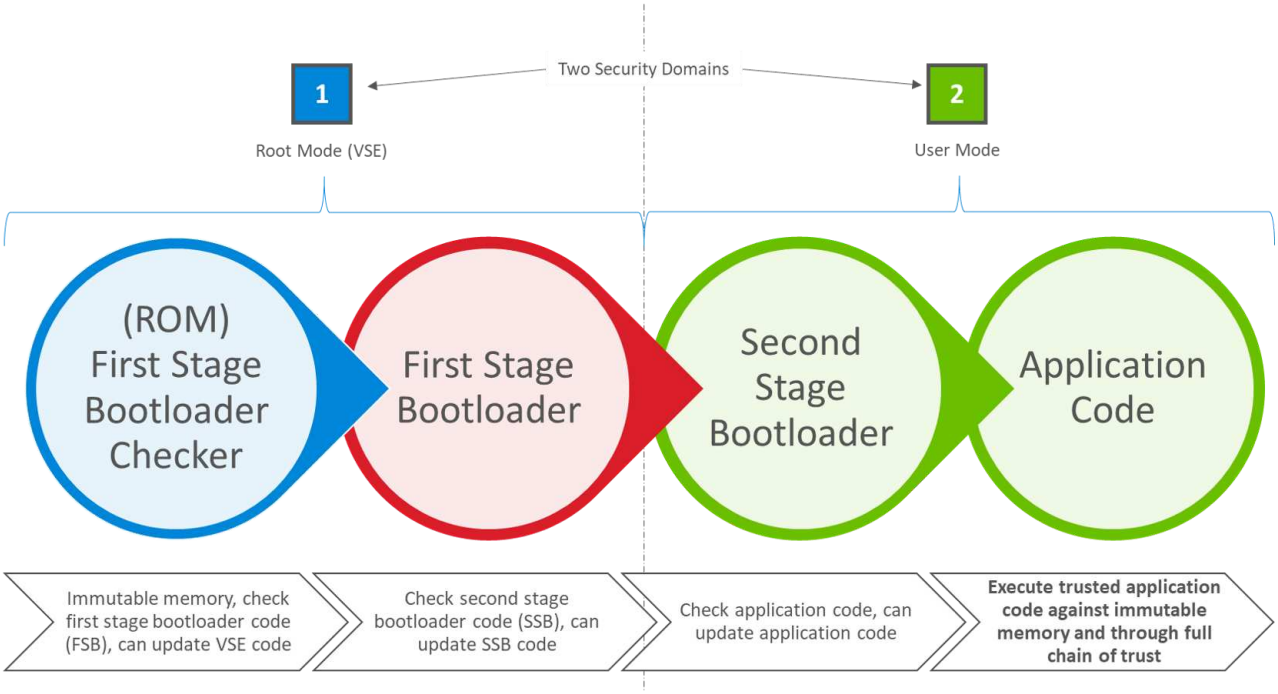


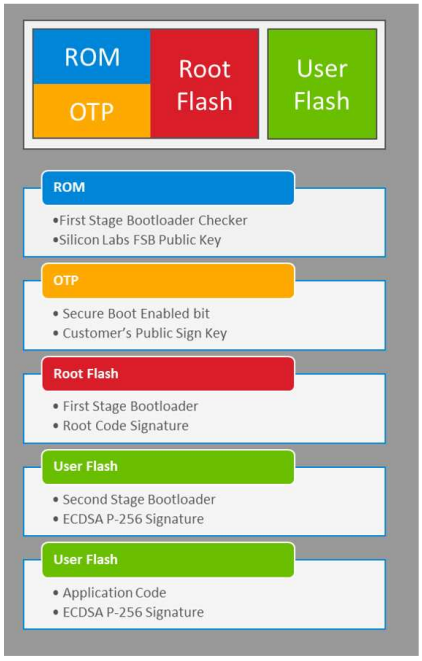
3

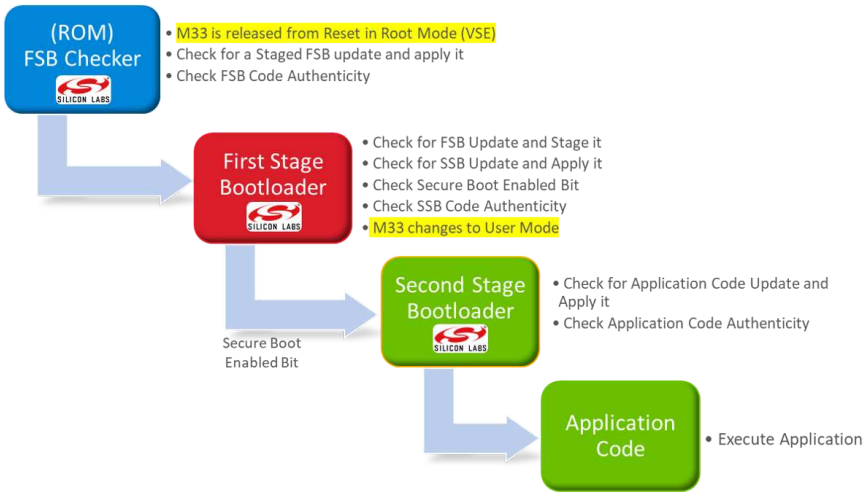
VSE

In VSE-SVM devices, the host MCU (Cortex-M33) assumes an elevated security state out of reset and securely boots itself from code that originates in ROM. The following figures illustrate the Secure Boot process and flow on Series 2VSE devices.

Copyright © 2025 Silicon Laboratories. All rights reserved.







Certificate-based Secure Boot in Series 2 Devices

Refer to the "Gecko Bootloader Security Features" section in UG266: Silicon Labs Gecko Bootloader User's Guide for GSDK 3.2/Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher and Certificate-Based Secure Boot example for details about the certificate-based Secure Boot process in Series 2 devices.

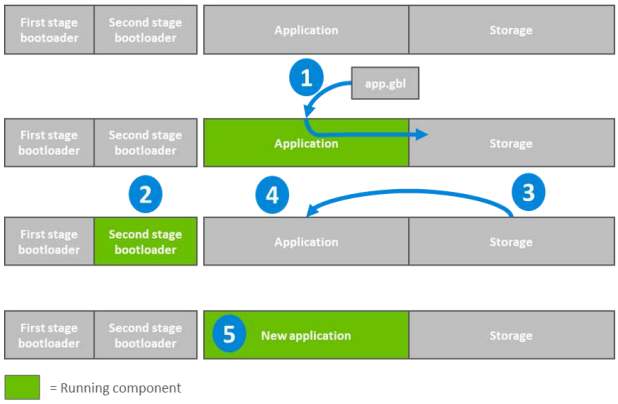
The certificate-based Secure Boot uses key delegation to minimize the exposure of the Private Sign Key, reducing the chance to revoke the Public Sign Key.

If the certificate's private key is leaked, all devices that have been programmed with that certificate are at risk until they can be updated with an image containing a certificate with a higher version.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Secure Loader

In Series 2 devices, the Secure Loader is firmware pre-loaded into the chip. Silicon Labs maintains the Secure Loader and deploys through secure upgrade packages. It is the functional equivalent of the first-stage GBL on Series 1 devices (see UG266: Silicon Labs Gecko Bootloader User's Guide for GSDK 3.2 and Lower/Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher for more information). The Secure Loader validates the authenticity and integrity of a staged image before performing an upgrade operation. The Secure Loader requires the staged image to reside on-chip and the staged image must not overlap with the target destination address range. Firmware images that originate from off-chip, either off-chip storage, external NCP host interface, or through an OTA update procedure are expected to be staged either by the application or by SSB before calling the Secure Loader to perform the upgrade.



1. The existing application downloads the upgrade image onto the storage medium of the device (for example, the app.gbl file through OTA).
2. The device reboots into the second stage bootloader, and the bootloader enters firmware upgrade mode.
3. The secure loader in the second stage bootloader verifies the integrity and authenticity of the staged upgrade image in storage, prior to performing the upgrade.
4. The second stage bootloader applies the application upgrade from the upgrade image.
5. The device boots into the application. Application upgrade is complete.

Secure Boot Time

Secure boot extends the recovery time from all sources of device reset. The duration of each authentication operation depends on the factors below:

- Computes the SHA-256 digest (32 bytes) of the associated image, which is proportional to the size of the firmware image.
- Verifies the ECDSA-P256 signature of the SHA-256 digest above, which is independent of image size.
- The clock frequency of the Crypto Engine, which is the HSE or CRYPTOACC in VSE devices.

Table: Authentication Duration

Authentication	Enable/Disable	Duration
FSB code	Enable (cannot disable)	FSB code size dependent
SSB code	Disable by default	SSB code size and SE firmware version dependent
Application code	Disable by default	Application code size and SSB firmware (GBL) version dependent

> Note:
>
> - It will extend the boot time for certificates authentication if using Certificate-Based Secure Boot.
> - Refer to device-specific datasheets (like EFR32MG21B) for data about the boot timing of Series 2 devices.
> - Refer to Secure Boot Configuration on how to enable the SSB and application code authentication.

Secure Boot Configuration

The following sections describe how to configure the Secure Boot of the SSB (GBL) and application firmware.

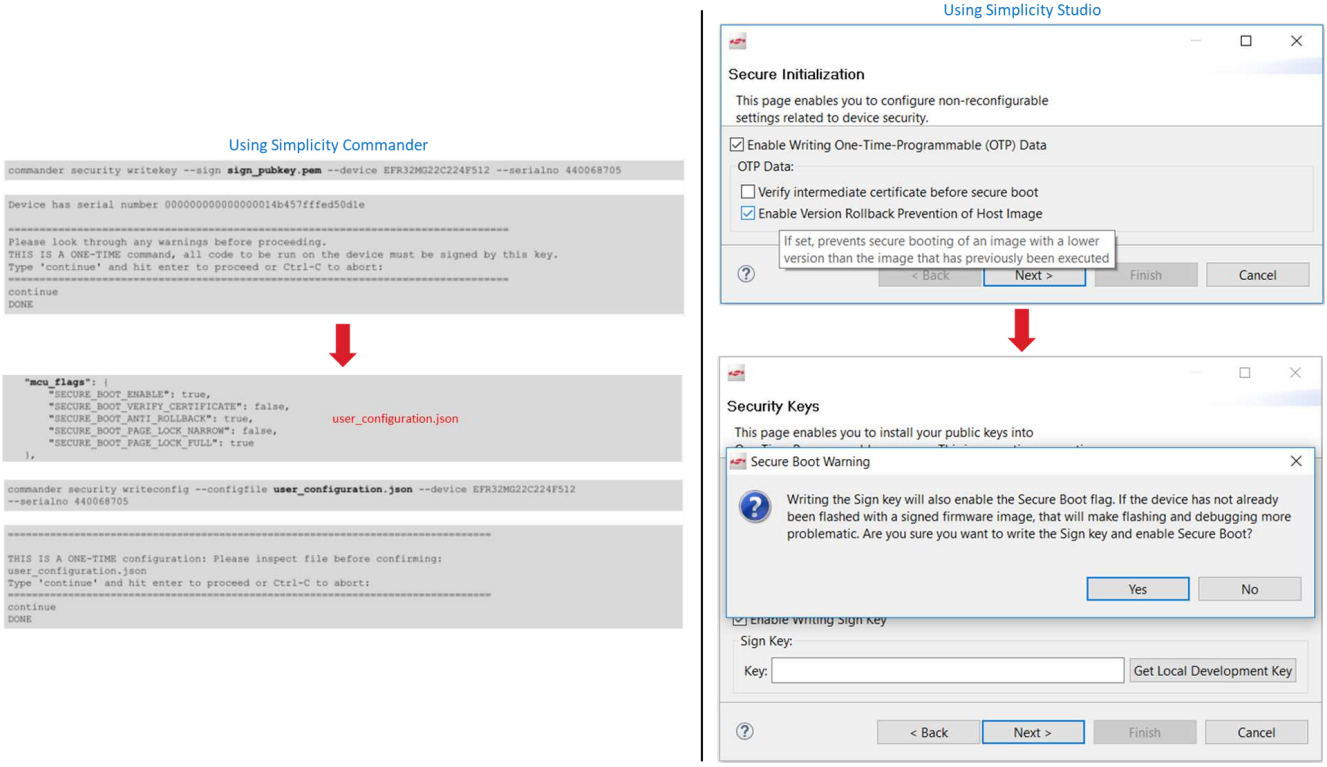
SSB

Copyright © 2025 Silicon Laboratories. All rights reserved.

In Series 2 devices, the immutable OTP memory stores the Public Sign Key and Secure Boot Enable flag. The user cannot change its respective value once either is programmed. Once the Public Sign Key is provisioned, it remains provisioned to that key value for the life of the device. Once Secure Boot is enabled, it remains enabled for the life of the device. Both of these assignment operations are **IRREVOCABLE**.

The Public Sign Key used for Series 2 devices is the public portion of an ECDSA key pair over the NIST prime curve P-256. The Public Sign Key is a customer key and is typically provisioned during the initial product manufacturing and device programming phase. It is common for all products that share the same firmware image to be loaded with the same Public Sign Key. The key loaded into the device is a public key and has no confidentiality requirements. The private key associated with that public key, which will be used to sign firmware images or certificates, should be tightly held, ideally secured in the HSM or equivalent key storage instrument.

The user can use Simplicity Commander, SE Manager, or Simplicity Studio to program the Public Sign Key and configure the SSB Secure Boot in SE OTP.



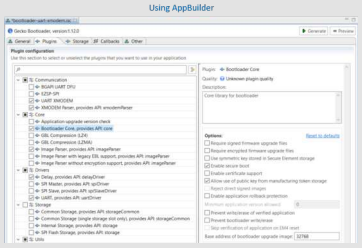
Application Firmware

The user can use the AppBuilder or Bootloader-core software component in the GBL project to configure the security options of the application firmware.

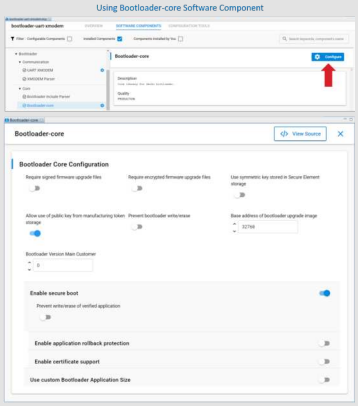
The user can reconfigure the Secure Boot configuration of the application firmware by upgrading the GBL with the new custom settings.

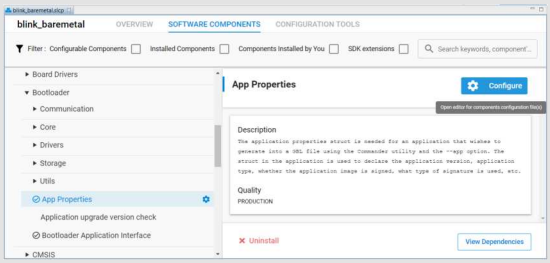
Copyright © 2025 Silicon Laboratories. All rights reserved.

Using AppBuilder



Using Bootloader-core Software Component





Copyright © 2025 Silicon Laboratories. All rights reserved.

Examples

Examples

Overview

The following table describes the examples for Series 2 Secure Boot.

Table: Series 2 Secure Boot Examples

Example	Device (Radio Board)	SE Firmware	Tool
Provision Public Sign Key and Secure Boot Enabling	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Commander
"	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	SE Manager
"	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Studio 5
Provision GBL Decryption Key	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Commander
"	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	SE Manager
Signing for ECDSA-P256-SHA256 Secure Boot	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Commander
Signing for Certificate-Based Secure Boot	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Commander
Generate a GBL Upgrade Image File	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Commander
Upgrade to Certificate-Based Secure Boot	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Commander
Certificate Revocation	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Commander
Upgrade to Secure Boot with RTSL	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	SE Manager & Simplicity Commander
Recover Devices when Secure Boot Fails	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Commander

> **Note**: Unless specified in the example, these examples can be applied to other Series 2 devices.

Using Simplicity Commander

1. This application note uses Simplicity Commander v1.12.0. The procedures and console output may be different for the other versions of Simplicity Commander. The latest version of Simplicity Commander can be downloaded from MCU Programming options.

```
commander --version
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
Simplicity Commander 1v12p0b1057
JLink DLL version: 7.52d
Qt 5.12.10 Copyright (C) 2017 The Qt Company Ltd.
EMDLL Version: 0v17p19b0
mbed TLS version: 2.16.6
Emulator found with SN=440048205 USBAddr=0
DONE
```

2. The Simplicity Commander's Command Line Interface (CLI) is invoked by `commander.exe` in the Simplicity Commander folder. The location for Simplicity Studio 5 in Windows is `C:\SiliconLabs\SimplicityStudio\v5\developer\adapter_packs\commander`. For ease of use, it is highly recommended to add the path of `commander.exe` to the system `PATH` in Windows.
3. If more than one WSTK is connected via USB, the target WSTK must be specified using the `--serialno \<J-Link serial number>` option.
4. If the WSTK is in debug mode OUT, the target device must be specified using the `--device \<device name>` option.

For more information about Simplicity Commander, see UG162: Simplicity Commander Reference Guide.

Using an External Tool

The Secure Boot examples use the **OpenSSL** to sign the image files and certificates. The Windows version of OpenSSL can be downloaded from https://slproweb.com/products/Win32OpenSSL.html. This application note uses OpenSSL Version 1.1.1h (Win64).

```
openssl version
```

```
OpenSSL 1.1.1h  22 Sep 2020
```

The OpenSSL's Command Line Interface (CLI) is invoked by `openssl.exe` in the OpenSSL folder. The location in Windows (Win64) is `C:\Program Files\OpenSSL-Win64\bin`. For ease of use, it is highly recommended to add the path of `openssl.exe` to the system `PATH` in Windows.

Using a Platform Example

Simplicity Studio 5 includes the SE Manager platform example for key provisioning and Secure Boot enabling. This application note uses platform example of GSDK v3.2.3. The console output may be different on other versions of the GSDK.

Refer to the corresponding `readme` file for details about each SE Manager platform example. This file also includes the procedures to create the project and run the example.

Generate Key and Signing

This section describes how to generate a key to sign an image file or certificate for Secure Boot.

Using Simplicity Commander

1. Run the `util genkey` command to generate the ECDSA-P256 Sign Key pair (`sign_key.pem` and `sign_pubkey.pem`) and Public Sign Key token file (sign_pubkey.txt). The Simplicity Commander can program the Public Sign Key in token file (sign_pubkey.txt) to the top page of the main flash.

```
commander util genkey --type ecc-p256 --privkey sign_key.pem --pubkey sign_pubkey.pem
--tokenfile sign_pubkey.txt
```

```
Generating ECC P256 key pair...
Writing private key file in PEM format to sign_key.pem
Writing public key file in PEM format to sign_pubkey.pem
Writing EC tokens to sign_pubkey.txt...
DONE
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

> **Note**: The same procedure can apply to generate the bootloader certificate and application certificate key pairs for Certificate-Based Secure Boot.

2. Use the `convert` command with the Private Key (like `sign_key.pem`) from step 1 to sign an image file or certificate. Refer to Signing for ECDSA-P256-SHA256 Secure Boot and Signing for Certificate-Based Secure Boot for more information about the Simplicity Commander signing process.

Using an HSM and Simplicity Commander

1. The user can use HSM to generate the ECDSA-P256 Sign Key pair. The Private Sign Key is securely held in HSM and the Public Sign Key can be exported in a specific format (like `sign_pubkey.pem`).

> **Note**: The same procedure can apply to generate the bootloader certificate and application certificate key pairs for Certificate-Based Secure Boot.

2. Use the util keytotoken command to convert the Public Sign Key from step 1 to token format (sign_pubkey.txt). The Simplicity Commander can program the Public Sign Key in token file (sign_pubkey.txt) to the top page of the main flash.

```
commander util keytotoken sign_pubkey.pem --outfile sign_pubkey.txt
```

```
Writing EC tokens to sign_pubkey.txt...
DONE
```

3. Use the convert command with --extsign option to prepare an unsigned image or certificate for HSM.
4. Use the Private Key from step 1 to generate a signature for the unsigned image or certificate from step 3.
5. Use the convert command with the signature from step 4 to generate a signed image or certificate. Refer to Signing for ECDSA-P256-SHA256 Secure Boot and Signing for Certificate-Based Secure Boot for more information about the HSM and Simplicity Commander signing process.

> **Note**: The Simplicity Commander v1.11.0 or above supports signature in DER format. The older version of Simplicity Commander can only handle signatures in Raw format.

Provision Public Sign Key and Secure Boot Enabling

The Public Sign Key in SE OTP is used to verify the host image signature or certificate during Secure Boot. The user should provision this key before setting the Secure Boot enabled flag in SE OTP. On HSE-SVH devices, the user requires to provision the anti-tamper protection configuration with Secure Boot settings.

If the user sets the `SECURE_BOOT_ANTI_ROLLBACK` option with Secure Boot, the SE will store the version counter (4 bytes) for anti-rollback of GBL (SSB) to SE flash and check the bootloader version during an upgrade and on every boot. The version counter will not roll to 0 if it reaches the maximum value (bootloader cannot upgrade anymore). The anti-rollback does not prevent flashing an older signed GBL hex image to the device.

The following table describes the anti-rollback protection on signed GBL when `SECURE_BOOT_ANTI_ROLLBACK` is enabled or disabled.

- The GBL handles the anti-rollback protection when upgrading the GBL through the GBL upgrade image file (.gbl).
- The SE handles the anti-rollback protection (if `SECURE_BOOT_ANTI_ROLLBACK` enabled) when booting the GBL.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Action	SECURE_BOOT_ANTI_ROLLBACK Disable	SECURE_BOOT_ANTI_ROLLBACK Enable
Use a GBL upgrade image file	Reject upgrade if an equal or lower GBL version is detected.	Reject upgrade if an equal or lower GBL version is detected.
Flash and boot a GBL hex image	Accept to flash and boot regardless of the GBL version.	Accept to flash regardless of the GBL version. But it cannot boot if a lower GBL version is detected.

> **Note**: It needs to execute a mass erase (commander device masserase or commander security erasedevice then reset) before flashing a GBL hex image (.s37) to the device if `SECURE_BOOT_PAGE_LOCK_NARROW` or `SECURE_BOOT_PAGE_LOCK_FULL` option in SE OTP is enabled.

For simplicity, the Secure Boot examples in this application note do not enable the following options for Secure Boot.

- `SECURE_BOOT_PAGE_LOCK_NARROW`
- `SECURE_BOOT_PAGE_LOCK_FULL`

Simplicity Commander

The following procedures assume the required files are in the same folder.

1. Follow the procedures in Generate Key and Signing to generate the ECDSA-P256 Sign Key pair (`sign_key.pem` and `sign_pubkey.pem`) and Public Sign Key token file (`sign_pubkey.txt`).
2. Run the `security writekey` command to provision the Public Sign Key (`sign_pubkey.pem`). The Public Sign Key cannot be changed once written.

```
commander security writekey --sign sign_pubkey.pem --device EFR32MG21A010F1024 --serialno 440048205
```

```
Device has serial number 00000000000000014b457fffe045b21
================================================================================
Please look through any warnings before proceeding.
THIS IS A ONE-TIME command, all code to be run on the device must be signed by this key.
Type 'continue' and hit enter to proceed or Ctrl-C to abort:
================================================================================
continue
DONE
```

3. Run the `security readkey` command to verify the Public Sign Key with the Public Sign Key in the token file (`sign_pubkey.txt`).

```
commander security readkey --sign --device EFR32MG21A010F1024 --serialno 440048205
```

```
C4AF4AC69AAB9512DB50F7A26AE5B4801183D85417E729A56DA974F4E08A562C
DE6019DEA9411332DC1A743372D170B436238A34597C410EA177024DE20FC819
DONE
```

4. For Series 2VSE devices (like EFR32MG22C224F512IM40), run the `flash` command to program the Public Sign Key in the token file (`sign_pubkey.txt`) to the top page of the main flash for ECDSA-P256-SHA256 Secure Boot. It is optional on Series 2HSE devices.

```
commander flash --tokengroup znet --tokenfile sign_pubkey.txt --device EFR32MG22C224F512IM40 --serialno 440048205
```

```
Writing 8192 bytes starting at address 0x0007e000
Comparing range 0x0007E000 - 0x0007FFFF (8 KiB)
Programming range 0x0007E000 - 0x0007FFFF (8 KiB)
DONE
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

> **Note**: The MCU Series 2 devices (like EFM32PG22C200F512IM40) require Simplicity Commander Version 1.12.2 or above to support the `flash --tokengroup znet` command.

5. Run the `security genconfig` command to generate the `user_configuration.json` file for secure boot.

```
commander security genconfig --nostore --outfile user_configuration.json --device EFR32MG21A010F1024
--serialno 440048205
DONE
```

Name	Description
SECURE_BOOT_ENABLE	If set, verifies the host image on the Cortex-M33 before releasing the Cortex-M33 from reset.
SECURE_BOOT_VERIFY_CERTIFICATE	If set, requires certificate-based signing of the host image.
SECURE_BOOT_ANTI_ROLLBACK	If set, prevents secure upgrading to a host image with a lower version than the image that is currently stored in flash.
SECURE_BOOT_PAGE_LOCK_NARROW	If set, locks flash pages that have been validated by the Secure Boot process to prevent re-flashing by other means than through the SE. Write/erase locks pages from 0 through the page where the Secure Boot host image signature is located, not including the last page if the signature is not on a page boundary.
SECURE_BOOT_PAGE_LOCK_FULL	If set, locks flash pages that have been validated by the Secure Boot process to prevent re-flashing by other means than through the SE. Write/erase locks pages from 0 through the page where the Secure Boot host image signature is located, including the last page if the signature is not on a page boundary.

> **Note**: The host image is the firmware in the device's flash starting address. It is usually the GBL.

6. Use a text editor to modify the default secure boot settings to the desired configurations used in this application note.

```
"mcu_flags": {
    "SECURE_BOOT_ENABLE": true,
    "SECURE_BOOT_VERIFY_CERTIFICATE": false,
    "SECURE_BOOT_ANTI_ROLLBACK": true,
    "SECURE_BOOT_PAGE_LOCK_NARROW": false,
    "SECURE_BOOT_PAGE_LOCK_FULL": false
}
```

ECDSA-P256-SHA256 Secure Boot

```
"mcu_flags": {
    "SECURE_BOOT_ENABLE": true,
    "SECURE_BOOT_VERIFY_CERTIFICATE": true,
    "SECURE_BOOT_ANTI_ROLLBACK": true,
    "SECURE_BOOT_PAGE_LOCK_NARROW": false,
    "SECURE_BOOT_PAGE_LOCK_FULL": false
}
```

Certificate-Based Secure Boot

> **Note**: If `SECURE_BOOT_ENABLE` is false, the SE will ignore the other four options regardless they are true or false. The EFR32xG23 and future Series 2 devices do not allow this setting to program to the SE OTP.

7. Follow the procedure in Signing for ECDSA-P256-SHA256 Secure Boot to flash a bootloader image for signature based secure boot.
   OR
   Follow the procedure in Signing for Certificate-Based Secure Boot to flash a bootloader image for certificate based secure boot
8.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Run the `security writeconfig` command to program the secure boot configuration to the SE OTP. The user can execute this command once per device.

```
commander security writeconfig --configfile user_configuration.json --device EFR32MG21A010F1024
--serialno 440048205
```

```
================================================================================
THIS IS A ONE-TIME configuration: Please inspect file before confirming:
user_configuration.json
Type 'continue' and hit enter to proceed or Ctrl-C to abort:
================================================================================
continue
DONE
```

9. Run the `security readconfig` command to check the secure boot configuration of the device.

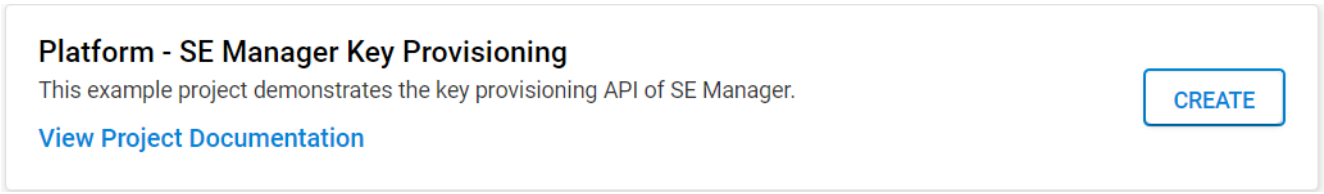
```
commander security readconfig --serialno 440048205
```





SE Manager Key Provisioning Platform Example

Click the `View Project Documentation` link to open the `readme` file.



1. Modify the default Public Sign Key in `public_sign_key[PUB_KEY_SIZE]` array in `app_process.c` to the desired values.

```
/// Public sign key
SL_ALIGN(4) static const uint8_t public_sign_key[PUB_KEY_SIZE] = {
  0xc4, 0xaf, 0x4a, 0xc6, 0x9a, 0xab, 0x95, 0x12,
  0xdb, 0x50, 0xf7, 0xa2, 0x6a, 0xe5, 0xb4, 0x80,
  0x11, 0x83, 0xd8, 0x54, 0x17, 0xe7, 0x29, 0xa5,
  0x6d, 0xa9, 0x74, 0xf4, 0xe0, 0x8a, 0x56, 0x2c,
  0xde, 0x60, 0x19, 0xde, 0xa9, 0x41, 0x13, 0x32,
  0xdc, 0x1a, 0x74, 0x33, 0x72, 0xd1, 0x70, 0xb4,
  0x36, 0x23, 0x8a, 0x34, 0x59, 0x7c, 0x41, 0x0e,
  0xa1, 0x77, 0x02, 0x4d, 0xe2, 0x0f, 0xc8, 0x19
};
```

2. Modify the default secure boot settings in `init_se_otp_conf()` function `in app_se_manager_key_provisioning.c` to the desired configuration.

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
// Overwrite secure boot options in SL_SE_OTP_INIT_DEFAULT if necessary.
otp_init.enable_secure_boot = true;
otp_init.verify_secure_boot_certificate = false;
otp_init.enable_anti_rollback = true;
otp_init.secure_boot_page_lock_narrow = false;
otp_init.secure_boot_page_lock_full = false;
```

> **Note:** If `enable_secure_boot` is false, the SE will ignore the other four options regardless of whether they are true or false. The EFR32xG23 and future Series 2 devices do not allow this setting to program to the SE OTP.

3. Follow the procedures in Generate an Unsigned Application Image to generate the unsigned application image if the GBL is present in the device.
4. Build the project and run the application. Follow the procedures in Signing for ECDSA-P256-SHA256 Secure Boot or Signing for Certificate-Based Secure Boot if a signed application image is required.
5. Then press `SPACE` to skip the programming of the AES-128 key (HSE devices only). Optional to press `ENTER` to program the hard-coded GBL Decryption Key to HSE OTP.

```
SE Manager Key Provisioning Example - Core running at 38000 kHz.
 . SE manager initialization... SL_STATUS_OK (cycles: 9 time: 0 us)
 . Get current SE firmware version... SL_STATUS_OK (cycles: 3578 time: 94 us)
 + Current SE firmware version (MSB..LSB): 00010209
 . Read SE OTP configuration... SL_STATUS_COMMAND_IS_INVALID (cycles: 3908 time: 102 us)
 . Press ENTER to program 128-bit AES key in SE OTP or press SPACE to skip.
 . Encrypt 16 bytes plaintext with 128-bit AES OTP key... SL_STATUS_FAIL (cycles: 4627 time: 121 us)
 . Press ENTER to program public sign key in SE OTP or press SPACE to skip.
```

6. Press `ENTER` to program the hard-coded Public Sign Key to SE OTP.

```
 + Warning: The public sign key in SE OTP cannot be changed once written!
 + Press ENTER to confirm or press SPACE to skip if you are not sure.
```

7. Press `ENTER` to confirm the operation. The SE returns `SL_STATUS_INVALID_PARAMETER` if the Public Sign Key is present in SE OTP.

```
 . Initialize public sign key... SL_STATUS_OK (cycles: 56052 time: 1475 us)
 . Get public sign key... SL_STATUS_OK (cycles: 8450 time: 222 us)
 + The public sign key (64 bytes):
   C4 AF 4A C6 9A AB 95 12 DB 50 F7 A2 6A E5 B4 80
   11 83 D8 54 17 E7 29 A5 6D A9 74 F4 E0 8A 56 2C
   DE 60 19 DE A9 41 13 32 DC 1A 74 33 72 D1 70 B4
   36 23 8A 34 59 7C 41 0E A1 77 02 4D E2 0F C8 19
 . Press ENTER to program public command key in SE OTP or press SPACE to skip.
```

```
 . Initialize public sign key... SL_STATUS_INVALID_PARAMETER (cycles: 4375 time: 115 us)
 . Get public sign key... SL_STATUS_OK (cycles: 8435 time: 221 us)
 + The public sign key (64 bytes):
   C4 AF 4A C6 9A AB 95 12 DB 50 F7 A2 6A E5 B4 80
   11 83 D8 54 17 E7 29 A5 6D A9 74 F4 E0 8A 56 2C
   DE 60 19 DE A9 41 13 32 DC 1A 74 33 72 D1 70 B4
   36 23 8A 34 59 7C 41 0E A1 77 02 4D E2 0F C8 19
 . Press ENTER to program public command key in SE OTP or press SPACE to skip.
```

8. Press `SPACE` to skip the programming of the Public Command Key. Optional to press `ENTER` to program the hard-coded Public Command Key to SE OTP.

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
. Get public command key... SL_STATUS_FAIL (cycles: 4126 time: 108 us)
. Press ENTER to initialize SE OTP for secure boot configuration or press SPACE to skip.
```

9. Press `ENTER` to program the secure boot configuration.

```
+ Warning: The SE OTP configuration cannot be changed once written!
+ Press ENTER to confirm or press SPACE to skip if you are not sure.
```

10. Press `ENTER` to confirm the operation. The SE returns `SL_STATUS_COMMAND_IS_INVALID` if an invalid setting from step 2 or the secure boot configuration has been programmed in SE OTP.

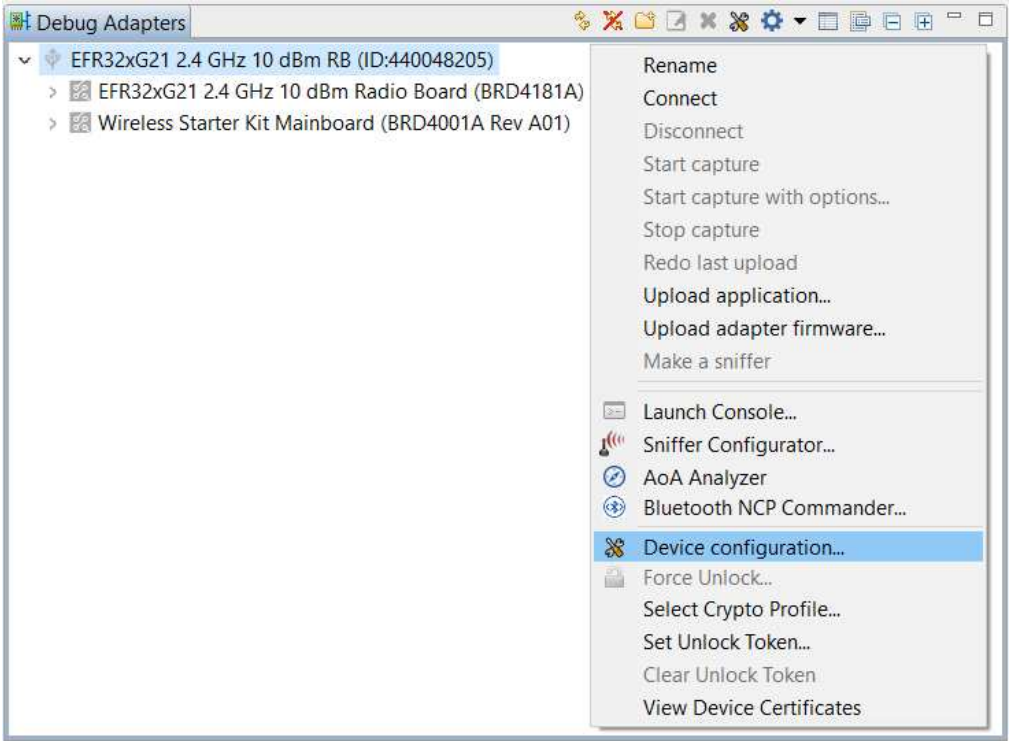
```
. Initialize SE OTP... SL_STATUS_OK (cycles: 267256 time: 7033 us)
+ Read SE OTP configuration... SL_STATUS_OK (cycles: 6865 time: 180 us)
+ Secure boot                 : Enabled
+ Secure boot verify certificate : Disabled
+ Secure boot anti-rollback    : Enabled
+ Secure boot page lock narrow  : Disabled
+ Secure boot page lock full    : Disabled
. SE manager deinitialization... SL_STATUS_OK (cycles: 5 time: 0 us)
```

```
. Initialize SE OTP... SL_STATUS_COMMAND_IS_INVALID (cycles: 3989 time: 104 us)
. SE manager deinitialization... SL_STATUS_OK (cycles: 5 time: 0 us)
```

Simplicity Studio

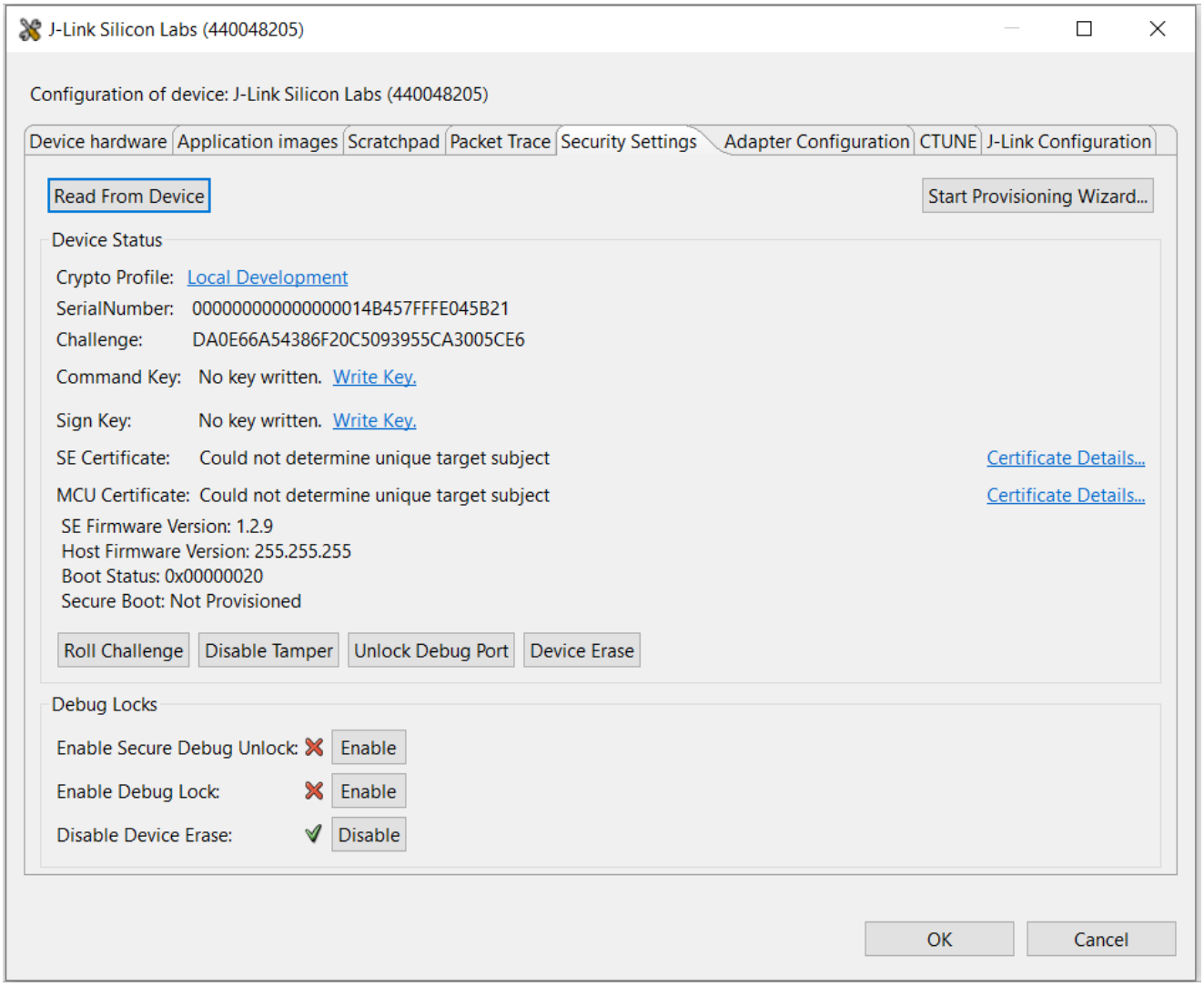
The security operations are performed in the Security Settings of Simplicity Studio. This application note uses Simplicity Studio v5.2.3.1. The procedures and pictures may be different for the other versions of Simplicity Studio 5.

1. Right-click the selected debug adapter **RB (ID:J-Link serial number)** to display the context menu.



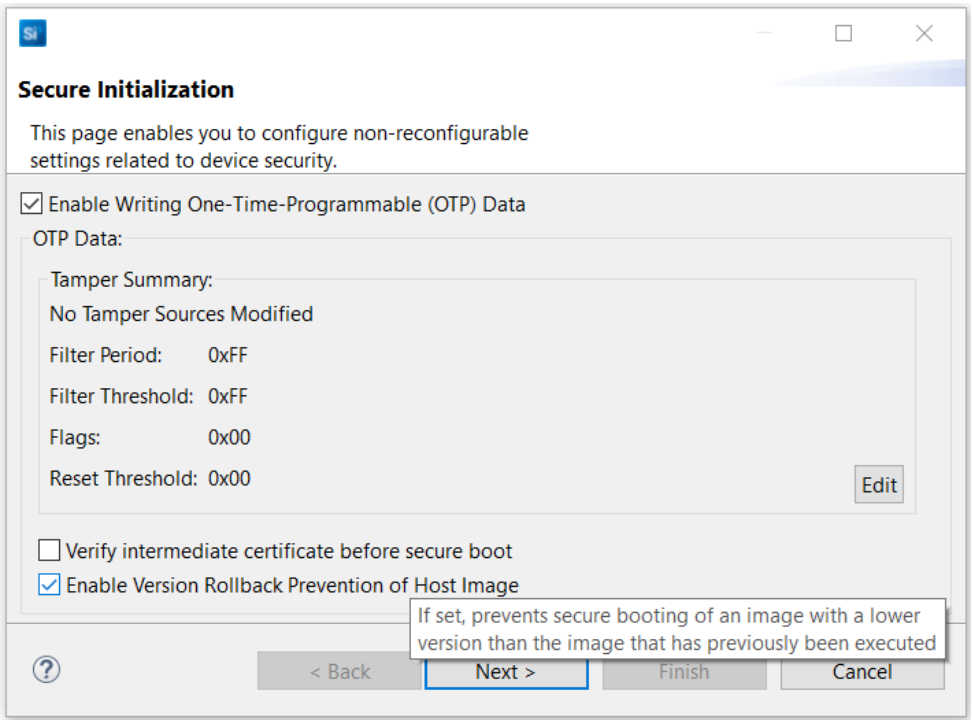
2. Click **Device configuration...** to open the **Configuration of device: J-Link Silicon Labs (serial number)** dialog box. Click the **Security Settings** tab to get the selected device configuration.

Copyright © 2025 Silicon Laboratories. All rights reserved.



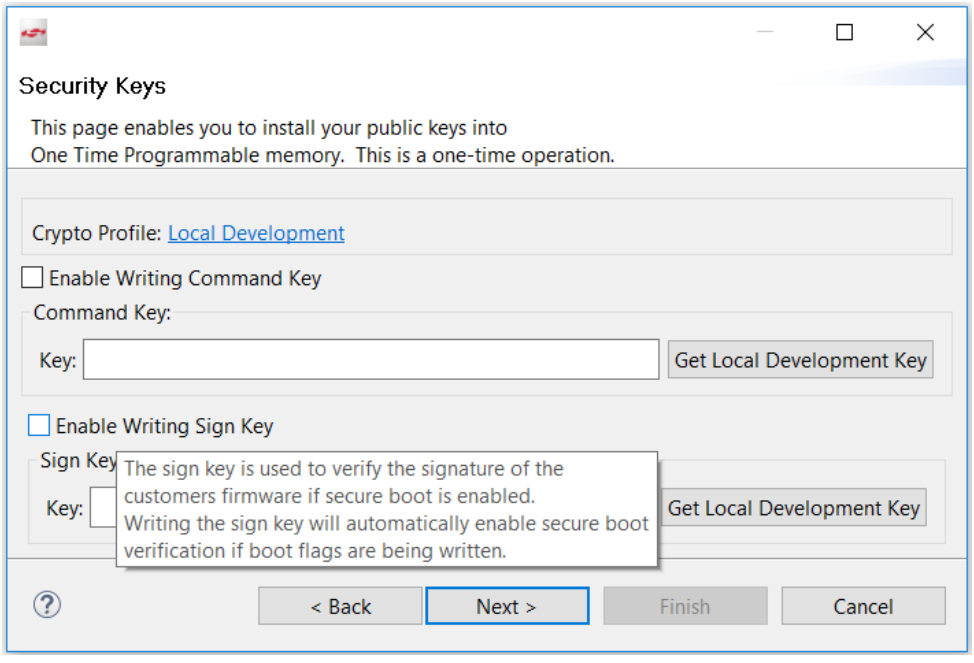
3. Click **[Start Provisioning Wizard...]** in the upper right corner to display the **Secure Initialization** dialog box. Checking the **Enable Version Rollback Prevention of Host Image** option is recommended. The **Verify intermediate certificate before secure boot** option is for Certificate-based Secure Boot.

Copyright © 2025 Silicon Laboratories. All rights reserved.



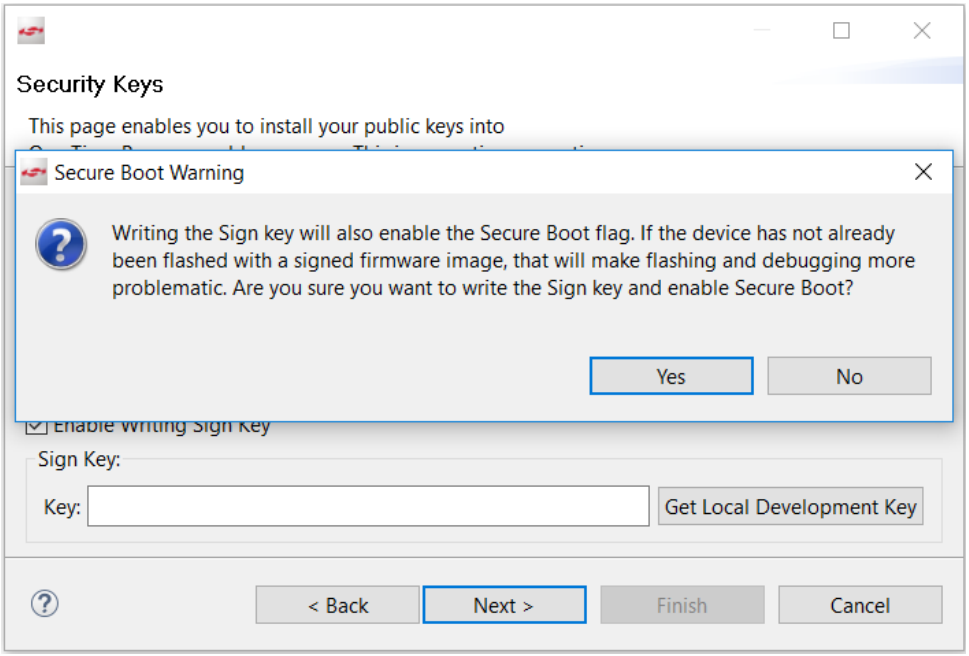
> **Note:** The SECURE_BOOT_PAGE_LOCK_NARROW and SECURE_BOOT_PAGE_LOCK_FULL options are not yet available in Simplicity Studio.

4. Click **[Next >]**. The **Security Keys** dialog box is displayed.



5. Checking the **Enable Writing Sign Key** checkbox will automatically enable Secure Boot. The following **Secure Boot Warning** is displayed. Click **[Yes]** to confirm.

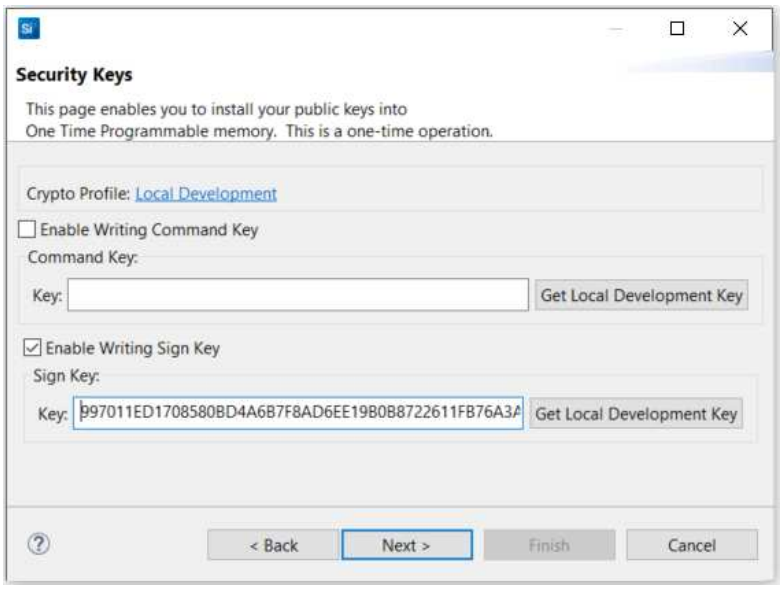
Copyright © 2025 Silicon Laboratories. All rights reserved.



6. Open the Public Sign Key token file (`sign_pubkey.txt`).

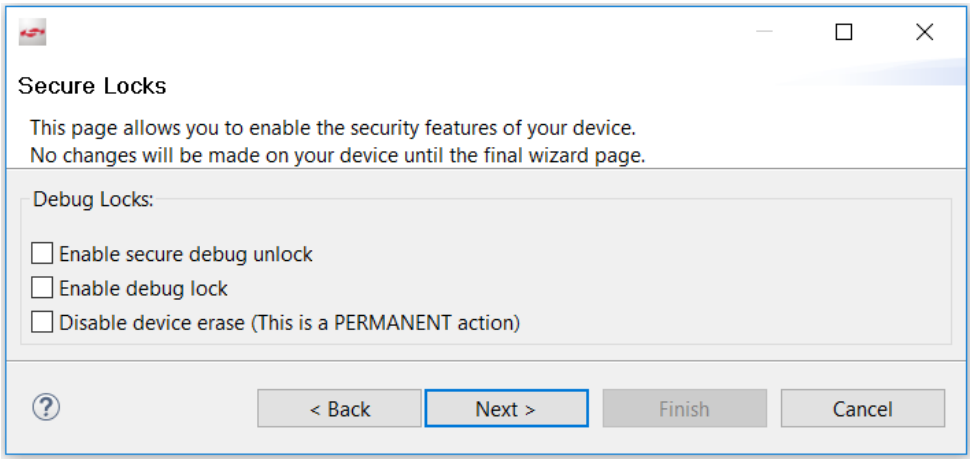
```
MFG_SIGNED_BOOTLOADER_KEY_X : 997011ED1708580BD4A6B7F8AD6EE19B0B8722611FB76A3A5702D5141180E101
MFG_SIGNED_BOOTLOADER_KEY_Y : 0AC8673C8ACC26EE2B534C004F4A4B7EBBC23D04506DD66E3EF0DDC81E3CA55E
```

7. Copy Public Sign Key (X-point `9970...` first, then Y-point `0AC8...`) to **Key:** box under **Sign Key:**.

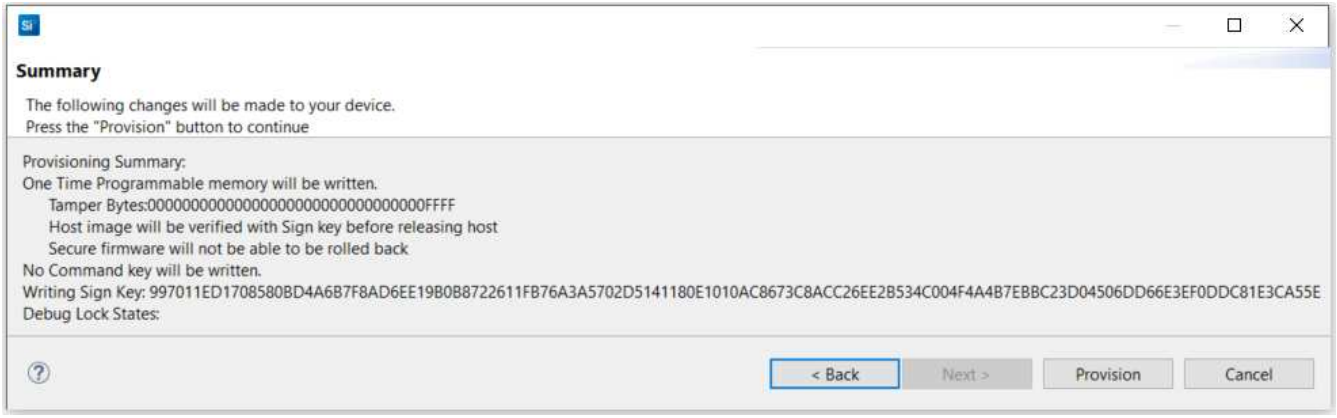


8. Click **[Next >]**. The **Secure Locks** dialog box is displayed. When Secure Boot is enabled, the **Debug Locks** are not set by default. Refer to Series 2 Secure Debug for more information about these locks.

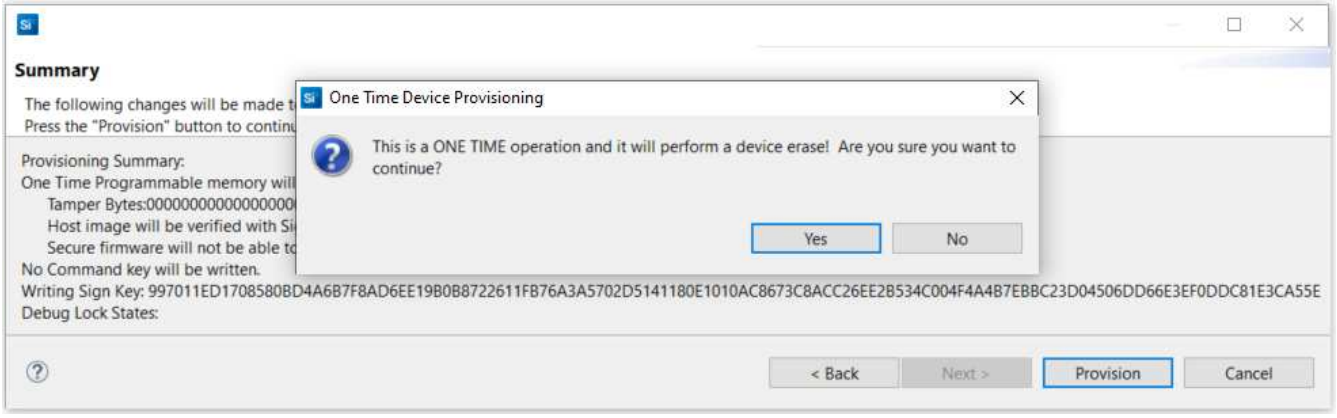
Copyright © 2025 Silicon Laboratories. All rights reserved.



9. Click **[Next >]** to display the **Summary** dialog box.

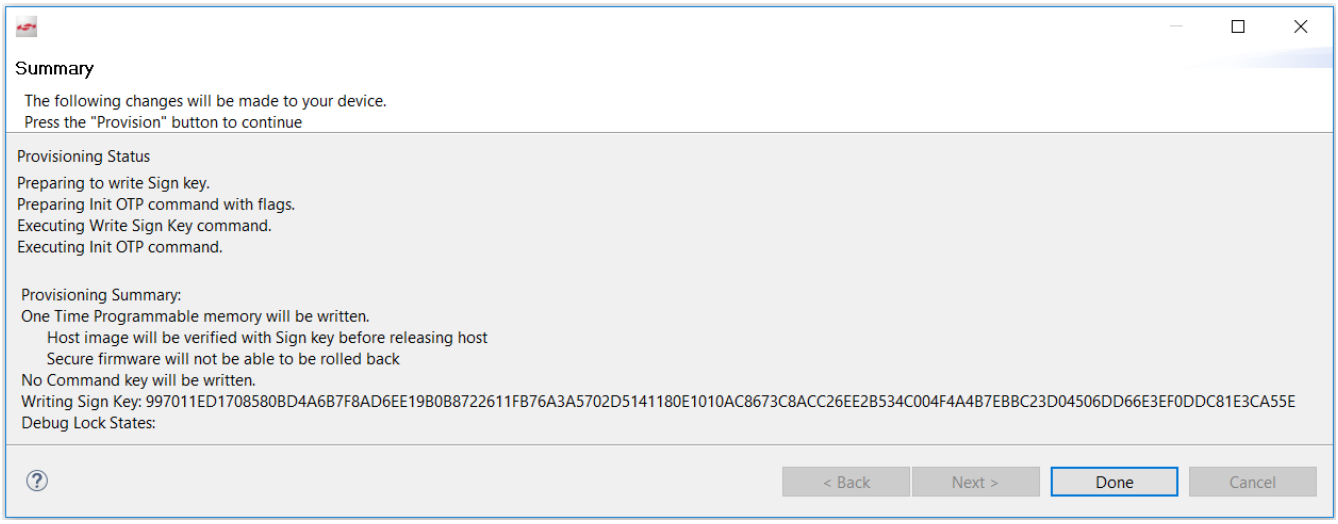


10. If the information displayed is correct, click **[Provision]**. Click **[Yes]** to confirm. The Public Sign Key and Secure Boot configuration cannot be changed once written.



11. The **Summary** dialog box displays the **Provisioning Status**.

Copyright © 2025 Silicon Laboratories. All rights reserved.



12. Click **[Done]** to exit the provisioning process. The device configuration is updated.

Provision GBL Decryption Key

The GBL Decryption Key is used to decrypt the GBL upgrade image file payloads during firmware upgrade. The user should provision this key before enabling the **Require encrypted firmware upgrade files** option in AppBuilder or Bootloader-core Software Component.

The following figure and table describe two (VSE) or three (HSE) options to select which GBL Decryption Key for GBL upgrade image file (bootloader, SE, or application) decryption.





Option for GBL Decryption Key Selection	GBL Decryption Key Storage
1. Use symmetric key stored in Secure Element storage (HSE devices only and GSDK ≥ v3.0).	The 128-bit symmetric key stored in HSE OTP is used for GBL upgrade image file decryption.
2. Use symmetric key stored in Application Properties Struct (GSDK ≥ v4.1).	The 128-bit symmetric key stored in the GBL Application Properties Struct is used for GBL upgrade image file decryption. The key is stored in the Secure flash if TrustZone is implemented.
3. Default storage if none of the above options are selected.	The 128-bit symmetric key stored on the top page of the main flash is used for GBL upgrade image file decryption. The key is stored in the Non-secure flash if TrustZone is implemented.

Note:

- Option 2 requires `ApplicationProperties_t` struct v1.2 or higher (defined in `application_properties.h` in the Windows folder `C:\Users\<PC USER NAME>\SimplicityStudio\SDKs\gecko_sdk\platform\bootloader\api`) in GSDK v4.1 or higher to store the GBL Decryption Key.

```
/// Major version number of the AppliationProperties_t struct
#define APPLICATION_PROPERTIES_VERSION_MAJOR (1UL)
/// Minor version number of the AppliationProperties_t struct
#define APPLICATION_PROPERTIES_VERSION_MINOR (2UL)
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
/// Application Properties struct
typedef struct {
  /// @brief Magic value indicating this is an ApplicationProperties_t struct.
  /// Must equal @ref APPLICATION_PROPERTIES_MAGIC
  uint8_t magic[16];
  /// Version number of this struct
  uint32_t structVersion;
  /// Type of signature this application is signed with
  uint32_t signatureType;
  /// Location of the signature. Typically points to the end of the application
  uint32_t signatureLocation;
  /// Information about the application
  ApplicationData_t app;
  /// Pointer to information about the certificate
  ApplicationCertificate_t *cert;
  /// Pointer to Long Token Data Section
  uint8_t *longTokenSectionAddress;
  /// Parser Decryption Key
  const uint8_t decryptKey[16];
} ApplicationProperties_t;
```

- Option 2 must be implemented before signing the GBL image for ECDSA-P256-SHA256 or certificate-basedSecure Boot.
- The options for the GBL Decryption Key are mutually exclusive. Either one of the two (VSE) or three (HSE) key storages will be selected for decryption.
- From a security point of view, it is highly recommended to use or upgrade to option 1 for HSE devices and option 2 for VSE devices.
- If the GBL Decryption Key in the selected option is compromised, the simple way is to upgrade the GBL to option 2 (if the existing option is 1 or 3) with the new GBL Decryption Key.

Simplicity Commander

The following procedures describe how to program the GBL Decryption Key for the options below. All procedures assume the required files are in the same folder.

- Use symmetric key stored in Secure Element storage (HSE devices only and GSDK ≥ v3.0)
- Use symmetric key stored in Application Properties Struct (GSDK ≥ v4.1)
- Default Storage on the Top Page of the Main Flash

1. Generate a 128-bit Symmetric Key
   Run the `util genkey` to generate the token file for the GBL Decryption Key.

   ```
   commander util genkey --type aes-ccm --outfile aes_key.txt
   ```

   ```
   Using Windows' Cryptographic random number generator
   DONE
   ```

   The `aes_key.txt` contains the randomly generated AES-128 key. Use the text editor to replace the randomly generated key in `aes_key.txt` with the desired GBL Decryption Key as below.

   ```
   # Key randomly generated by 'util genkey'
   TOKEN_MFG_SECURE_BOOTLOADER_KEY: 81A5E21FA15286F1DF445C2CC120FA3F
   ```

2. (**Use symmetric key stored in Secure Element storage**) Run the `security writekey` to provision the GBL Decryption Key. The GBL Decryption Key cannot be changed once written.

   ```
   commander security writekey --decrypt aes_key.txt --device EFR32MG21A010F1024 --serialno 440030580
   ```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
Device has serial number 000000000000000000d6ffffead3d94
================================================================================
Please look through any warnings before proceeding.
THIS IS A ONE-TIME command, any encrypting of GBL files must be done with this key.
Type 'continue' and hit enter to proceed or Ctrl-C to abort:
================================================================================
continue
DONE
```

> **Note:** It cannot read back the GBL Decryption Key from the HSE OTP.

3. (**Use symmetric key stored in Application Properties Struct**) Run the `convert` command to program the GBL Decryption Key to the `Application Properties Struct` of the GBL.

```
commander convert bootloader-uart-xmodem.s37 --aeskey aes_key.txt --outfile bootloader-uart-xmodem.s37
```

```
Parsing file bootloader-uart-xmodem.s37...
Writing to bootloader-uart-xmodem.s37...
Overwriting file: bootloader-uart-xmodem.s37...
DONE
```

> **Note:**
>
> - The `--aeskey` option in the convert command requires **Simplicity Commander v1.12.3 or above**.
> - The GBL Decryption Key can only be added to the GBL with `Application Properties Struct` v1.2 or higher.

4. (**Default Storage on the Top Page of the Main Flash**) Run the `flash` command to program the GBL Decryption Key in the token file to the top page of the main flash.

```
commander flash --tokengroup znet --tokenfile aes_key.txt --device EFR32MG21A010F1024 --serialno 440030580
```

```
Writing 8192 bytes starting at address 0x000fe000
Comparing range 0x000FE000 - 0x000FFFFF (8 KB)
Programming range 0x000FE000 - 0x000FFFFF (8 KB)
DONE
```

> **Note:** The MCU Series 2VSE devices (like EFM32PG22C200F512IM40) require Simplicity Commander Version 1.12.2 or above to support the `flash --tokengroup` znet command.

SE Manager Key Provisioning Platform Example

This example only applies to option 1 for HSE devices. Click the `View Project Documentation` link to open the `readme` file.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Platform - SE Manager Key Provisioning

This example project demonstrates the key provisioning API of SE Manager.

View Project Documentation

CREATE

1. Modify the default GBL Decryption Key in `aes_key[16]` array in `app_process.c` to the desired values.

```
/// 128-bit AES key
SL_ALIGN(4) static const uint8_t aes_key[16] = {
  0x81, 0xa5, 0xe2, 0x1f, 0xa1, 0x52, 0x86, 0xf1,
  0xdf, 0x44, 0x5c, 0x2c, 0xc1, 0x20, 0xfa, 0x3f
};
```

2. Modify the `ciphertext[16]` array in `app_process.c` to the expected ciphertext for AES ECB on 16 bytes zero plaintext to verify the GBL Decryption Key in step 1.

```
/// Ciphertext to verify 128-bit AES key
static const uint8_t ciphertext[16] = {
  0x66, 0xd2, 0x0f, 0x99, 0x65, 0x3e, 0xa8, 0xd0,
  0x83, 0x05, 0xa6, 0x39, 0xd4, 0x4e, 0x98, 0xa6
};
```

3. Follow the procedures in Generate an Unsigned Application Image to generate the unsigned application image if the GBL is present in the device.
4. Build the project and run the application. Follow the procedures in Signing for ECDSA-P256-SHA256 Secure Boot or Signing for Certificate-Based Secure Boot if a signed application image is required.
5. Then press `ENTER` to program the hard-coded GBL Decryption Key to HSE OTP.

```
SE Manager Key Provisioning Example - Core running at 38000 kHz.
  . SE manager initialization... SL_STATUS_OK (cycles: 9 time: 0 us)
  . Get current SE firmware version... SL_STATUS_OK (cycles: 3578 time: 94 us)
  + Current SE firmware version (MSB..LSB): 00010209
  . Read SE OTP configuration... SL_STATUS_COMMAND_IS_INVALID (cycles: 3908 time: 102 us)
  . Press ENTER to program 128-bit AES key in SE OTP or press SPACE to skip.
  + Warning: The 128-bit AES key in SE OTP cannot be changed once written!
  + Press ENTER to confirm or press SPACE to skip if you are not sure.
```

6. Press `ENTER` to confirm the operation. The program either returns `SL_STATUS_OK` or `SL_STATUS_INVALID_PARAMETER` (already present) and performs AES ECB encryption to verify the GBL Decryption Key in HSE OTP.

```
  . Initialize 128-bit AES key... SL_STATUS_OK (cycles: 39059 time: 1027 us)
  . Encrypt 16 bytes plaintext with 128-bit AES OTP key... SL_STATUS_OK (cycles: 11013 time: 289 us)
  + Compare encrypted message with expected ciphertext... OK
  . Press ENTER to program public sign key in SE OTP or press SPACE to skip.
```

```
  . Initialize 128-bit AES key... SL_STATUS_INVALID_PARAMETER (cycles: 4474 time: 117 us)
  . Encrypt 16 bytes plaintext with 128-bit AES OTP key... SL_STATUS_OK (cycles: 11001 time: 289 us)
  + Compare encrypted message with expected ciphertext... OK
  . Press ENTER to program public sign key in SE OTP or press SPACE to skip.
```

7. Press `SPACE` to skip the programming of the Public Sign Key.

```
  . Get public sign key... SL_STATUS_FAIL (cycles: 4126 time: 108 us)
  . Press ENTER to program public command key in SE OTP or press SPACE to skip.
```

8. Press `SPACE` to skip the programming of the Public Command Key.

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
. Get public command key... SL_STATUS_FAIL (cycles: 4126 time: 108 us)
. Press ENTER to initialize SE OTP for secure boot configuration or press SPACE to skip.
```

9. Press `SPACE` to skip the programming of the secure boot configuration.

```
. SE manager deinitialization... SL_STATUS_OK (cycles: 10 time: 0 us)
```

Secure Boot

The user should usually not enable the Secure Boot during the development phase to avoid a clash on debugging. The Secure Boot feature is enabled near firmware release and uses the following sections to validate the configuration and system functionality.

Generate an Unsigned GBL Image

There are two ways to configure the application firmware through a GBL project.

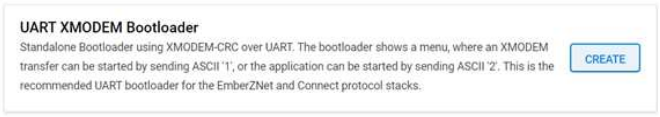
1. Use AppBuilder (`.iscfile`) in **GSDK v3.2 and lower**.
2. Use Bootloader-core software component (`.slcpfile`) in **GSDK v4.0 and higher**.

The following notes apply to the AppBuilder and Bootloader-core software component.

- Enabling the **Allow use of public key from manufacturing token storage** option is mandatory on VSE devices (cannot be disabled in AppBuilder and is discarded in Bootloader-core software component) for ECDSA-P256-SHA256 Secure Boot. The HSE device ignores this default enabled option if the Public Sign Key has been provisioned in OTP.
- The GBL stores the application version counter at the end of the bootloader flash space if **Enable application rollback protection** option (GSDK ≥ v3.0) is enabled. The GBL checks the application version during an upgrade and on every boot. The anti-rollback does not prevent flashing an older application image to the device.
- The **Minimum application version allowed** option (GSDK ≥ v3.0) configures the minimum application version that should be allowed to boot. This option maintains the application version counter that will reset to 0 after upgrading the GBL.
- The **Enable application rollback protection** option is not applicable if the SECURE_BOOT_PAGE_LOCK_FULL in SE OTP is enabled. See section "Secure Boot with Application Rollback Protection" in UG266 (for GSDK v3.2 and lower) or Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher (for GSDK v4.0 and higher) for details about the application rollback protection.
- The GBL size and starting address are device-dependent. For more information about the bootloader size and starting address on Series 2 devices, see section "Memory Space For Bootloading" in UG103.6: Bootloader Fundamentals.

Refer to Generate a GBL Upgrade Image File for the **Require signed firmware upgrade files** and **Require encrypted firmware upgrade files** options on the GBL upgrade image file. For simplicity, the Secure Boot examples in this application note do not enable these options. Refer to UG266/Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher for information about these options.

The following sections describe how to build the unsigned GBL image from the **UART XMODEM Bootloader** (GSDK < v4.1) or **Bootloader - NCP UART XMODEM** (GSDK ≥ v4.1).





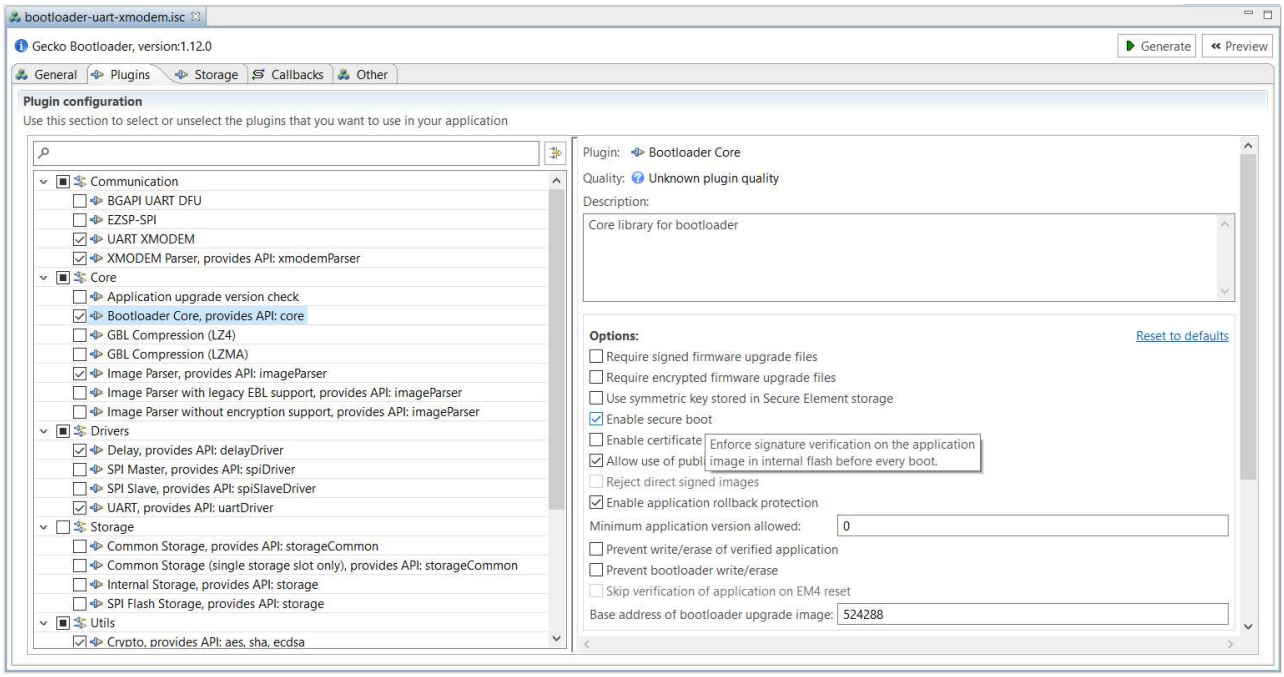
AppBuilder

This application note uses UART XMODEM Bootloader example v1.12.0 in GSDK v3.2.3. The procedures and pictures may be different for the other versions of this example.

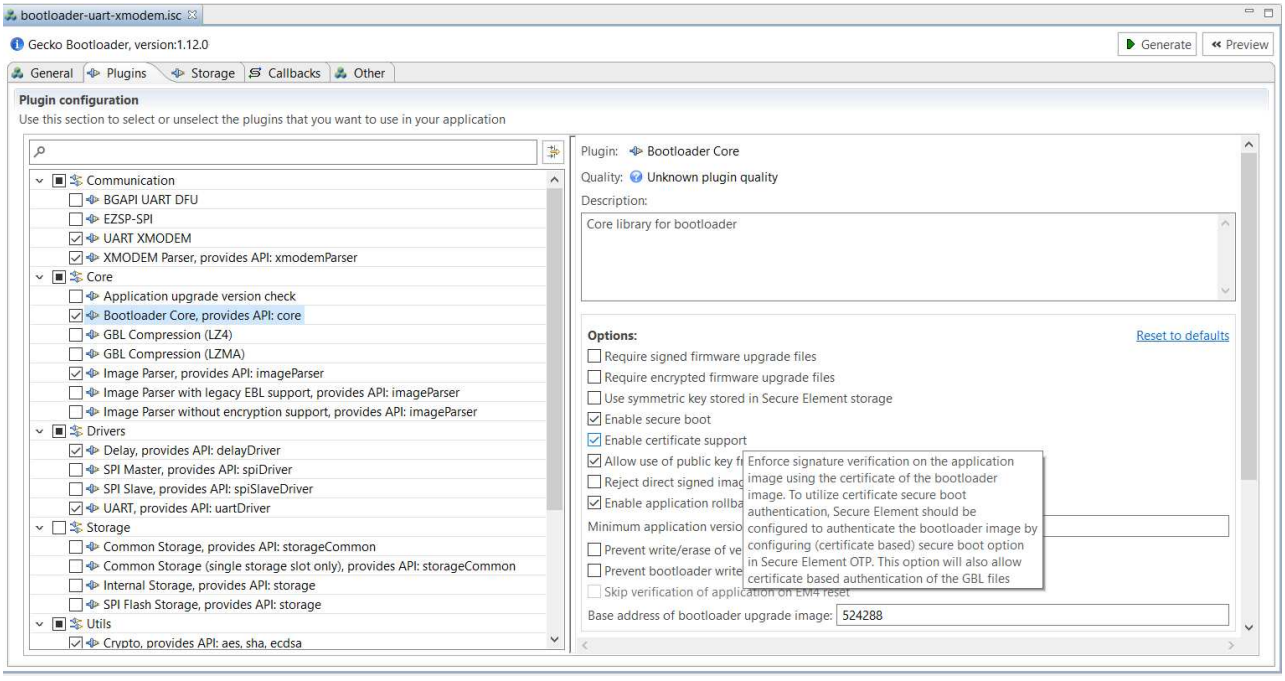
1. Create a UART XMODEM Bootloader project.

Copyright © 2025 Silicon Laboratories. All rights reserved.

2. The **Plugins** tab in AppBuilder (`bootloader-uart-xmodem.isc`) shows the default configurations for the UART XMODEM Bootloader example.
3. Use **Bootloader Core, provides API: core** in the **Plugins** tab to set up the application firmware configurations.
    1. This application note uses the configuration below for ECDSA-P256-SHA256 Secure Boot.

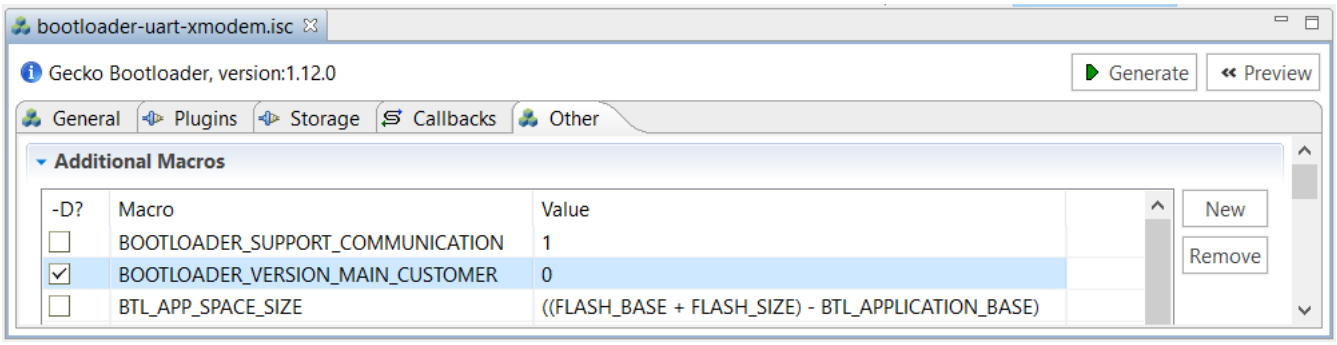


    2. This application note uses the configuration below for Certificate-based Secure Boot.

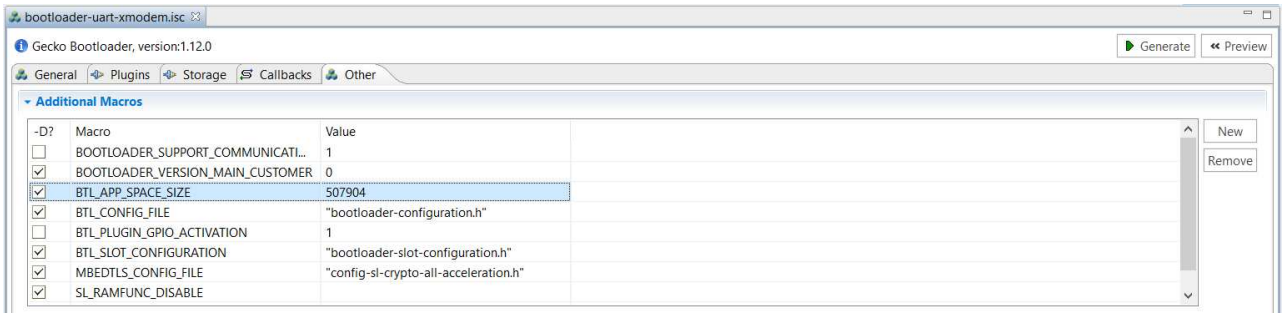


4. Enter a higher version number (default is 0) to the macro **BOOTLOADER_VERSION_MAIN_CUSTOMER** → **Value** in the **Additional Macros** field on the **Other** tab for anti-rollback protection of GBL.

Copyright © 2025 Silicon Laboratories. All rights reserved.



5. The default setting of GBL will overwrite the existing application image when upgrading the GBL or SE. It forces to update the application image even without changes on the firmware. Use the Appbuilder settings below to keep the existing application image when upgrading the GBL or SE.
   1. Enter the required application image size to the macro **BTL_APP_SPACE_SIZE** → **Value** in the **Additional Macros** field on the **Other** tab. Check the **-D?** checkbox to add this definition to the project.
      This application note uses `507904` (496 kB) to replace the default value of `((FLASH_BASE + FLASH_SIZE) - BTL_APPLICATION_BASE)`.



   2. The **Base address of bootloader upgrade image** ≥ (**BTL_APP_SPACE_SIZE** + size of the GBL).
      The example in this application note uses EFR32MG21A010F1024:
      `Base address of bootloader upgrade image` = `507904` (496 kB) + `16384` (16 kB for GBL) = `524288` (512 kB)



> **Note**: The default value of `Base address of bootloader upgrade image` is `32768` (32 kB).

   3. The (**Base address of bootloader upgrade image** + size of the GBL or SE + upgrade file overhead) ≤ the available size of the device main flash for application use (see project linker file for details).

The example in this application note uses EFR32MG21A010F1024:
512 kB (Base address of bootloader upgrade image) + 16 kB (GBL) or 48 kB (SE) + overhead bytes < 1024 kB (size of main flash)
For more information about the size of the GBL and SE, see section "Storage Space Size Configuration" in UG266.

> **Note**: It requires GBL v1.11.0 or above to support this feature.

Copyright © 2025 Silicon Laboratories. All rights reserved.

6. Click **[Generate]** in the right upper corner.
7. In the **Generation Successful** dialog, click **[OK]**.
8. Build the project to generate the unsigned GBL image file (`bootloader-uart-xmodem.s37`).
9. **(Optional)** Run the util appinfo command to check all available information (application properties) in an unsigned GBL image. The `App version` is the GBL version for the `SECURE_BOOT_ANTI_ROLLBACK` option.

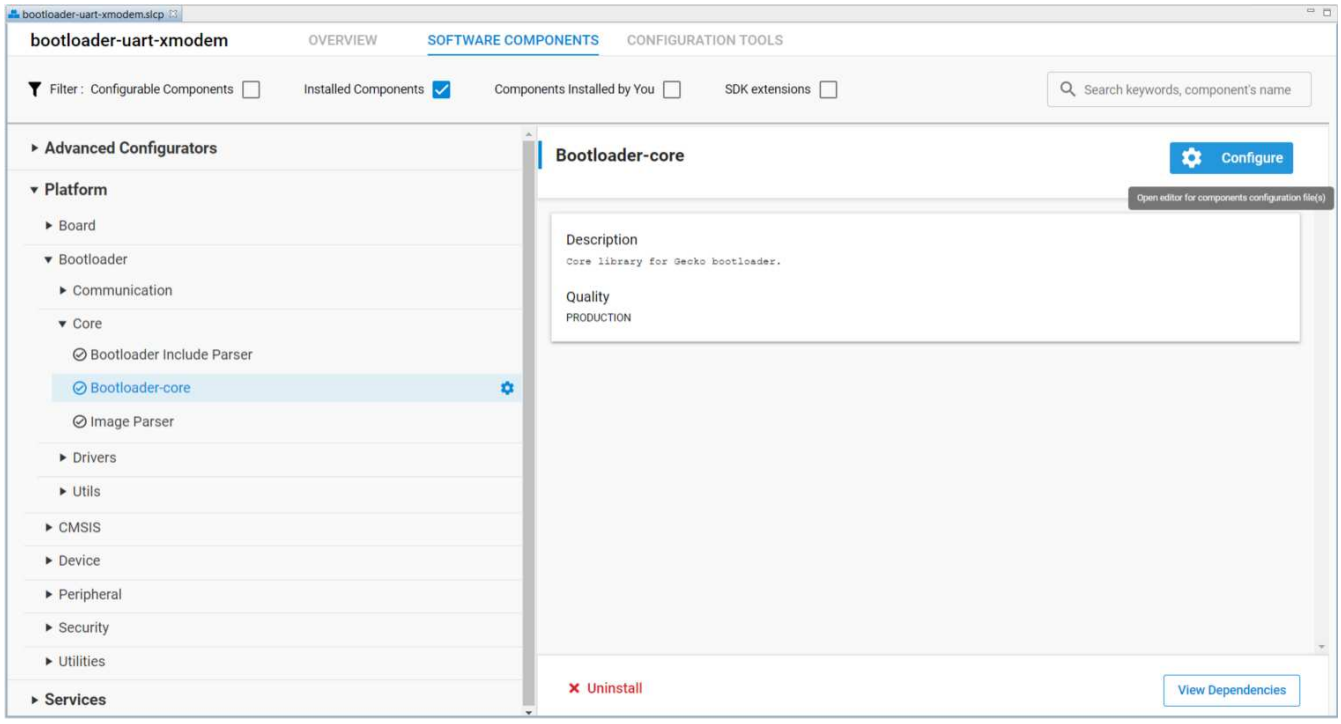
```
commander util appinfo bootloader-uart-xmodem.s37
```

```
Parsing file bootloader-uart-xmodem.s37...
Found application properties in image.
Application properties info:
Application properties location : 0x00002b1c
Signature location              : 0x00002d08Signature type              : No signature
Long token section address      : Not set (0x00000000)
Application data info:
For Series 2 devices: If rollback prevention is enabled in the OTP configuration, the device will not boot if the device has seen an application with a higher version number.
App type                        : Bootloader (APPLICATION_TYPE_BOOTLOADER)App version                : 0x010c0000
Product ID                      : Not set (0x00000000000000000000000000000000)
No certificate found in image.
For Series 2 devices: If the configuration flag SECURE_BOOT_VERIFY_CERTIFICATE is set or a device has previously seen certificate based signing, it will not accept direct signing.
DONE
```

Bootloader-core Software Component

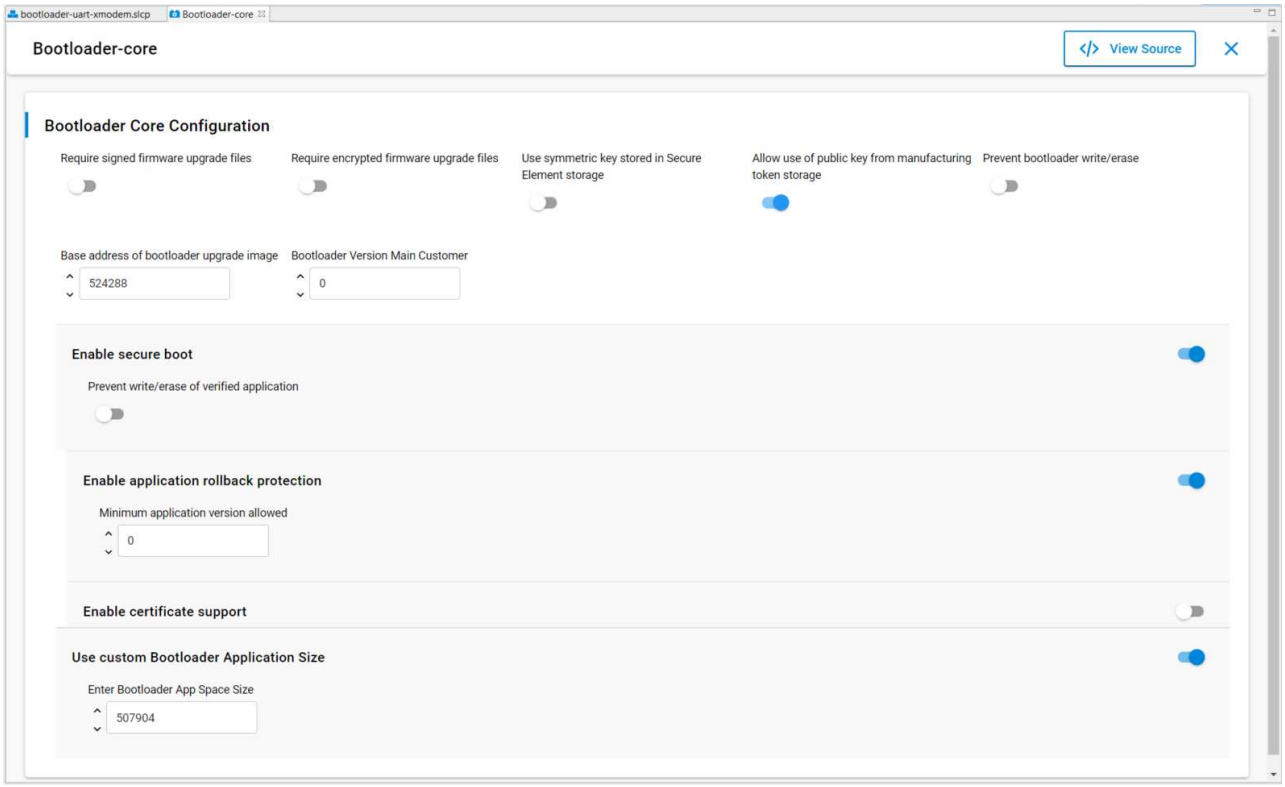
This application note uses UART XMODEM Bootloader example v2.0.0 in GSDK v4.0. The procedures and pictures may be different for the other versions of this example.

1. Create a UART XMODEM Bootloader project.
2. Checking the **Installed Components** under the **SOFTWARE COMPONENTS** tab shows the list of installed components (`bootloader-uart-xmodem.slcp`) in the UART XMODEM Bootloader example.
3. Click **[Configure]** in the **Bootloader-core** component to open the **Bootloader Core Configuration**.

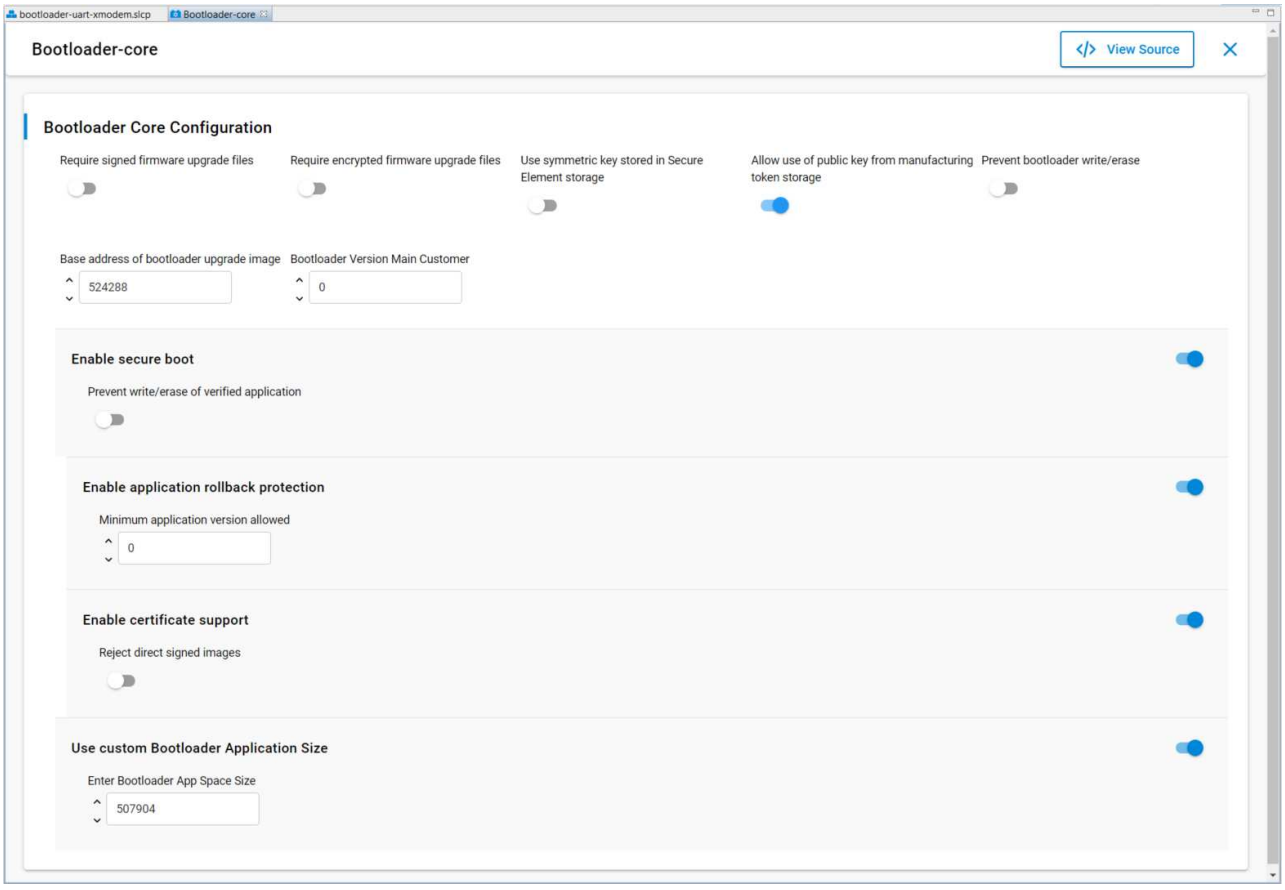


4. Use **Bootloader Core Configuration** in **Bootloader-core** to set up the application firmware configurations.
    1. This application note uses the configuration below for ECDSA-P256-SHA256 Secure Boot.

Copyright © 2025 Silicon Laboratories. All rights reserved.



2. This application note uses the configuration below for Certificate-based Secure Boot.

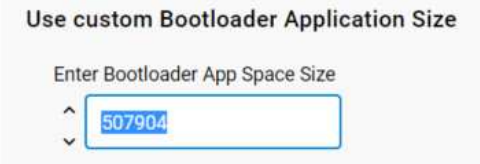


Copyright © 2025 Silicon Laboratories. All rights reserved.

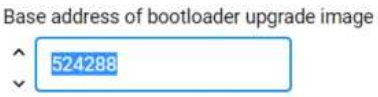
5. Enter a higher version number (default is 0) to **BOOTLOADER VERSION MAIN CUSTOMER** for anti-rollback protection of GBL.



6. The default setting of GBL will overwrite the existing application image when upgrading the GBL or SE. It forces to update the application image even without changes on the firmware. Use the Bootloader-core settings below to keep the existing application image when upgrading the GBL or SE.
   1. Enter the required application image size to the **Enter Bootloader App Space Size** dialog box after enabling the **Use custom Bootloader Application Size** option.
      This application note uses `507904` (496 kB) to replace the default value of `0`.



   2. The **Base address of bootloader upgrade image** ≥ (**Enter Bootloader App Space Size** + size of the GBL).
      The example in this application note uses EFR32MG21A010F1024:
      `Base address of bootloader upgrade image` = `507904` (496 kB) + `16384` (16 kB for GBL) = `524288` (512 kB)



> **Note**: The default value of `Base address of bootloader upgrade image` is `32768` (32 kB).

   3. The (**Base address of bootloader upgrade image** + size of the GBL or SE + upgrade file overhead) ≤ the available size of the device main flash for application use (see project linker file for details).

   The example in this application note uses EFR32MG21A010F1024:
   512 kB (Base address of bootloader upgrade image) + 16 kB (GBL) or 48 kB (SE) + overhead bytes < 1024 kB (size of main flash)
   For more information about the size of the GBL and SE, see section *Storage Space Size Configuration* in Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher.
7. Click **[X]** in the right upper corner to exit the **Bootloader Core Configuration**.
8. Build the project to generate the unsigned GBL image file (bootloader-uart-xmodem.s37).
9. **(Optional)** Run the convert command to program the GBL Decryption Key to the `Application Properties Struct` if this GBL Decryption Key option in GBL (GSDK ≥ v4.1) is selected.

```
commander convert bootloader-uart-xmodem.s37 --aeskey aes_key.txt --outfile bootloader-uart-xmodem.s37
```

```
Parsing file bootloader-uart-xmodem.s37...
Writing to bootloader-uart-xmodem.s37...
Overwriting file: bootloader-uart-xmodem.s37...
DONE
```

10. **(Optional)** Run the `util appinfo` command to check all available information (application properties) in an unsigned GBL image. The `App version` is the GBL version for the `SECURE_BOOT_ANTI_ROLLBACK` option.

```
commander util appinfo bootloader-uart-xmodem.s37
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
Parsing file bootloader-uart-xmodem.s37...
Found application properties in image.
Application properties info:
Application properties location : 0x00002b30
Signature location              : 0x00002c44Signature type              : No signature
Long token section address      : Not set (0x00000000)
Application data info:
For Series 2 devices: If rollback prevention is enabled in the OTP configuration, the device will not boot if the device has seen an application with a higher version number.
App type                        : Bootloader (APPLICATION_TYPE_BOOTLOADER)App version                : 0x02000000
Product ID                      : Not set (0x00000000000000000000000000000000)
No certificate found in image.
For Series 2 devices: If the configuration flag SECURE_BOOT_VERIFY_CERTIFICATE is set or a device has previously seen certificate based signing, it will not accept direct signing.
DONE
```

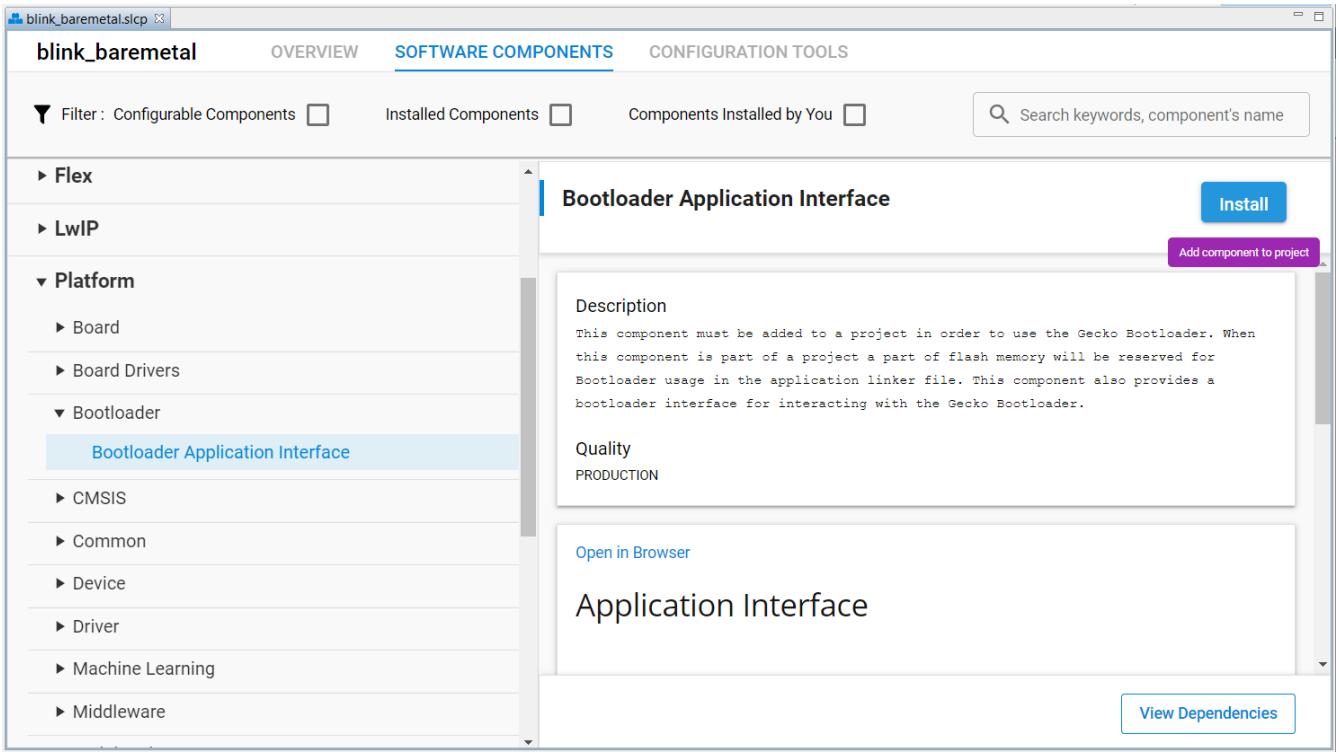
> **Note**: For the TrustZone-aware bootloaders, the unsigned GBL image is the combined image of Secure and Non-secure bootloaders. The **Bootloader-core** component is installed in the Secure bootloader.

Generate an Unsigned Application Image

This section describes how to generate an unsigned application image for the GBL.

1. For Series 2 devices, the application image should place on the main flash page after the GBL. For more information about the application starting address, see section "Memory Space For Bootloading" in UG103.6: Bootloader Fundamentals.
2. **(Simplicity Studio 4)** Refer to the section "Creating Applications for Use with the Bootloader" in AN0042: USB/UART Bootloader or the Knowledge Article (Simplicity IDE) in Silicon Labs Community to set up the application image start address.
3. **(Simplicity Studio 5)** The user can use the **Bootloader Application Interface** component to set up the start address of the application image. This application note uses **Platform - Blink Bare-metal** example in GSDK v3.2.3. The procedures and pictures may be different on other versions of the GSDK. The following steps can apply to other platform examples in GSDK.
    1. Create a Platform - Blink Bare-metal project.
    2. The **Software Components** tab shows the list of available components (`blink_bardmetal.slcp`) that the user can install in the project.
    3. Select **Platform** → **Bootloader** → **Bootloader Application Interface**.
    4. Click **[Install]**.

Copyright © 2025 Silicon Laboratories. All rights reserved.



> **Note**: For the wireless protocol stack example, the Bootloader Application Interface component is already present in the project.

4. The application image should contain an `ApplicationProperties_t` struct (defined in `application_properties.h` in the Windows folder below) declaring the application version, capabilities, and other metadata.
   For GSDK v3.2 and lower: `C:\SiliconLabs\SimplicityStudio\v5\developer\sdks\gecko_sdk_suite\<GSDK VERSION>\platform\bootloader\api`
   For GSDK v4.0 and higher: `C:\Users\<PC USER NAME>\SimplicityStudio\SDKs\gecko_sdk\platform\bootloader\api`
   Below is an example source file `app_properties.c` with `ApplicationProperties_t` struct for Secure Boot on GSDK v3.2 and lower.

```
#include <stddef.h>
#include "application_properties.h"
const ApplicationProperties_t sl_app_properties = {
.magic = APPLICATION_PROPERTIES_MAGIC,
.structVersion = APPLICATION_PROPERTIES_VERSION,
.signatureType = APPLICATION_SIGNATURE_NONE,.signatureLocation = 0,
.app = {
  .type = APPLICATION_TYPE_MCU,
  .version = 1UL,
  .capabilities = 0UL,
  .productId = {0U},
},
};
```

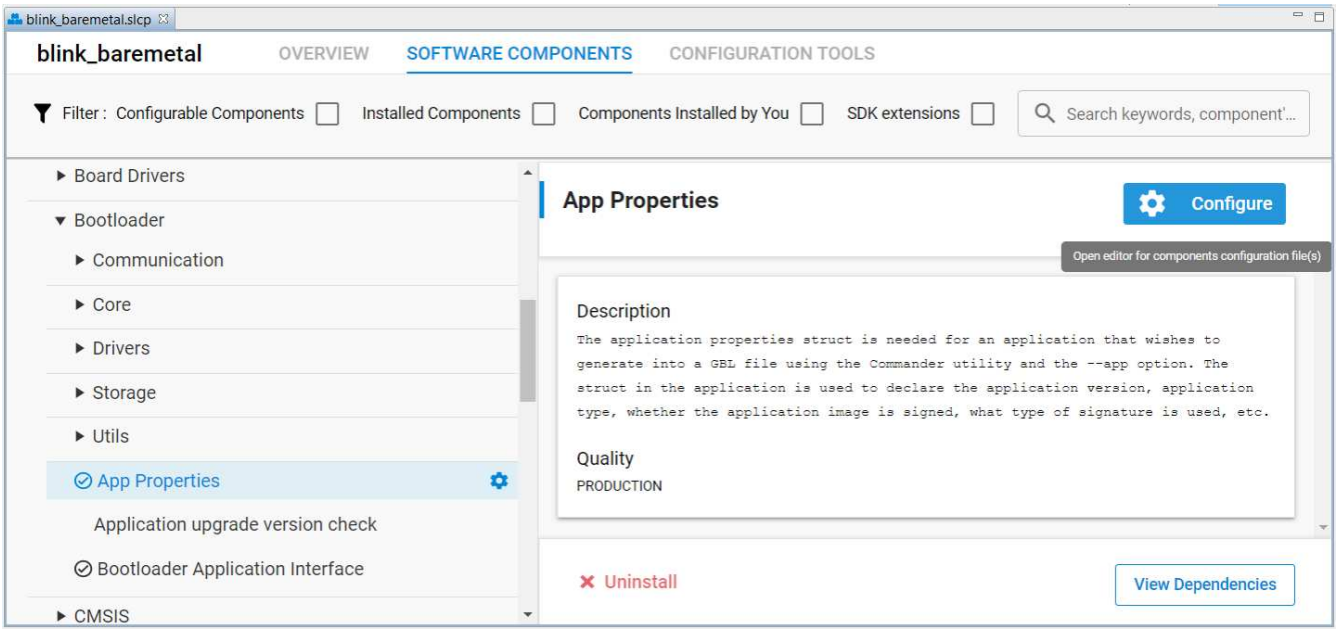
   The `signatureType` and `signatureLocation` are filled by Simplicity Commander when signing the application image using the convertcommand.
5. The following table describes how to add the `app_properties.c` file in step 4 to Platform - Blink Bare-metal project. For the wireless protocol stack example, the `app_properties.c` file with `ApplicationProperties_t` struct is already present in the project.

Copyright © 2025 Silicon Laboratories. All rights reserved.

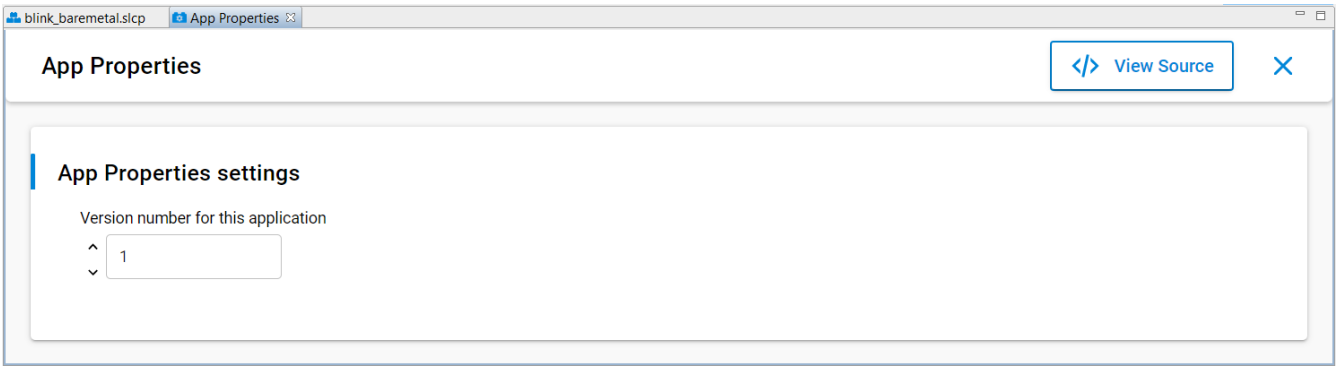
Simplicity Studio 4 & Simplicity Studio 5 with GSDK v3.2 and lower	Simplicity Studio 5 with GSDK v4.0 and higher
Manually added	Automatically added after installing the Bootloader Application Interface component in step 3 to the project.

> **Note**: Refer to the Knowledge Article in Silicon Labs Community to add `app_properties.c` to the project in Simplicity Studio 4.

6. **(Simplicity Studio 4 & Simplicity Studio 5 with GSDK v3.2 and lower)** Enter a higher version number to `.version` in `app_properties.c` for anti-rollback protection (if enabled) of the application.
7. **(Simplicity Studio 5 with GSDK v4.0 and higher)** Click **[Configure]** in the **App Properties** component under **Platform → Bootloader** to open the **App Properties** configuration. The example below uses GSDK v4.0. The procedures and pictures may be different on other versions of the GSDK.



Enter a higher version number to **Version number for this application** dialog box in **App Properties settings** for anti-rollback protection (if enabled) of the application.



> **Note**: The `app_properties.c` is in the Windows folder below.

Copyright © 2025 Silicon Laboratories. All rights reserved.

`C:\Users\<PC USER NAME>\SimplicityStudio\SDKs\gecko_sdk\platform\bootloader\app_properties`

8. Build the project to generate the unsigned application image file (`blink_bardmetal.s37`).
9. **(Optional)** Run the `util appinfocommand` to check all available information about `ApplicationProperties_t` struct in an unsigned application image. The `App version` is for the **Enable application rollback protection** option in the AppBuilder or Bootloader-core software component.

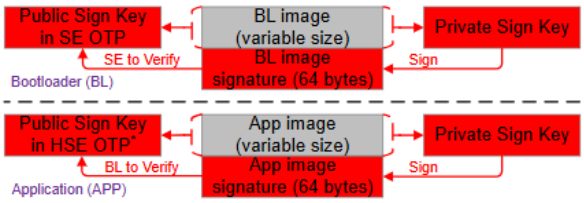
```
commander util appinfo blink_baremetal.s37
```

```
Parsing file blink_baremetal.s37...
Found application properties in image.
Application properties info:
Application properties location : 0x00006198
Signature location              : Not set (0x00000000) Signature type              : No signature
Long token section address      : Not set (0x00000000)
Application data info:
For Series 2 devices: If rollback prevention is enabled in the OTP configuration, the device will not boot if the device has seen an application with a higher version number.
App type                        : MCU application (APPLICATION_TYPE_MCU) App version               : 0x00000001
Product ID                      : Not set (0x00000000000000000000000000000000)
No certificate found in image.
For Series 2 devices: If the configuration flag SECURE_BOOT_VERIFY_CERTIFICATE is set or a device has previously seen certificate based signing, it will not accept direct signing.
DONE
```

> **Note:** For the TrustZone-aware applications, the unsigned application image is the combined image of Secure and Non-secure applications. The `ApplicationProperties_t` struct is located in the Secure application.

Signing for ECDSA-P256-SHA256 Secure Boot

The following figure describes the signing and verification for ECDSA-P256-SHA256 Secure Boot.



Note:

- The bootloader cannot access the Public Sign Key in VSE OTP to verify the application image. Therefore VSE devices need to store a Public Sign Key copy on the top page of the main flash (see section *Key Storage* in UG266/Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher).

Device	FSB to Verify the Bootloader Image	SSB to Verify the Application Image
HSE	Use the Public Sign Key in HSE OTP	Use the Public Sign Key in HSE OTP
VSE	Use the Public Sign Key in VSE OTP	Use the Public Sign Key on the top page of the main flash

The HSE device ignores the default enabled Allow use of public key from manufacturing token storage option once the Public Sign Key has been provisioned.

- To have better protection on the Public Sign Key. The certificate-basedSecure Boot is strongly recommended on VSE devices since the SSB does not require accessing the Public Sign Key to verify the application signature.

Copyright © 2025 Silicon Laboratories. All rights reserved.

The following sections provide two methods to sign the bootloader image and application image files. All procedures assume the required files are in the same folder.

1. Using Simplicity Commander
2. Using an HSM and Simplicity Commander

Bootloader Image File

1. If the SE OTP is not provisioned, follow the procedures in Simplicity Commander to set up the ECDSA-P256-SHA256 Secure Boot configuration for the bootloader.
2. Follow the procedures in AppBuilderor Bootloader-core Software Component to set up the ECDSA-P256-SHA256 Secure Boot configuration for the user application to generate an unsigned bootloader image.
3. **(Using Simplicity Commander)** Run the `convert` command with **Private Sign Key** to overwrite the unsigned bootloader image file with the signed bootloader image file (`bootloader-uart-xmodem.s37`).

```
commander convert bootloader-uart-xmodem.s37 --secureboot --keyfile sign_key.pem --verify sign_pubkey.pem
--outfile bootloader-uart-xmodem.s37
```

```
Parsing file bootloader-uart-xmodem.s37...
Found Application Properties at 0x00002b1c
Writing Application Properties signature pointer to point to 0x00002d08
Setting signature type in Application Properties: 0x00000001
Image SHA256: c53bb8a3fd88a5071bfb71444324bb136b276160318488ff89011bbd269e114e
R = AB62F3A52B13D137FBCC6A2176D4D1852E06B6E4E6B2673DC251FC491450CBDA
S = 9C7C7AF2624165FD90FB3B114E3FA6FE4F4C5625B15C9F3D50DCB04DD06A7B19
Verifying signed image...
Writing to bootloader-uart-xmodem.s37...
Overwriting file: bootloader-uart-xmodem.s37...
DONE
```

4. **(Using an HSM and Simplicity Commander)** Run the `convert` command with **--extsign** option to generate an external signing bootloader image file (`bootloader-uart-xmodem.extsign`).

```
commander convert bootloader-uart-xmodem.s37 --secureboot --extsign --outfile bootloader-uart-xmodem
```

```
Parsing file bootloader-uart-xmodem.s37...
Found Application Properties at 0x00002b1c
Writing Application Properties signature pointer to point to 0x00002d08
Setting signature type in Application Properties: 0x00000001
Writing to bootloader-uart-xmodem.extsign...
DONE
```

Use an HSM containing the Private Sign Key to generate the signature for the external signing bootloader image. This example uses the OpenSSL with the **Private Sign Key** to simulate this process. The signature is in the `bl_signature.der` .

```
openssl dgst -sha256 -binary -sign sign_key.pem -out bl_signature.der bootloader-uart-xmodem.extsign
```

Run the `convert` command with the **bootloader image signature** to overwrite the unsigned bootloader image file with the signed bootloader image file (`bootloader-uart-xmodem.s37`).

```
commander convert bootloader-uart-xmodem.s37 --secureboot --signature bl_signature.der
--verify sign_pubkey.pem --outfile bootloader-uart-xmodem.s37
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
Parsing file bootloader-uart-xmodem.s37...
Parsing signature file bl_signature.der...
R = 0E9FC64F41B55367894908D3ADAC40E8D145E33224C4BAA8151EC3EFD107A154
S = F56230AA6484E55270F22A4D164377CA918F66A367656AB6E10CB3F58641CE84
Found Application Properties at 0x00002b1c
Writing Application Properties signature pointer to point to 0x00002d08
Setting signature type in Application Properties: 0x00000001
Verifying signed image...
Writing to bootloader-uart-xmodem.s37...
Overwriting file: bootloader-uart-xmodem.s37...
DONE
```

5. **(Optional)** Run the `util appinfocommand` to check all available information about `ApplicationProperties_t` struct in a signed GBL image.

```
commander util appinfo bootloader-uart-xmodem.s37
```

```
Parsing file bootloader-uart-xmodem.s37...
Found application properties in image.
Application properties info:
Application properties location : 0x00002b30
Signature location              : 0x00002c44Signature type              : ECDSA-P256
Long token section address      : Not set (0x00000000)
Application data info:
For Series 2 devices: If rollback prevention is enabled in the OTP configuration, the device will not boot if the device has seen an application with a higher version number.
App type                        : Bootloader (APPLICATION_TYPE_BOOTLOADER)App version                : 0x02000000
Product ID                      : Not set (0x00000000000000000000000000000000)
No certificate found in image.
For Series 2 devices: If the configuration flag SECURE_BOOT_VERIFY_CERTIFICATE is set or a device has previously seen certificate based signing, it will not accept direct signing.
DONE
```

6. The signed bootloader image file (`.s37`) can be used for production programming or for generating a GBL upgrade image file for bootloader upgrade.
7. Run the `flash` command to program the signed bootloader image (`bootloader-uart-xmodem.s37`) to the device if the device does not have a bootloader.

```
commander flash bootloader-uart-xmodem.s37 --device EFR32MG21A010F1024 --serialno 440048205
```

```
Parsing file bootloader-uart-xmodem.s37...
Writing 16384 bytes starting at address 0x00000000
Comparing range 0x00000000 - 0x00003FFF (16 KiB)
Programming range 0x00000000 - 0x00001FFF (8 KiB)
Programming range 0x00002000 - 0x00003FFF (8 KiB)
DONE
```

Application Image File

1. Follow the procedures in Generate an Unsigned Application Image to generate an unsigned application image for the bootloader.
2. **(Using Simplicity Commander)** Run the `convert` command with **Private Sign Key** to overwrite the unsigned application image file with the signed application image file (`blink_baremetal.s37`).

```
commander convert blink_baremetal.s37 --secureboot --keyfile sign_key.pem --verify sign_pubkey.pem
--outfile blink_baremetal.s37
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
Parsing file blink_baremetal.s37...
Found Application Properties at 0×000061bc
Writing Application Properties signature pointer to point to 0×000064d8
Setting signature type in Application Properties: 0×00000001
Image SHA256: 8b58ec567126aa1f6baa88afc916581477745aca6f47697ec093512fc30dcc6f
R = 056E3AA36BD882B5467D44A56DB7CC1AEE44D45BC9B98FAB05BE2C032573A1F7
S = BE1D27CE7877D0BC761C0F02690CC74251EBE3A458474C573C21B3A738A03577
Verifying signed image...
Writing to blink_baremetal.s37...
Overwriting file: blink_baremetal.s37...
DONE
```

3. **(Using an HSM and Simplicity Commander)** Run the `convert` command with **--extsign** option to generate an external signing application image file (`blink_baremetal.extsign`).

```
commander convert blink_baremetal.s37 --secureboot --extsign --outfile blink_baremetal
```

```
Parsing file blink_baremetal.s37...
Found Application Properties at 0×00006198
Writing Application Properties signature pointer to point to 0×0000643c
Setting signature type in Application Properties: 0×00000001
Writing to blink_baremetal.extsign...
DONE
```

Use an HSM containing the Private Sign Key to generate the signature for the external signing application image. This example uses the OpenSSL with the **Private Sign Key** to simulate this process. The signature is in the `app_signature.der` .

```
openssl dgst -sha256 -binary -sign sign_key.pem -out app_signature.der blink_baremetal.extsign
```

Run the `convert` command with the **application image signature** to overwrite the unsigned application image file with the signed application image file. (`blink_baremetal.s37`).

```
commander convert blink_baremetal.s37 --secureboot --signature app_signature.der --verify sign_pubkey.pem
--outfile blink_baremetal.s37
```

```
Parsing file blink_baremetal.s37...
Parsing signature file app_signature.der...
R = BD5BDC866CE67DA104B1E7B686C45B7BF96F2643154D37ACC63DACDF69C27E89
S = 2DD3BFFAC857A5B0BD8C9B4DDB23D21944D062F8E431D36541B84EF411C1CC92
Found Application Properties at 0×000061bc
Writing Application Properties signature pointer to point to 0×000064d8
Setting signature type in Application Properties: 0×00000001
Verifying signed image...
Writing to blink_baremetal.s37...
Overwriting file: blink_baremetal.s37...
DONE
```

4. **(Optional)** Run the `util appinfocommand` to check all available information about `ApplicationProperties_t` struct in a signed application image.

```
commander util appinfo blink_baremetal.s37
```

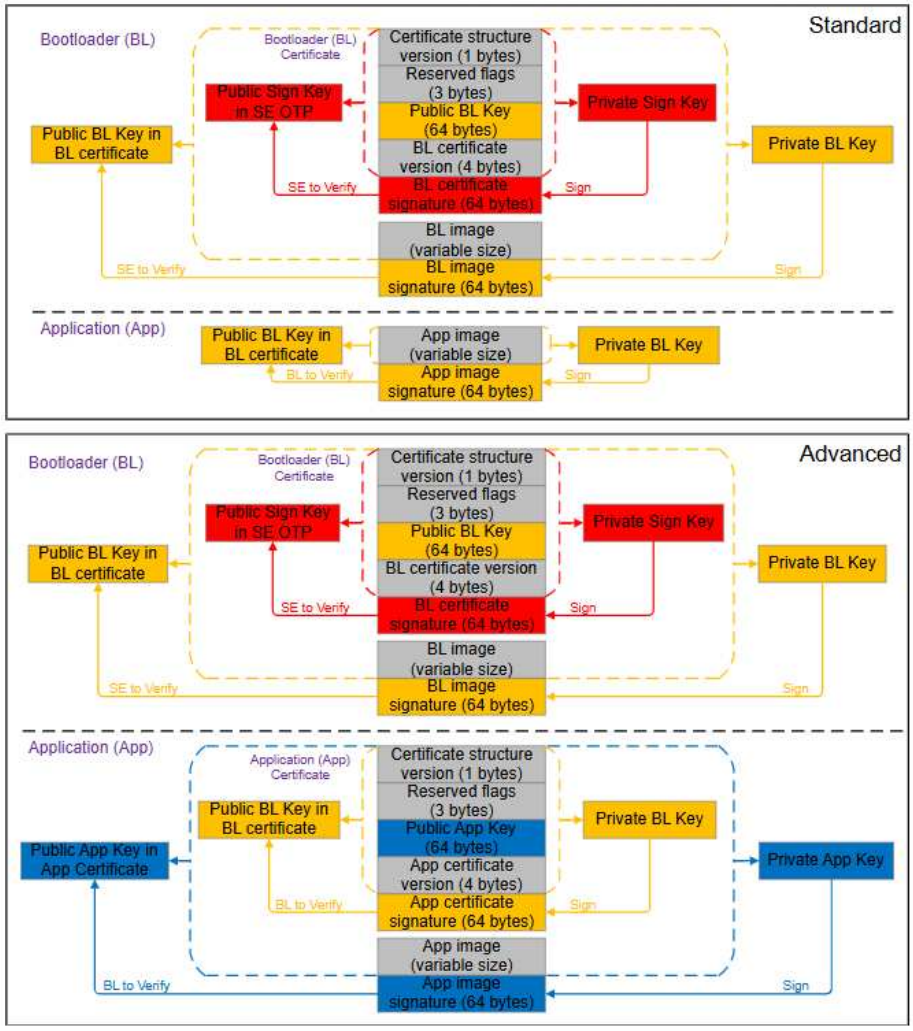
Copyright © 2025 Silicon Laboratories. All rights reserved.

```
Parsing file blink_baremetal.s37...
Found application properties in image.
Application properties info:
Application properties location : 0x000061bc
Signature location : 0x000064d8Signature type : ECDSA-P256
Long token section address : Not set (0x00000000)
Application data info:
For Series 2 devices: If rollback prevention is enabled in the OTP configuration, the device will not boot if the device has seen an application with a higher version number.
App type : MCU application (APPLICATION_TYPE_MCU)App version : 0x00000001
Product ID : Not set (0x00000000000000000000000000000000)
No certificate found in image.
For Series 2 devices: If the configuration flag SECURE_BOOT_VERIFY_CERTIFICATE is set or a device has previously seen certificate based signing, it will not accept direct signing.
DONE
```

5. The signed application image file (`.s37`) can be used for production programming or for generating a GBL upgrade image file for application upgrade.

Signing for Certificate-Based Secure Boot

The following figure describes the signing and verification for certificate-based Secure Boot. The user can freely switch between standard and advanced certificate-based Secure Boot by upgrading the application firmware without and with the application certificate.



Copyright © 2025 Silicon Laboratories. All rights reserved.

Certificate

The following table describes the elements of a certificate.

Table: Certificate Structure

Element	Description
Certificate structure version	The version of the certificate structure.
Reserved flags	Reserved in the current certificate structure version.
Certificate public key	ECDSA-P256 public key, X and Y coordinates concatenated, used to validate the image.
Certificate version	The version of the running certificate.
Certificate signature	ECDSA-P256 signature, used for the authentication of the public key and the certificate version.

> **Note:**
>
> - The `application_properties.h` in the Windows folder below defines the parameters of the certificate structure (`ApplicationCertificate_t`).
>   For GSDK v3.2 and lower: `C:\SiliconLabs\SimplicityStudio\v5\developer\sdks\gecko_sdk_suite\<GSDK VERSION>\platform\bootloader\api`
>   For GSDK v4.0 and higher: `C:\Users\<PC USER NAME>\SimplicityStudio\SDKs\gecko_sdk\platform\bootloader\api`
> - The certificate is not in X.509 format.

Private/Public Key Pair

The following table describes two Private/Public Key pairs used in certificates for certificate-based Secure Boot. The user can use Simplicity Commander or HSM to generate these key pairs.

Table: Certificates and Key Pairs for Certificate-Based Secure Boot Examples

Certificate	Private Key	Public Key	Description
Bootloader (bl_cert.bin) (1)	bl_cert_key.pem (Private Bootloader Key)	bl_cert_pubkey.pem (Public Bootloader Key)	The bootloader certificate is signed by the Private Sign Key corresponding to the Public Sign Key in SE OTP.
Application (app_cert.bin) (2)	app_cert_key.pem (Private Application Key)	app_cert_pubkey.pem (Public Application Key)	The application certificate is signed by the Private Bootloader Key corresponding to the Public Bootloader Key in the bootloader certificate.

> **Note:**
>
> 1. a. Certificate version in the bootloader certificate < certificate version in SE flash - the certificate is rejected.
>   b. Certificate version in the bootloader certificate = certificate version in SE flash - the certificate is accepted.
>   c. Certificate version in the bootloader certificate > certificate version in SE flash - the certificate is accepted.
>   The certificate version in SE flash is updated to match (revocation mechanism).
> 2. The certificate version in the application certificate is compared with the certificate version in the bootloader certificate. The application certificate is accepted if its version is equal to or higher than the certificate version in the bootloader certificate.

The following sections provide two methods to sign the bootloader image and application image files. All procedures assume the required files are in the same folder.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Using Simplicity Commander
2. Using an HSM and Simplicity Commander

Bootloader Image File

1. If the SE OTP is not provisioned, follow the procedures in Simplicity Commander to set up the certificate-based Secure Boot configuration for the bootloader.
2. Follow the procedures in AppBuilder or Bootloader-core Software Component to set up the certificate-based Secure Boot configuration for the user application to generate an unsigned bootloader image.
3. **(Using Simplicity Commander)** Run the `util gencert` command with **Public Bootloader Key** and **Private Sign Key** to generate the bootloader certificate (`bl_cert.bin`). Refer to Certificates and Key Pairs for Certificate-Based Secure Boot Examples for details about the `--cert-version` for bootloader certificate.

```
commander util gencert --cert-type secureboot --cert-version 1 --cert-pubkey bl_cert_pubkey.pem
--sign sign_key.pem --outfile bl_cert.bin
```

```
Successfully signed certificate
DONE
```

Run the `convert` command with **Bootloader Certificate** and **Private Bootloader Key** to overwrite the unsigned bootloader image file with the signed bootloader image file (`bootloader-uart-xmodem.s37`).

```
commander convert bootloader-uart-xmodem.s37 --secureboot --certificate bl_cert.bin
--keyfile bl_cert_key.pem --outfile bootloader-uart-xmodem.s37
```

```
Parsing file bootloader-uart-xmodem.s37...
Writing certificate to location 0x00002cf0
Private key matches public key in certificate.
Found Application Properties at 0x00002d78
Writing Application Properties signature pointer to point to 0x00002f64
Setting signature type in Application Properties: 0x00000001
Image SHA256: 3cf574b688853a801e8dc98687414db27f886c60c55dbf7fea2d47633df94e8d
R = C866592B4CB7BAD9EFC35985F1B9D52C65C26453D4808597EEEFFB16DC4AA962
S = 94CAA21ED5D7772F96BBF4D24A0711A94DCCB6D4D38DFA45182876B9BE2A8DE3
Verifying signed image...
Writing to bootloader-uart-xmodem.s37...
Overwriting file: bootloader-uart-xmodem.s37...
DONE
```

4. **(Using an HSM and Simplicity Commander)** Run the `util gencert` command with **Public Bootloader Key** and **--extsign** option to generate an external signing bootloader certificate (`bl_cert.extsign`). Refer to Certificates and Key Pairs for Certificate-Based Secure Boot Examples for details about the `--cert-version` for bootloader certificate.

```
commander util gencert --cert-type secureboot --cert-version 1 --cert-pubkey bl_cert_pubkey.pem--extsign --outfile bl_cert
DONE
```

Use an HSM containing the Private Sign Key to generate the signature for the external signing bootloader certificate. This example uses the OpenSSL with the **Private Sign Key** to simulate this process. The signature is in the `bl_cert_signature.der` .

```
openssl dgst -sha256 -binary -sign sign_key.pem -out bl_cert_signature.der bl_cert.extsign
```

Run the `util signcert` command with the **bootloader certificate signature** to generate the bootloader certificate (`bl_cert.bin`).

```
commander util signcert bl_cert.extsign --cert-type secureboot --signature bl_cert_signature.der
--verify sign_pubkey.pem --outfile bl_cert.bin
```

```
R = 065A58EA6CE6BBA44F3C59C6D255A901DBBC55FA97F261658B2026ABC8CD9680
S = 8A0011AA6393BC284B13C8313EE6772030DE07E213E74CA0FEA740F3D33E6518
Successfully verified signature
Successfully signed certificate
DONE
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Run the `convert` command with **Bootloader Certificate** and **--extsign** option to generate an external signing bootloader image file (`bootloader-uart-xmodem.extsign`).

```
commander convert bootloader-uart-xmodem.s37 --secureboot --certificate bl_cert.bin--extsign
--outfile bootloader-uart-xmodem
```

```
Parsing file bootloader-uart-xmodem.s37...
Writing certificate to location 0×00002cf0
Found Application Properties at 0×00002d78
Writing Application Properties signature pointer to point to 0×00002f64
Setting signature type in Application Properties: 0×00000001
Writing to bootloader-uart-xmodem.extsign...
DONE
```

Use an HSM containing the Private Bootloader Key to generate the signature for the external signing bootloader image. This example uses the OpenSSL with the **Private Bootloader Key** to simulate this process. The signature is in the `bl_signature.der` .

```
openssl dgst -sha256 -binary -sign bl_cert_key.pem -out bl_signature.der bootloader-uart-xmodem.extsign
```

Run the `convert` command with the **Bootloader Certificate** and **bootloader image signature** to overwrite the unsigned bootloader image file with the signed bootloader image file (`bootloader-uart-xmodem.s37`).

```
commander convert bootloader-uart-xmodem.s37 --secureboot --certificate bl_cert.bin
--signature bl_signature.der --outfile bootloader-uart-xmodem.s37
```

```
Parsing file bootloader-uart-xmodem.s37...
Writing certificate to location 0×00002cf0
Parsing signature file bl_signature.der...
R = 7AAA17E8F875F2572AF7692079ED2C48A1329D3AA2E616E7E13007CE68C32A21
S = 6E5F1ACD929A1AC05DE9F4BC5CBDE0E076B32EDA353C5E571C7850AAB86BCCFB
Found Application Properties at 0×00002d78
Writing Application Properties signature pointer to point to 0×00002f64
Setting signature type in Application Properties: 0×00000001
Verifying signed image...
Writing to bootloader-uart-xmodem.s37...
Overwriting file: bootloader-uart-xmodem.s37...
DONE
```

5. **(Optional)** Run the `util verifysign` command with **Public Sign Key** to verify that the **Bootloader Certificate and image** were correctly signed.

```
commander util verifysign bootloader-uart-xmodem.s37 --verify sign_pubkey.pem
```

```
Parsing file bootloader-uart-xmodem.s37...
Found application properties at 0×00002d78
Found certificate at 0×00002cf0
Successfully verified certificate signature with verification key.
Using certificate key to verify application signature.
Successfully verified application signature.
DONE
```

6. **(Optional)** Run the `util appinfo` command to check all available information about `ApplicationProperties_t` struct in a signed GBL image.

```
commander util appinfo bootloader-uart-xmodem.s37
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
Parsing file bootloader-uart-xmodem.s37...
Found application properties in image.
Application properties info:
Application properties location : 0x00002d00
Signature location              : 0x00002e14Signature type              : ECDSA-P256
Long token section address      : Not set (0x00000000)
Application data info:
For Series 2 devices: If rollback prevention is enabled in the OTP configuration, the device will not boot if the device has seen an application with a higher version number.
App type                        : Bootloader (APPLICATION_TYPE_BOOTLOADER)App version                : 0x02000000
Product ID                      : Not set (0x00000000000000000000000000000000)
Found certificate at 0x00002c78
Application certificate info:
Certificate located at          : 0x00002c78
Certificate version             : 0x00000001
Certificate key                 : 0xb1bc6f6fa56640ed522b2ee0f5b3cf7e5d48f60be8148f0dc08440f0a4e1dca4
                                  7c04119ed6a1be31b7707e5f9d001a659a051003e95e1b936f05c37ea793ad63
Certificate signature           : 0xef3b53368d4cd7821eb30a96140bbde8840378cfea30687a8c10642e1c7728fd
                                  309f976adf46e4eac62a2233f0c1f08f4e58344bdec61775b5282ceb351bb3d0
DONE
```

7. The signed bootloader image file (`.s37`) can be used for production programming or for generating a GBL upgrade image file for bootloader upgrade.
8. Run the `flash` command to program the signed bootloader image (`bootloader-uart-xmodem.s37`) to the device if the device does not have a bootloader.

```
commander flash bootloader-uart-xmodem.s37 --device EFR32MG21A010F1024 --serialno 440048205
```

```
Parsing file bootloader-uart-xmodem.s37...
Writing 16384 bytes starting at address 0x00000000
Comparing range 0x00000000 - 0x00003FFF (16 KiB)
Programming range 0x00000000 - 0x00001FFF (8 KiB)
Programming range 0x00002000 - 0x00003FFF (8 KiB)
DONE
```

Application Image File (Standard Certificate-Based)

1. Follow the procedures in Generate an Unsigned Application Image to generate an unsigned application image for the bootloader.
2. **(Using Simplicity Commander)** Run the `convert` command with **Private Bootloader Key** to overwrite the unsigned application image file with the signed application image file (`blink_baremetal.s37`).

```
commander convert blink_baremetal.s37 --secureboot --keyfile bl_cert_key.pem --verify bl_cert_pubkey.pem
--outfile blink_baremetal.s37
```

```
Parsing file blink_baremetal.s37...
Found Application Properties at 0x000061bc
Writing Application Properties signature pointer to point to 0x000064d8
Setting signature type in Application Properties: 0x00000001
Image SHA256: 8b58ec567126aa1f6baa88afc916581477745aca6f47697ec093512fc30dcc6f
R = 994739A26AB520A88A5550F1643AE263D88A952F185F96EE7021FA43DEA6138C
S = 65B7112715E2F999A6B216C32D3331AB63B2D31A0A1311DF36EEE62269F8D6AA
Verifying signed image...
Writing to blink_baremetal.s37...
Overwriting file: blink_baremetal.s37...
DONE
```

3. **(Using an HSM and Simplicity Commander)** Run the `convert` command with **--extsign** option to generate an external signing application image file (`blink_baremetal.extsign`).

```
commander convert blink_baremetal.s37 --secureboot --extsign --outfile blink_baremetal
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
Parsing file blink_baremetal.s37...
Found Application Properties at 0×000061bc
Writing Application Properties signature pointer to point to 0×000064d8
Setting signature type in Application Properties: 0×00000001
Writing to blink_baremetal.extsign...
DONE
```

Use an HSM containing the Private Bootloader Key to generate the signature for the external signing application image. This example uses the OpenSSL with the **Private Bootloader Key** to simulate this process. The signature is in the `app_signature.der`.

```
openssl dgst -sha256 -binary -sign bl_cert_key.pem -out app_signature.der blink_baremetal.extsign
```

Run the `convert` command with the **application image signature** to overwrite the unsigned application image file with the signed application image file. (`blink_baremetal.s37`).

```
commander convert blink_baremetal.s37 --secureboot --signature app_signature.der
--verify bl_cert_pubkey.pem --outfile blink_baremetal.s37
```

```
Parsing file blink_baremetal.s37...
Parsing signature file app_signature.der...
R = 8DA79B020E954D24C23423D80627E046E44052736F6546902F016D64464E82DE
S = 9D5A1CC424E97A5AD0352A4EEA6BBF565FED5FC61FF99E63AA73DFFEAD9EE399
Found Application Properties at 0×000061bc
Writing Application Properties signature pointer to point to 0×000064d8
Setting signature type in Application Properties: 0×00000001
Verifying signed image...
Writing to blink_baremetal.s37...
Overwriting file: blink_baremetal.s37...
DONE
```

4. **(Optional)** Run the `util verifysign` command with **Public Bootloader Key** to verify that the application image file was correctly signed.

```
commander util verifysign blink_baremetal.s37 --verify bl_cert_pubkey.pem
```

```
Parsing file blink_baremetal.s37...
Found application properties at 0×000061bc
Did not find application certificate in file
If the configuration flag SECURE_BOOT_VERIFY_CERTIFICATE is set or a device has previously seen certificate based signing, it will not
accept direct signing.
Successfully verified application signature.
DONE
```

5. **(Optional)** Run the `util appinfo` command to check all available information about `ApplicationProperties_t` struct in a signed application image.

```
commander util appinfo blink_baremetal.s37
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
Parsing file blink_baremetal.s37...
Found application properties in image.
Application properties info:
Application properties location : 0x000061bc
Signature location : 0x000064d8Signature type : ECDSA-P256
Long token section address : Not set (0x00000000)
Application data info:
For Series 2 devices: If rollback prevention is enabled in the OTP configuration, the device will not boot if the device has seen an application with a higher version number.
App type : MCU application (APPLICATION_TYPE_MCU)App version : 0x00000001
Product ID : Not set (0x00000000000000000000000000000000)
No certificate found in image.
For Series 2 devices: If the configuration flag SECURE_BOOT_VERIFY_CERTIFICATE is set or a device has previously seen certificate based signing, it will not accept direct signing.
DONE
```

6. The signed application image file (`.s37`) can be used for production programming or for generating a GBL upgrade image file for application upgrade.

Application Image File (Advanced Certificate-Based)

1. Follow the procedures in Generate an Unsigned Application Image to generate an unsigned application image for the bootloader.
2. **(Using Simplicity Commander)** Run the util gencert command with **Public Application Key** and **Private Bootloader Key** to generate the application certificate (app_cert.bin). Refer to Certificates and Key Pairs for Certificate-Based Secure Boot Examples for details about the `--cert-version` for application certificate.

```
commander util gencert --cert-type secureboot --cert-version 1 --cert-pubkey app_cert_pubkey.pem
--sign bl_cert_key.pem --outfile app_cert.bin
```

```
Successfully signed certificate
DONE
```

Run the convert command with **Application Certificate** and **Private Application Key** to overwrite the unsigned application image file with the signed application image file. (blink_baremetal.s37). This command will inject the application certificate into the application image before signing.

```
commander convert blink_baremetal.s37 --secureboot --certificate app_cert.bin --keyfile app_cert_key.pem
--outfile blink_baremetal.s37
```

```
Parsing file blink_baremetal.s37...
Writing certificate to location 0x000064d8
Private key matches public key in certificate.
Found Application Properties at 0x000061bc
Writing Application Properties signature pointer to point to 0x00006560
Setting signature type in Application Properties: 0x00000001
Image SHA256: 38fd11214c36abf3bb4c4eeda8cfdd2ca2ac2ff1e07072d555a06c74700a23f5
R = 6B4E3BB454513CAA4569415AE8F79453973AAC7FD1FC4914284B65010F3790A6
S = 1657CAAABED57988018726103835 8C83B1780A67CC41475370D94ED4445A5557
Verifying signed image...
Writing to blink_baremetal.s37...
Overwriting file: blink_baremetal.s37...
DONE
```

3. **(Using an HSM and Simplicity Commander)** Run the util gencert command with **Public Application Key** and **--extsign** option to generate an external signing application certificate (app_cert.extsign). Refer to Certificates and Key Pairs for Certificate-Based Secure Boot Examples for details about the `--cert-version` for application certificate.

```
commander util gencert --cert-type secureboot --cert-version 1 --cert-pubkey app_cert_pubkey.pem--extsign --outfile app_cert
```

```
DONE
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Use an HSM containing the Private Bootloader Key to generate the signature for the external signing application certificate. This example uses the OpenSSL with the **Private Bootloader Key** to simulate this process. The signature is in the app_cert_signature.der.

```
openssl dgst -sha256 -binary -sign bl_cert_key.pem -out app_cert_signature.der app_cert.extsign
```

Run the util signcert command with the **application certificate signature** to generate the application certificate (app_cert.bin).

```
commander util signcert app_cert.extsign --cert-type secureboot --signature app_cert_signature.der
--verify bl_cert_pubkey.pem --outfile app_cert.bin
```

```
R = 279D4FA1B801D108F82E30B0CF1164BF597549287290BD3883C5847B91095CCE
S = 567F0E219D2089EF4D79C3D94E43D2FADFE1899B71492ED358E6A1B46AE8162F
Successfully verified signature
Successfully signed certificate
DONE
```

Run the convert command with the **Application Certificate** and **--extsign** option to generate an external signing application image file (blink_baremetal.extsign).

```
commander convert blink_baremetal.s37 --secureboot --certificate app_cert.bin--extsign --outfile blink_baremetal
```

```
Parsing file blink_baremetal.s37...
Writing certificate to location 0×000064d8
Found Application Properties at 0×000061bc
Writing Application Properties signature pointer to point to 0×00006560
Setting signature type in Application Properties: 0×00000001
Writing to blink_baremetal.extsign...
DONE
```

Use an HSM containing the Private Application Key to generate the signature for the external signing application image. This example uses the OpenSSL with the **Private Application Key** to simulate this process. The signature is in the app_signature.der.

```
openssl dgst -sha256 -binary -sign app_cert_key.pem -out app_signature.der blink_baremetal.extsign
```

Run the convert command with the **Application Certificate** and **application image signature** to overwrite the unsigned application image file with the signed application image file. (blink_baremetal.s37). This command will inject the application certificate into the application image before signing.

```
commander convert blink_baremetal.s37 --secureboot --certificate app_cert.bin --signature app_signature.der
--verify app_cert_pubkey.pem --outfile blink_baremetal.s37
```

```
Parsing file blink_baremetal.s37...
Writing certificate to location 0×000064d8
Parsing signature file app_signature.der...
R = ADC2BAB959EC76CB2806C4649277669CF6E62A92ABFBBF20E551971449A8FCE0
S = B6B80130F95F62671372D1B2E471C82ADB08946C1F8938EED11F9822CE763A54
Found Application Properties at 0×000061bc
Writing Application Properties signature pointer to point to 0×00006560
Setting signature type in Application Properties: 0×00000001
Verifying signed image...
Verifying signed image...
Writing to blink_baremetal.s37...
Overwriting file: blink_baremetal.s37...
DONE
```

4. **(Optional)** Run the util verifysign command with **Public Bootloader Key** to verify that the **Application Certificate and image** were correctly signed.

```
commander util verifysign blink_baremetal.s37 --verify bl_cert_pubkey.pem
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
Parsing file blink_baremetal.s37...
Found application properties at 0x000061bc
Found certificate at 0x000064d8
Successfully verified certificate signature with verification key.
Using certificate key to verify application signature.
Successfully verified application signature.
DONE
```

5. **(Optional)** Run the util appinfo command to check all available information about `ApplicationProperties_t` struct in a signed application image.

```
commander util appinfo blink_baremetal.s37
```

```
Parsing file blink_baremetal.s37...
Found application properties in image.
Application properties info:
Application properties location : 0x000061bc
Signature location              : 0x00006560Signature type              : ECDSA-P256
Long token section address      : Not set (0x00000000)
Application data info:
For Series 2 devices: If rollback prevention is enabled in the OTP configuration, the device will not boot if the device has seen an application with a higher version number.
App type                        : MCU application (APPLICATION_TYPE_MCU)
App version                     : 0x00000001
Product ID                      : Not set (0x00000000000000000000000000000000)
Found certificate at 0x000064d8
Application certificate info:
Certificate located at          : 0x000064d8
Certificate version             : 0x00000001
Certificate key                 : 0xe562003cd86e225decfd35712e431a19ecd5031a079b06c1d473620a6be9f57a
                                  879820100fee074f28b5885fd6759f480b62aaa0717f96e245aab6635cfb1e11
Certificate signature           : 0x039aaba62b5258e68d16e167c3a611c719c542bb3483f5d4b522472b06adf30f
                                  8cfcc484bf8551a208256e3d2d8c9194a7d2ac551e2cac659a99822308a40aa6
DONE
```

6. The signed application image file (.s37) can be used for production programming or for generating a GBL upgrade image file for application upgrade.

Generate a GBL Upgrade Image File

This section describes how to generate the GBL upgrade image files for updating the bootloader, Secure Engine, and application firmware.

Note:

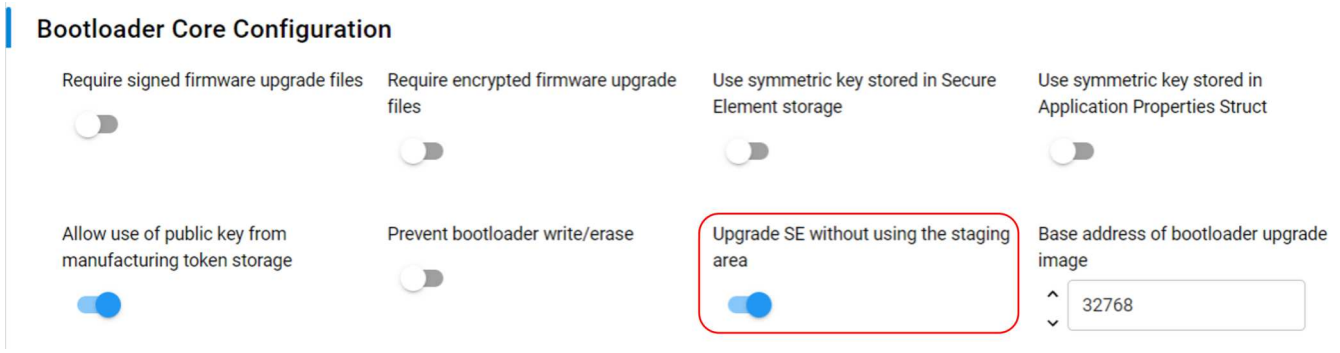
1. Following the procedures in AppBuilder or Bootloader-core Software Component to avoid overwriting the existing application image (if necessary) when upgrading the bootloader or SE.
2. For a standalone bootloader with communication interface, you can only generate three separate GBL upgrade image files containing bootloader, SE, and application images.
3. For an application bootloader with storage, you can generate a single GBL upgrade image file (see example below) with a combination of bootloader, SE, and application images.

```
commander gbl create all.gbl --app app.s37 --bootloader bl.s37 --seupgrade se.seu
```

4. A signed GBL upgrade image file is required if the user enables the **Require signed firmware upgrade files** option in AppBuilder or Bootloader-core Software Component. The following table shows which private key(s) can be used to sign the GBL upgrade image file (bootloader, SE, or application) on HSE and VSE devices. The VSE devices store a Public Sign Key copy on the top page of the main flash to verify the GBL upgrade image file for ECDSA-P256-SHA256 Secure Boot.
| Secure Boot |HSE |VSE | |-|-|-| | ECDSA-P256-SHA256 |Private Sign Key |Private Sign Key (Public Sign Key in main flash) | | Certificate-Based |Private Sign Key or Private Bootloader Key |Private Bootloader Key |

Copyright © 2025 Silicon Laboratories. All rights reserved.

5. An encrypted GBL upgrade image file is required if the user enables the **Require encrypted firmware upgrade files** option in AppBuilder or Bootloader-core Software Component. Refer to Provision GBL Decryption Key on how to provision the GBL Decryption Key for this option.
6. For an application bootloader with storage, the user can enable the **Upgrade SE without using the staging area** option in **GSDK v4.1.1 or higher** to directly fetch the SE image from the GBL upgrade image file in storage instead of copying the image to the pre-configured upgrade location.



To use the above option, the SE image cannot be in the encrypted part of the GBL upgrade image file if the **Require encrypted firmware upgrade files** option is enabled. Use the `--seunencrypted` option in **Simplicity Commander v1.13.0 or higher** (see example below) to generate an encrypted GBL upgrade image file with a SE image outside the encrypted part of the file.

```
commander gbl create se-upgrade.gbl --seupgrade secure-element.seu --seunencrypted --app myapp.s37
--encrypt aes_key.txt
```

The following sections provide two methods to sign the bootloader, Secure Engine, and application upgrade image files if the **Require signed firmware upgrade files** option is enabled.

1. Using Simplicity Commander
2. Using an HSM and Simplicity Commander

The sections also include encryption examples with an AES-128 key (like `aes_key.txt`) for the **Require encrypted firmware upgrade files** option. All procedures assume the required files are in the same folder.

Bootloader Upgrade

Copyright © 2025 Silicon Laboratories. All rights reserved.

1. **(Unsigned)** Run the `gbl create` command with **--bootloader** option to generate the bootloader GBL upgrade image file (`bootloader-uart-xmodem.gbl`) with the signed bootloader image file (`bootloader-uart-xmodem.s37`) from Signing for ECDSA-P256-SHA256 Secure Boot or Signing for Certificate-Based Secure Boot.
   Without encryption:

```
commander gbl create bootloader-uart-xmodem.gbl --bootloader bootloader-uart-xmodem.s37
```

```
Initializing GBL file...
Adding bootloader to GBL...
Writing GBL file bootloader-uart-xmodem.gbl...
DONE
```

   With encryption:

```
commander gbl create bootloader-uart-xmodem.gbl --bootloader bootloader-uart-xmodem.s37
--encrypt aes_key.txt
```

```
Initializing GBL file...
Adding bootloader to GBL...
Encrypting GBL...
Writing GBL file bootloader-uart-xmodem.gbl...
DONE
```

2. **(Signed - Using Simplicity Commander)** Run the `gbl create` command with **--bootloader** option to generate the signed bootloader GBL upgrade image file (`bootloader-uart-xmodem.gbl`) with **Private Sign Key** or **Private Bootloader Key** and the signed bootloader image file (`bootloader-uart-xmodem.s37`) from Signing for ECDSA-P256-SHA256 Secure Boot or Signing for Certificate-Based Secure Boot.
   Without encryption:

```
commander gbl create bootloader-uart-xmodem.gbl --bootloader bootloader-uart-xmodem.s37 --sign sign_key.pem
```

```
commander gbl create bootloader-uart-xmodem.gbl --bootloader bootloader-uart-xmodem.s37
--sign bl_cert_key.pem
```

```
Initializing GBL file...
Adding bootloader to GBL...
Signing GBL...
Image SHA256: 3eb09993ffca5f9b34df3f38b65ab9d2f6619b828b014a186516016d4bbd80f7
R = C21E0C19254AC4F62374BBCA65DEBB42C7349384F5527330CD030A51DC2170F7
S = E1680C3670DE68D731086845E2726EF3BF07B96EB54AA2DB2F390F60BDB6DAB2
Writing GBL file bootloader-uart-xmodem.gbl...
DONE
```

   With encryption:

```
commander gbl create bootloader-uart-xmodem.gbl --bootloader bootloader-uart-xmodem.s37 --sign sign_key.pem--encrypt aes_key.txt
```

```
commander gbl create bootloader-uart-xmodem.gbl --bootloader bootloader-uart-xmodem.s37
--sign bl_cert_key.pem--encrypt aes_key.txt
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
Initializing GBL file...
Adding bootloader to GBL...
Encrypting GBL...
Signing GBL...
Image SHA256: a2ef5e19e6b2ec327010af9fcb7de861c6a311987b7d2c39ef6439ca8b9999c4
R = B74972171109A05A9B1B45E1C8A2BCD57EAB1EA8F8A7936BBDB14CF6FA36B28C
S = 9B673083684B01C2D05BD702A4D29A6EDCF88C471C3FA8B71DDE00AE85861930
Writing GBL file bootloader-uart-xmodem.gbl...
DONE
```

3. **(Signed - Using an HSM and Simplicity Commander)** Run the `gbl create` command with **--bootloader** and **--extsign** options to generate an external signing bootloader GBL upgrade image file (`bootloader-uart-xmodem.extsign`) with the signed bootloader image file (`bootloader-uart-xmodem.s37`) from Signing for ECDSA-P256-SHA256 Secure Boot or Signing for Certificate-Based Secure Boot.
   Without encryption:

```
commander gbl create bootloader-uart-xmodem --bootloader bootloader-uart-xmodem.s37 --extsign
```

```
Initializing GBL file...
Adding bootloader to GBL...
Preparing GBL for external signing...
Writing GBL file bootloader-uart-xmodem.extsign...
DONE
```

   With encryption:

```
commander gbl create bootloader-uart-xmodem --bootloader bootloader-uart-xmodem.s37 --extsign--encrypt aes_key.txt
```

```
Initializing GBL file...
Adding bootloader to GBL...
Encrypting GBL...
Preparing GBL for external signing...
Writing GBL file bootloader-uart-xmodem.extsign...
DONE
```

   Use an HSM containing the Private Sign Key or Private Bootloader Key to generate the signature for the external signing bootloader GBL upgrade image file. This example uses the OpenSSL with the **Private Sign Key** or **Private Bootloader Key** to simulate this process. The signature is in the `gbl_signature.der` .

```
openssl dgst -sha256 -binary -sign sign_key.pem -out gbl_signature.der bootloader-uart-xmodem.extsign
```

```
openssl dgst -sha256 -binary -sign bl_cert_key.pem -out gbl_signature.der bootloader-uart-xmodem.extsign
```

   Run the `gbl sign` command with the **signature** above to generate a signed bootloader GBL upgrade image file (`bootloader-uart-xmodem.gbl`).

```
commander gbl sign bootloader-uart-xmodem.extsign --signature gbl_signature.der --verify sign_pubkey.pem
--outfile bootloader-uart-xmodem.gbl
```

```
commander gbl sign bootloader-uart-xmodem.extsign --signature gbl_signature.der
--verify bl_cert_pubkey.pem --outfile bootloader-uart-xmodem.gbl
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
Reading GBL data from bootloader-uart-xmodem.extsign...
Parsing signature file gbl_signature.der...
R = 90F0A3C0D5D9ED2DC10EB3F55595FF21AB31307DC6283E3F3B7494A30FB741D4
S = 2765041F515A960F048CA250BFAB92031D4D1E569FB3F917C9329E7362C17B51
Writing signature to GBL...
Verifying GBL...
Successfully verified GBL signatureWriting GBL file bootloader-uart-xmodem.gbl...
DONE
```

4. Follow the procedures in Upload a GBL Upgrade Image File to upgrade the bootloader with the bootloader GBL upgrade image file.

Secure Engine Upgrade

1. **(Unsigned)** Run the `gbl create` command with **--seupgrade** option to generate the SE GBL upgrade image file (`s2c1_se_fw_upgrade_1v2p9.gbl`) with the SE image file (`s2c1_se_fw_upgrade_1v2p9.seu`).
   Without encryption:

```
commander gbl create s2c1_se_fw_upgrade_1v2p9.gbl --seupgrade s2c1_se_fw_upgrade_1v2p9.seu
```

```
Initializing GBL file...
Adding Secure Element upgrade image to GBL...
Writing GBL file s2c1_se_fw_upgrade_1v2p9.gbl...
DONE
```

   With encryption:

```
commander gbl create s2c1_se_fw_upgrade_1v2p9.gbl --seupgrade s2c1_se_fw_upgrade_1v2p9.seu
--encrypt aes_key.txt
```

```
Initializing GBL file...
Adding Secure Element upgrade image to GBL...
Encrypting GBL...
Writing GBL file s2c1_se_fw_upgrade_1v2p9.gbl...
DONE
```

2. **(Signed - Using Simplicity Commander)** Run the `gbl create` command with **--seupgrade** option to generate the signed SE GBL upgrade image file (`s2c1_se_fw_upgrade_1v2p9.gbl`) with **Private Sign Key** or **Private Bootloader Key** and the SE image file (`s2c1_se_fw_upgrade_1v2p9.seu`).
   Without encryption:

```
commander gbl create s2c1_se_fw_upgrade_1v2p9.gbl --seupgrade s2c1_se_fw_upgrade_1v2p9.seu
--sign sign_key.pem
```

```
commander gbl create s2c1_se_fw_upgrade_1v2p9.gbl --seupgrade s2c1_se_fw_upgrade_1v2p9.seu
--sign bl_cert_key.pem
```

```
Initializing GBL file...
Adding Secure Element upgrade image to GBL...
Signing GBL...
Image SHA256: 599d7fc35996b4715441b642709ed262525d09d811d4726e423c0d605ec0f0bf
R = EF8EC2DDEDDF44DF88FEAD4ED0A9FDC6351B4D745D5A05BFB87204791871A525
S = FCB26EF005D97E8C5341153A210AE9927E1CF646A3E473FFB90DA8C857E6421F
Writing GBL file s2c1_se_fw_upgrade_1v2p9.gbl...
DONE
```

   With encryption:

```
commander gbl create s2c1_se_fw_upgrade_1v2p9.gbl --seupgrade s2c1_se_fw_upgrade_1v2p9.seu
--sign sign_key.pem--encrypt aes_key.txt
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
commander gbl create s2c1_se_fw_upgrade_1v2p9.gbl --seupgrade s2c1_se_fw_upgrade_1v2p9.seu
--sign bl_cert_key.pem--encrypt aes_key.txt
```

```
Initializing GBL file...
Adding Secure Element upgrade image to GBL...
Encrypting GBL...
Signing GBL...
Image SHA256: a5ab368c99c49503a7dfb6aef1724dc3f883eabddcac7b089148035483e24322
R = 289F05910A8E0735648260FA7A1C67731CA86FB2DFCB9B405EC8D297892915A7
S = F21D18351442D0E5F49CFBEDA2C0EFA8B7F0911B4B6216EB48250CB5889ECAFD
Writing GBL file s2c1_se_fw_upgrade_1v2p9.gbl...
DONE
```

3. **(Signed - Using an HSM and Simplicity Commander)** Run the `gbl create` command with **--seupgrade** and **--extsign** options to generate an external signing SE GBL upgrade image file (`s2c1_se_fw_upgrade_1v2p9.extsign`) with the SE image file (`s2c1_se_fw_upgrade_1v2p9.seu`).

   Without encryption:

```
commander gbl create s2c1_se_fw_upgrade_1v2p9 --seupgrade s2c1_se_fw_upgrade_1v2p9.seu --extsign
```

```
Initializing GBL file...
Adding Secure Element upgrade image to GBL...
Preparing GBL for external signing...
Writing GBL file s2c1_se_fw_upgrade_1v2p9.extsign...
DONE
```

   With encryption:

```
commander gbl create s2c1_se_fw_upgrade_1v2p9 --seupgrade s2c1_se_fw_upgrade_1v2p9.seu --extsign--encrypt aes_key.txt
```

```
Initializing GBL file...
Adding Secure Element upgrade image to GBL...
Encrypting GBL...
Preparing GBL for external signing...
Writing GBL file s2c1_se_fw_upgrade_1v2p9.extsign...
DONE
```

   Use an HSM containing the Private Sign Key or Private Bootloader Key to generate the signature for the external signing SE GBL upgrade image file. This example uses the OpenSSL with the **Private Sign Key** or **Private Bootloader Key** to simulate this process. The signature is in the `gbl_signature.der` .

```
openssl dgst -sha256 -binary -sign sign_key.pem -out gbl_signature.der s2c1_se_fw_upgrade_1v2p9.extsign
```

```
openssl dgst -sha256 -binary -sign bl_cert_key.pem -out gbl_signature.der s2c1_se_fw_upgrade_1v2p9.extsign
```

   Run the `gbl sign` command with the **signature** above to generate a signed SE GBL upgrade image file (`s2c1_se_fw_upgrade_1v2p9.gbl`).

```
commander gbl sign s2c1_se_fw_upgrade_1v2p9.extsign --signature gbl_signature.der --verify sign_pubkey.pem
--outfile s2c1_se_fw_upgrade_1v2p9.gbl
```

```
commander gbl sign s2c1_se_fw_upgrade_1v2p9.extsign --signature gbl_signature.der
--verify bl_cert_pubkey.pem --outfile s2c1_se_fw_upgrade_1v2p9.gbl
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
Reading GBL data from s2c1_se_fw_upgrade_1v2p9.extsign...
Parsing signature file gbl_signature.der...
R = 2798B98194EE02717C738B5866ABD8D234D0F0E096E90495D371D2507D8E1C67
S = 19F2586E2C6177D6B4EEC708E006F67334C989D0398D4233C686C98ECB6992FB
Writing signature to GBL...
Verifying GBL...
Successfully verified GBL signatureWriting GBL file s2c1_se_fw_upgrade_1v2p9.gbl...
DONE
```

4. Follow the procedures in Upload a GBL Upgrade Image File to upgrade the SE with the SE GBL upgrade image file.

> Note:
>
> - The `sign_key.pem/sign_pubkey.pem` key pair is for Signing for ECDSA-P256-SHA256 Secure Boot, and the `bl_cert_key.pem/bl_cert_pubkey.pem` key pair is for Signing for Certificate-Based Secure Boot.
> - Trying to apply a lower version of the SE image file (`.seu`) to the device will be ignored.

Application Upgrade

1. **(Unsigned)** Run the gbl create command with **--app** option to generate the application GBL upgrade image file (`blink_baremetal.gbl`) with the signed application image file (`blink_baremetal.s37`) from Signing for ECDSA-P256-SHA256 Secure Boot or Signing for Certificate-Based Secure Boot.
   Without encryption:

```
commander gbl create blink_baremetal.gbl --app blink_baremetal.s37
```

```
Parsing file blink_baremetal.s37...
Initializing GBL file...
Adding application to GBL...
Writing GBL file blink_baremetal.gbl...
DONE
```

   With encryption:

```
commander gbl create blink_baremetal.gbl --app blink_baremetal.s37 --encrypt aes_key.txt
```

```
Parsing file blink_baremetal.s37...
Initializing GBL file...
Adding application to GBL...
Encrypting GBL...
Writing GBL file blink_baremetal.gbl...
DONE
```

2. **(Signed - Using Simplicity Commander)** Run the `gbl create` command with **--app** option to generate the signed application GBL upgrade image file (`blink_baremetal.gbl`) with **Private Sign Key** or **Private Bootloader Key** and the signed application image file (`blink_baremetal.s37`) from Signing for ECDSA-P256-SHA256 Secure Boot or Signing for Certificate-Based Secure Boot.
   Without encryption:

```
commander gbl create blink_baremetal.gbl --app blink_baremetal.s37 --sign sign_key.pem
```

```
commander gbl create blink_baremetal.gbl --app blink_baremetal.s37 --sign bl_cert_key.pem
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
Parsing file blink_baremetal.s37...
Initializing GBL file...
Adding application to GBL...
Signing GBL...
Image SHA256: 116c1be47d799ab75afc7b3f4c9a8023e5cd031103b1d28c578eebfaf1ad73d2
R = CE4D85C058301A2437440E00385D97E496F1D8B5CAFFB8C184F8A88B5266E3E9
S = 90BBF754EBC0AB343CC32AA06ADED85F9D12D1A67CA6608F9085137142000A40
Writing GBL file blink_baremetal.gbl...
DONE
```

With encryption:

```
commander gbl create blink_baremetal.gbl --app blink_baremetal.s37 --sign sign_key.pem--encrypt aes_key.txt
```

```
commander gbl create blink_baremetal.gbl --app blink_baremetal.s37 --sign bl_cert_key.pem--encrypt aes_key.txt
```

```
Parsing file blink_baremetal.s37...
Initializing GBL file...
Adding application to GBL...
Encrypting GBL...
Signing GBL...
Image SHA256: 24092ed828e6fffc41e7ed40c046b80789ef2337da0d7373a15e59d27e07e0fc
R = 6143F307119402DC55C63220D54542B84EBEFB324963C63796A37B9845482B35
S = AE3644D59DF3A27F45B335CB4F79D2347364958E0F152AF745FB7042537D1B6A
Writing GBL file blink_baremetal.gbl...
DONE
```

3. **(Signed - Using an HSM and Simplicity Commander)** Run the `gbl create` command with **--app** and **--extsign** options to generate an external signing application GBL upgrade image file (`blink_baremetal.extsign`) with the signed application image file (`blink_baremetal.s37`) from Signing for ECDSA-P256-SHA256 Secure Boot or Signing for Certificate-Based Secure Boot.

   Without encryption:

```
commander gbl create blink_baremetal --app blink_baremetal.s37 --extsign
```

```
Parsing file blink_baremetal.s37...
Initializing GBL file...
Adding application to GBL...
Preparing GBL for external signing...
Writing GBL file blink_baremetal.extsign...
DONE
```

   With encryption:

```
commander gbl create blink_baremetal --app blink_baremetal.s37 --extsign--encrypt aes_key.txt
```

```
Parsing file blink_baremetal.s37...
Initializing GBL file...
Adding application to GBL...
Encrypting GBL...
Preparing GBL for external signing...
Writing GBL file blink_baremetal.extsign...
DONE
```

   Use an HSM containing the Private Sign Key or Private Bootloader Key to generate the signature for the external signing application GBL upgrade image file. This example uses the OpenSSL with the **Private Sign Key** or **Private Bootloader Key** to simulate this process. The signature is in the `gbl_signature.der` .

```
openssl dgst -sha256 -binary -sign sign_key.pem -out gbl_signature.der blink_baremetal.extsign
```

```
openssl dgst -sha256 -binary -sign bl_cert_key.pem -out gbl_signature.der blink_baremetal.extsign
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Run the `gbl sign` command with the **signature** above to generate a signed application GBL upgrade image file (`blink_baremetal.gbl`).

```
commander gbl sign blink_baremetal.extsign --signature gbl_signature.der --verify sign_pubkey.pem
--outfile blink_baremetal.gbl
```

```
commander gbl sign blink_baremetal.extsign --signature gbl_signature.der --verify bl_cert_pubkey.pem
--outfile blink_baremetal.gbl
```

```
Reading GBL data from blink_baremetal.extsign...
Parsing signature file gbl_signature.der...
R = 533499660E24F1620EF25D862FB607F46E9E4ECC41CBDECBE77C64EF1970D96A
S = FA8901878218F5F1DB0FAF8B074CE98A27C63FFDE63730CD49EE47E847B9811D
Writing signature to GBL...
Verifying GBL...
Successfully verified GBL signature
Writing GBL file blink_baremetal.gbl...
DONE
```

4. Follow the procedures in Upload a GBL Upgrade Image File to upgrade the application with the application GBL upgrade image file.

Note:

- The Simplicity Commander v1.11.0 or above supports GBL upgrade image file in `util verifysign` command.

```
commander util verifysign blink_baremetal.gbl --verify sign_pubkey.pem
```

```
Successfully verified GBL signature
DONE
```

- The Simplicity Commander v1.12.0 or above fixes a bug introduced in v1.11.0 when using the `--extsign` option on the GBL upgrade image file.

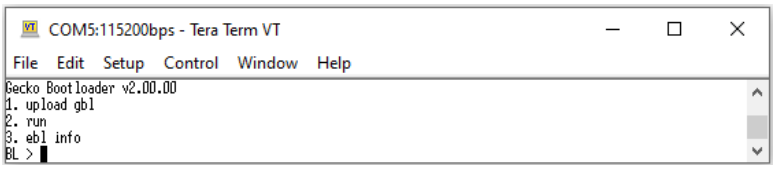
Upload a GBL Upgrade Image File

This section describes how to use UART XMODEM Bootloader v2.0.0 in GSDK v4.0 to upload a GBL upgrade image file (`.gbl`) to the device. The procedures and pictures may be different for the other versions of this example.

The GBL upgrade image file uses a proprietary format to store the upgrade image for a firmware upgrade. Use the `gbl create` command to generate the GBL upgrade image file for bootloader, application, and Secure Engine. Refer to UG266/Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher and Generate a GBL Upgrade Image File for more information about GBL upgrade image file creation.

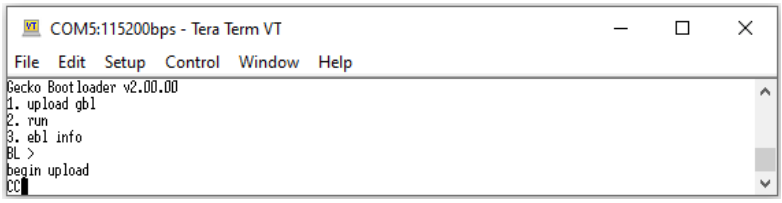
The user can use any terminal software that supports the XMODEM-CRC protocol for file transfer. This application note uses Tera Term as terminal software. The default serial port setting is 115200 bps 8-N-1.

1. Assume the UART XMODEM Bootloader and application firmware had already flashed to the radio board on WSTK.
2. Press the RESET and PB0 push buttons on the WSTK.
3. Release the RESET push button to run the UART XMODEM Bootloader.

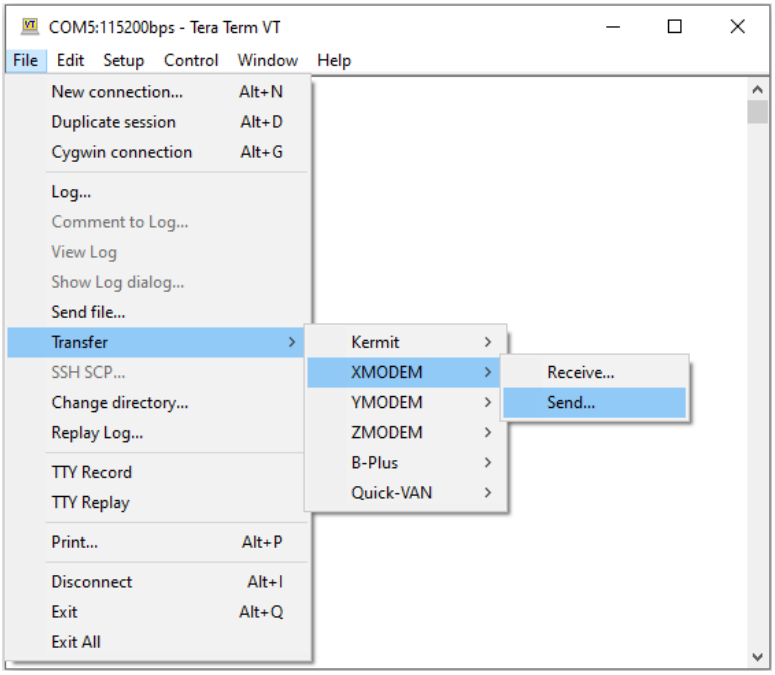


4. Release the PB0 push button. Press 1 (`upload gbl`) in Tera Term to upload a GBL upgrade image file.

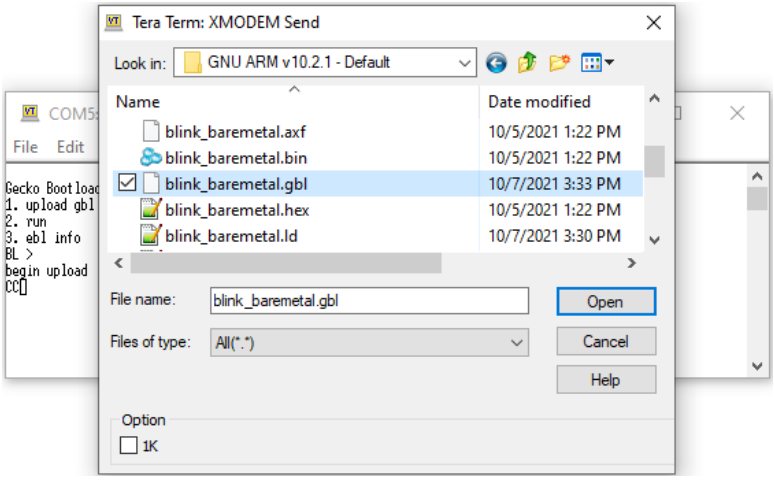
Copyright © 2025 Silicon Laboratories. All rights reserved.



5. Transfer a file through XMODEM-CRC in Tera Term, navigate to **File** → **Transfer** → **XMODEM** → **Send...**.

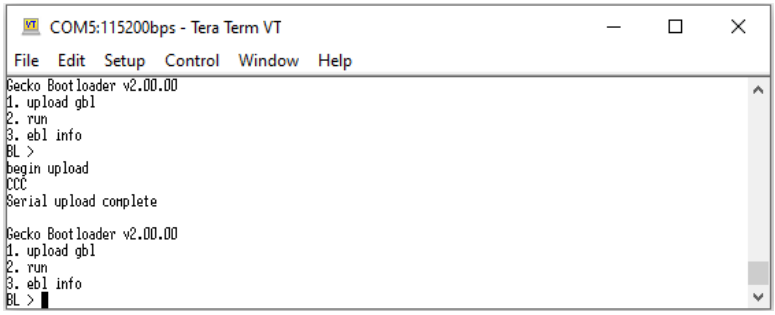


6. Select the target GBL upgrade image file. Click **[Open]** to upload.



7. If no error occurs, press 2 (`run`) to start a firmware upgrade.

Copyright © 2025 Silicon Laboratories. All rights reserved.



Upgrade to Certificate-Based Secure Boot

The user can upgrade the Series 2 devices deployed in the field from ECDSA-P256-SHA256 Secure Boot to certificate-based Secure Boot even the `SECURE_BOOT_VERIFY_CERTIFICATE` option in SE OTP is disabled.

```
commander security readconfig --serialno 440048205
```

```
MCU Flags
Secure Boot                : Enabled
Secure Boot Verify Certificate : Disabled
Secure Boot Anti Rollback      : Enabled
Secure Boot Page Lock Narrow   : Disabled
Secure Boot Page Lock Full     : Disabled
```

The following procedures for the upgrade to certificate-based Secure Boot is an **IRREVERSIBLE** process.

1. Follow the procedures in Generate Key and Signing to generate an ECDSA-P256 bootloader certificate key pair.
2. Follow the procedures in Signing for Certificate-Based Secure Boot to generate the signed GBL image file with the bootloader certificate key pair in step 1. The bootloader certificate version (`--cert-version` in the `util gencert` command) in this signed GBL image file must be **equal to or higher than one (≥ 1)**.
3. Follow the procedures in Generate a GBL Upgrade Image File to upgrade the bootloader to certificate-based Secure Boot. Use the Private Sign Key for ECDSA-P256-SHA256 Secure Boot to sign the bootloader GBL upgrade image file if required. SE will use the Public Bootloader Key to validate the bootloader image once SE identifies a bootloader certificate in the bootloader image. If the bootloader certificate version from step 2 is **higher than zero (> 0) and gets verified once**, SE will never again accept the ECDSA-P256-SHA256 Secure Boot signed bootloader image. Refer to the "Secure Boot Procedure" section in UG266/Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher for more information.

   ```
   commander security status --device EFR32MG21A010F1024 --serialno 440048205
   ```

   ```
   SE Firmware version : 1.2.9
   Serial number       : 00000000000000014b457fffe045a2d
   Debug lock          : Disabled
   Device erase        : Enabled
   Secure debug unlock : Disabled
   Tamper status       : Not OK
   Secure boot         : Enabled
   Boot status         : 0x18 - Failed: Secure Boot requires cert, but none found
   DONE
   ```

4. **(Standard Certificate-Based)** Follow the procedures in Signing for Certificate-Based Secure Boot to generate the signed application image file with the Private Bootloader Key in step 1.
5. **(Advanced Certificate-Based)** Follow the procedures in Generate Key and Signing to generate an ECDSA-P256 application certificate key pair.
   Follow the procedures in Signing for Certificate-Based Secure Boot to generate the signed application image file with the application certificate key pair in this step and the Private Bootloader Key in step 1. The application certificate version (`--`

Copyright © 2025 Silicon Laboratories. All rights reserved.

`cert-version` in the `util gencert` command) in this signed application image file must be **equal to or higher** than the bootloader certificate version in step 2 (Certificates and Key Pairs for Certificate-Based Secure Boot Examples).

6. Follow the procedures in Generate a GBL Upgrade Image File to upgrade application with the signed image from step 4 or 5 for certificate-based Secure Boot. Use the Private Sign Key or Private Bootloader key in **step 1** for certificate-based Secure Boot to sign the application GBL upgrade image file if required.

Certificate Revocation

The certificate revocation is the act of invalidating a certificate when its private key shows signs of being compromised. The following procedures describe how to revoke the Series 2 devices' bootloader certificates deployed in the field.

1. Follow the procedures in Generate Key and Signing to generate a new ECDSA-P256 bootloader certificate key pair.
2. Follow the procedures in Signing for Certificate-Based Secure Boot to generate the signed GBL image file with the bootloader certificate key pair in step 1. The bootloader certificate version (`--cert-version` in the `util gencert` command) in this signed GBL image file must be **higher than** the certificate version in SE flash (Certificates and Key Pairs for Certificate-Based Secure Boot Examples).
3. Follow the procedures in Generate a GBL Upgrade Image File to upgrade the bootloader with the signed image from step 2. Use the Private Sign Key or **existing**Private Bootloader Key for certificate-based Secure Boot to sign the bootloader GBL upgrade image file if required.
4. **(Standard Certificate-Based)** Follow the procedures in Signing for Certificate-Based Secure Boot to generate the signed application image file with the Private Bootloader Key in step 1.
5. **(Advanced Certificate-Based)** Follow the procedures in Signing for Certificate-Based Secure Boot to generate the signed application image file with the Private Bootloader Key in step 1. The application certificate version (`--cert-version` in the `util gencert` command) in this signed application image file must be **equal to or higher** than the bootloader certificate version in step 2 (Certificates and Key Pairs for Certificate-Based Secure Boot Examples).
   The user should generate a new ECDSA-P256 application certificate key pair if the Private Application Key for the application certificate is compromised.
6. Follow the procedures in Generate a GBL Upgrade Image File to upgrade the application with the signed image from step 4 or 5. Use the Private Sign Key or Private Bootloader key in **step 1** for certificate-based Secure Boot to sign the application GBL upgrade image file if required.

Upgrade to Secure Boot with RTSL

The following procedures describe upgrading Series 2 devices deployed in the field without Secure Boot to Secure Boot with RTSL.

1. **(Recommended)** Upgrade SE firmware to the latest version if available. See the *Gecko Bootloader Operation - Secure Engine Upgrade* section in UG266/Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher.
2. Follow the procedures in AppBuilder or Bootloader-core Software Component to prepare an unsigned GBL image with the required Secure Bootconfiguration for the application firmware.
3. Follow the procedures in Generate Key and Signing to generate the ECDSA-P256 Sign Key pair for Secure Boot. The key pairs for the bootloader certificate and application certificate (advanced) are required if using Signing for Certificate-Based Secure Boot.
4. Follow steps 1 to 2 in SE Manager Key Provisioning Platform Example if this HSEGBL Decryption Key option is selected. Use the Public Sign Key in step 3 and follow steps 1 to 3 in SE Manager Key Provisioning Platform Example to generate an **unsigned** image. Use this image to create an application GBL upgrade image file.
5. The original GBL (application Secure Boot is disabled) boots into the **unsigned** `SE Manager Key Provisioning Platform Example` after upgrading the application with the image file in step 4.
6. Follow steps 5 to 8 in SE Manager Key Provisioning Platform Example to install the Public Sign Key to SE OTP and GBL Decryption Key (optional) to HSE OTP. Press `SPACE` instead of `ENTER` in step 9 to **BYPASS** the programming of the Secure Boot configuration in SE OTP.

```
. Press ENTER to initialize SE OTP for secure boot configuration or press SPACE to skip.
. SE manager deinitialization... SL_STATUS_OK (cycles: 5 time: 0 us)
```

> Note:

Copyright © 2025 Silicon Laboratories. All rights reserved.

- Programming the Public Sign Key to the top page of the main flash (not included in this example) is required for the VSE device ECDSA-P256-SHA256 Secure Boot.
- Programming the GBL Decryption Key to the top page of the main flash (not included in this example) is required if the default storage option for GBL Decryption Key is selected and the **Require encrypted firmware upgrade files** option is enabled in step 2.

7. Follow the signing procedures in Signing for ECDSA-P256-SHA256 Secure Boot or Signing for Certificate-Based Secure Boot (Bootloader Image File section, skip the Secure Boot configuration for the bootloader) with the required key(s) generated in step 3 to sign the unsigned GBL image generated from step 2. Use this signed image to create a bootloader GBL upgrade image file.
8. Follow the signing procedures in Generate a GBL Upgrade Image File (Application Image File section) with the required key(s) generated in step 3 to sign the unsigned application image generated from step 4. Use this signed image to create an application GBL upgrade image file.

> **Note**: For the application bootloader with storage, the user can generate a single GBL upgrade image file for signed images from steps 7 and 8.

9. The Secure Boot in SE OTP is not yet enabled, so FSB does not verify the signature when upgrading to the signed GBL in step 7. The updated GBL (application Secure Boot enabled) verifies the signature when upgrading or booting to the signed `SE Manager Key Provisioning Platform Example` in step 8.
10. Follow steps 9 to 10 (use `SPACE` to skip previous steps for OTP key programming) in SE Manager Key Provisioning Platform Example to program the required Secure Boot configuration in SE OTP for signed GBL.
11. Update a **signed** custom application firmware to replace the signed `SE Manager Key Provisioning Platform Example` used for Secure Boot with RTSL upgrade.

> Note:
>
> - Refer to the "Enabling Secure Boot RTSL on Series 2 Devices" section (either Standalone Bootloaders or Application Bootloaders with Storage) in UG266/Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher for details.
> - The `SE Manager Key Provisioning Platform Example` used here is just for reference. The user can modify or write a new application to automate the processes for the Secure Boot with RTSL upgrade.
> - If the Require signed firmware upgrade files option is enabled in step 2, the GBL upgrade image files from steps 8 and 11 must be signed.
> - If the Require encrypted firmware upgrade files option is enabled in step 2, the GBL upgrade image files from steps 8 and 11 must be encrypted. And the GBL Decryption Key for the corresponding option in GBL must be in place.

Recover Devices when Secure Boot Fails

If a Secure Boot process fails (meaning firmware image at device starting address validation fails), the only way to recover is to flash a correctly signed image.

The following table describes the different debug lock scenarios on recovering the Secure Boot failure device.

Secure Debug	Device Erase	Debug Lock	State	Recover from Secure Boot Failure
Disabled	Enabled	Disabled	Unlock	Flash a correctly signed image.
Disabled	Enabled	Enabled	Standard debug lock	Flash a correctly signed image after standard debug unlocking the device.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Secure Debug	Device Erase	Debug Lock	State	Recover from Secure Boot Failure
Disabled	Disabled	Enabled	Permanent debug lock	There is no way to recover the device. Make sure the programmed image is correctly signed before locking the device.
Enabled	Disabled	Enabled	Secure debug lock	Flash a correctly signed image after secure debug unlocking the device.

> **Note**: The error code in the **Boot status** of examples below depends on boot failure caused by the host image (GBL).

The following procedures describe how to recover the Secure Boot failure device from the lock states below.

- Unlocked
- Standard debug locked
- Secure debug locked

1. Follow the procedure in Signing for ECDSA-P256-SHA256 Secure Boot or Signing for Certificate-Based Secure Boot to generate a correctly signed GBL.
2. **(Unlocked)** Run the `security status` command to get the boot status.

```
commander security status --device EFR32MG21A010F1024 --serialno 440048205
```

```
SE Firmware version : 1.2.9
Serial number       : 00000000000000014b457fffe045afd
Debug lock          : Disabled
Device erase        : Enabled
Secure debug unlock : Disabled
Tamper status       : Not OK
Secure boot         : Enabled
Boot status         : 0x12 - Failed: Error while checking signature of host firmware
DONE
```

Run the `flash` command to flash the correctly signed image (like `bootloader-uart-xmodem.s37`). If a failed Secure Boot is detected, the device will be erased before flashing the new image.

```
commander flash bootloader-uart-xmodem.s37 --device EFR32MG21A010F1024 --serialno 440048205
```

```
WARNING: Failed secure boot detected. Issuing a mass erase before flashing to recover the device...
Parsing file bootloader-uart-xmodem.s37...
Writing 16384 bytes starting at address 0x00000000
Comparing range 0x00000000 - 0x00003FFF (16 KiB)
Programming range 0x00000000 - 0x00001FFF (8 KiB)
Programming range 0x00002000 - 0x00003FFF (8 KiB)
DONE
```

3. **(Standard debug locked)** Run the `security status` command to get the boot status.

```
commander security status --device EFR32MG21A010F1024 --serialno 440048205
```

```
SE Firmware version : 1.2.9
Serial number       : 00000000000000014b457fffe045afd
Debug lock          : Enabled
Device erase        : Enabled
Secure debug unlock : Disabled
Tamper status       : Not OK
Secure boot         : Enabled
Boot status         : 0x12 - Failed: Error while checking signature of host firmware
DONE
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Run the `security erasedevice` command to unlock the device.

```
commander security erasedevice --device EFR32MG21A010F1024 --serialno 440048205
```

```
Successfully erased device
DONE
```

> **Note**: Issue a power-on or pin reset to complete the unlock process.

Run the `flash` command to flash the correctly signed image (like `bootloader-uart-xmodem.s37`). If a failed Secure Boot is detected, the device will be erased before flashing the new image.

```
commander `flash` bootloader-uart-xmodem.s37 --device EFR32MG21A010F1024 --serialno 440048205
```

```
WARNING: Failed secure boot detected. Issuing a mass erase before flashing to recover the device...
Parsing file bootloader-uart-xmodem.s37...
Writing 16384 bytes starting at address 0x00000000
Comparing range 0x00000000 - 0x00003FFF (16 KiB)
Programming range 0x00000000 - 0x00001FFF (8 KiB)
Programming range 0x00002000 - 0x00003FFF (8 KiB)
DONE
```

4. **(Secure debug locked)** Run the `security status` command to get the boot status.

```
commander security status --device EFR32MG21A010F1024 --serialno 440048205
```

```
SE Firmware version : 1.2.9
Serial number       : 000000000000000000d6ffffe0a3a5f
Debug lock          : Enabled
Device erase        : Disabled
Secure debug unlock : Enabled
Tamper status       : Not OK
Secure boot         : Enabled
Boot status         : 0x12 - Failed: Error while checking signature of host firmware
DONE
```

Run the `security unlock` command to unlock the device with the debug unlock token.

```
commander security unlock --device EFR32MG21A010F1024 --serialno 440048205
```

```
Unlocking with unlock payload:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_000000000000000000d6ffffe0a3a5f/challenge_020fc3c

Secure debug successfully unlocked
DONE
```

Run the `flash` command with the **--noreset** option to flash the correctly signed image (like `bootloader-uart-xmodem.s37`).

```
commander flash --noreset bootloader-uart-xmodem.s37 --device EFR32MG21A010F1024 --serialno 440048205
```

```
Parsing file bootloader-uart-xmodem.s37...
Writing 16384 bytes starting at address 0x00000000
Comparing range 0x00000000 - 0x00003FFF (16 KiB)
Erasing range 0x00000000 - 0x00003FFF (2 sectors, 16 KiB)
Programming range 0x00000000 - 0x00001FFF (8 KiB)
Programming range 0x00002000 - 0x00003FFF (8 KiB)
DONE
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

> **Note**: The `--noreset` option prevents the device from returning to the secure debug lock state before flashing.

5. Run the `security status` command to check the boot status. The example below is an unlocked device.

```
commander security status --device EFR32MG21A010F1024 --serialno 440048205
```

```
SE Firmware version : 1.2.9
Serial number       : 000000000000000014b457fffe045afd
Debug lock          : Disabled
Device erase        : Enabled
Secure debug unlock : Disabled
Tamper status       : OK
Secure boot         : Enabled
Boot status         : 0x20 - OK
DONE
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Debugging On Secure Boot Enabled Device

Debugging on Secure Boot Enabled Device

Assume a correctly signed GBL image has been programmed to the device. Follow the procedures in Generate an Unsigned Application Image to generate an unsigned application image for the GBL.

The Windows environment variable PATH should include the folder (`C:\SiliconLabs\SimplicityStudio\v5\developer\adapter_packs\commander`) that locates the `commander.exe` of Simplicity Commander.

The following sections describe how to debug an application firmware with Simplicity IDE, or IAR on a Secure Boot enabled device.

Simplicity IDE

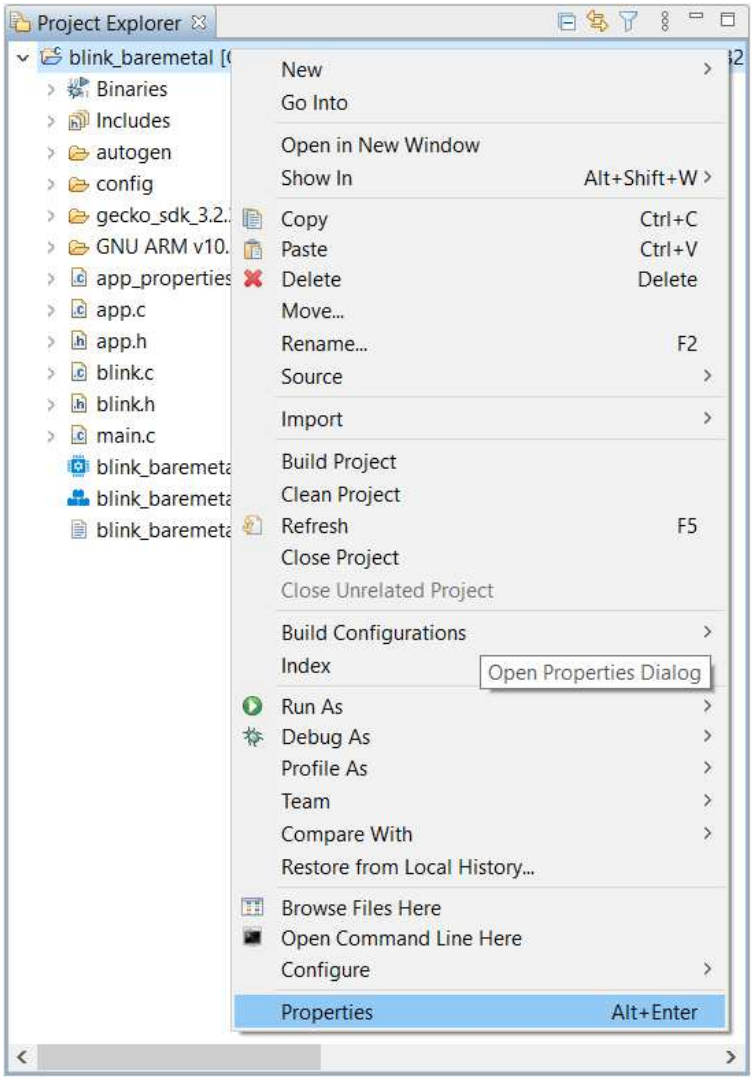
This application note uses Simplicity Studio v5.2.3.1. The procedures and pictures may be different for the other versions of Simplicity Studio 5.

1. The Simplicity IDE creates a folder below (`\<NAME>` is the Windows User Name on PC) in Windows when building the unsigned application image.
   `C:\Users\<NAME>\SimplicityStudio\v5_workspace\blink_baremetal\GNU ARM v10.2.1 - Default`
2. Follow the procedures in Signing for ECDSA-P256-SHA256 Secure Boot or Signing for Certificate-Based Secure Boot to create a batch file (Windows) to sign the unsigned application image and then flash it to the device. This application note uses ECDSA-P256-SHA256 Secure Boot (Using Simplicity Commander) as an example to create a `secure_boot_debug.bat` file below.

   ```
   commander convert blink_baremetal.s37 --secureboot --keyfile sign_key.pem --verify sign_pubkey.pem
   --outfile blink_baremetal.s37
   commander flash blink_baremetal.s37
   ```

3. Copy the batch file in step 2 and files (`sign_key.pem` and `sign_pubkey.pem` in this example) specified in `secure_boot_debug.bat` to the folder in step 1.
4. Right-click the project in the **Project Explorer** window, and then click **Properties** to open the properties dialog.

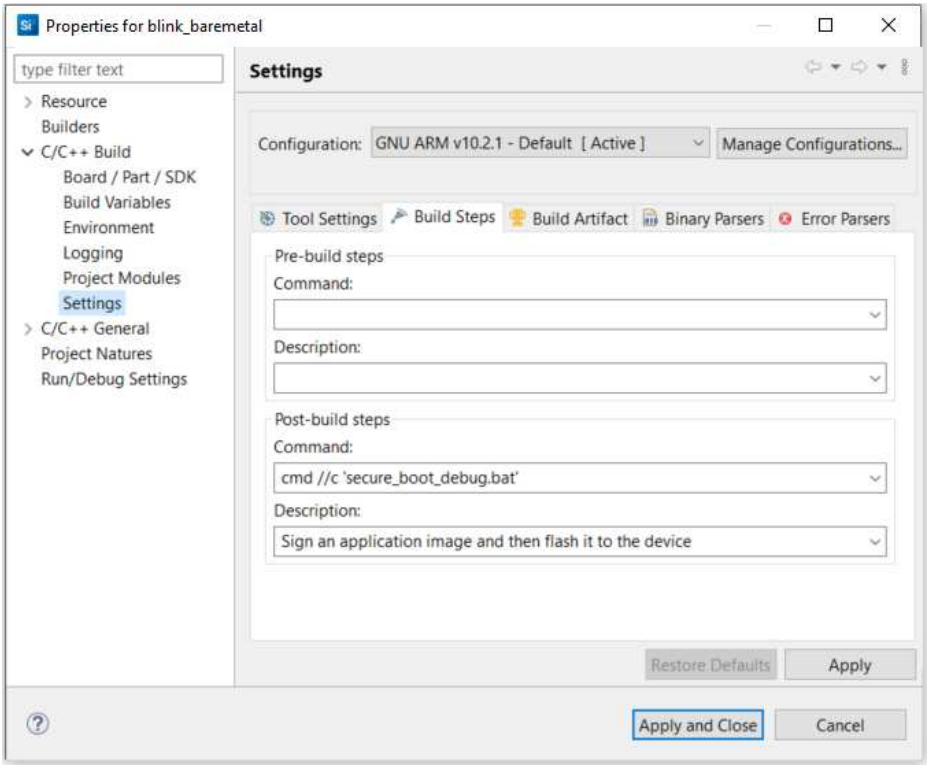
Copyright © 2025 Silicon Laboratories. All rights reserved.



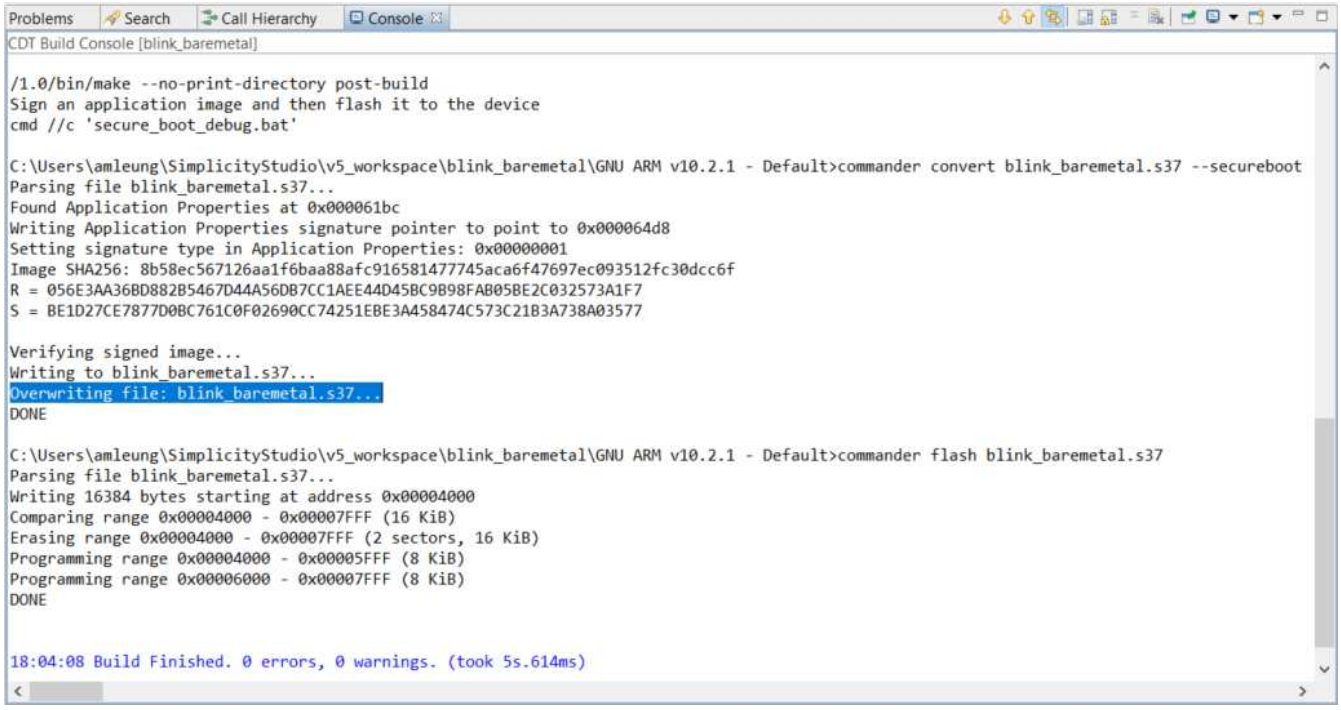
5. Select **C/C++ Build→Settings→Build Steps**. Enter the phrase below to the **Command:** box under the **Post-build steps** (enter text to **Description:** box is optional) to run the batch file as a post-build action. Click **[Apply and Close]** to exit.

```
cmd //c 'secure_boot_debug.bat'
```

Copyright © 2025 Silicon Laboratories. All rights reserved.



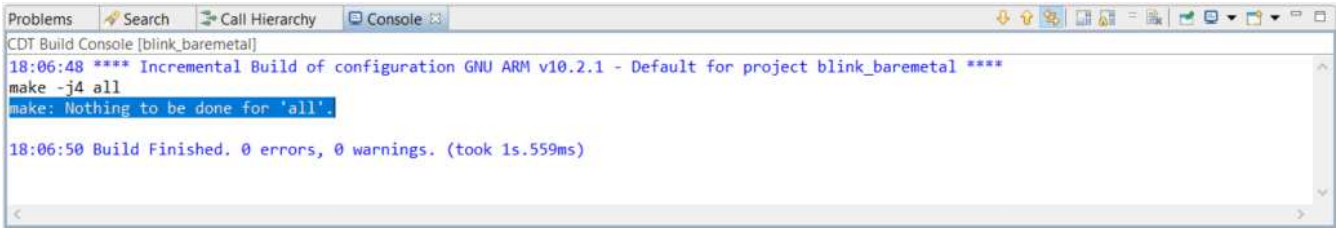
6. After building the project, the batch file in the **Post-build steps** overwrites the unsigned application image with the signed application image.



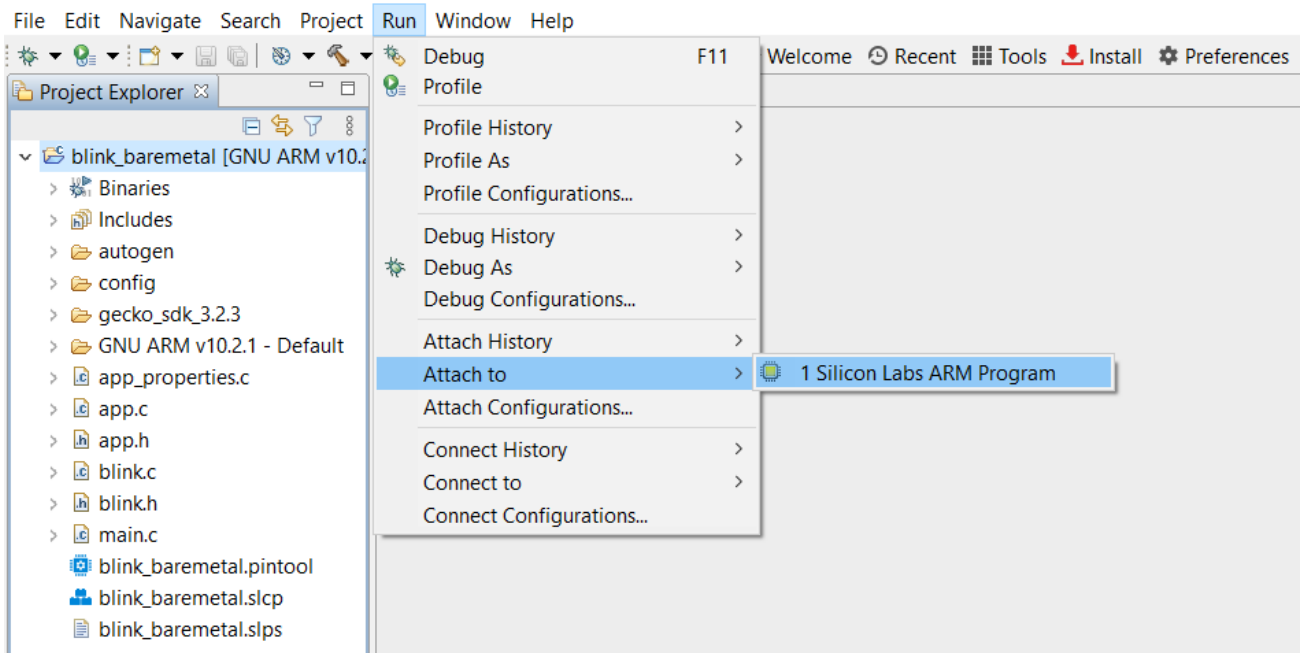
> **Note**: If the project is already up-to-date, it will not invoke the **Post-build steps** in step 5 to run the batch file. Use a dummy edit (add space or newline) on one of the source files in the project to trigger the build

Copyright © 2025 Silicon Laboratories. All rights reserved.

> action.



7. The application starts to run if no error in step 6.
8. Select the project in the **Project Explorer** window, click **Run→Attach to→1 Silicon Labs ARM Program** to attach to the running target for debugging on the signed application image.

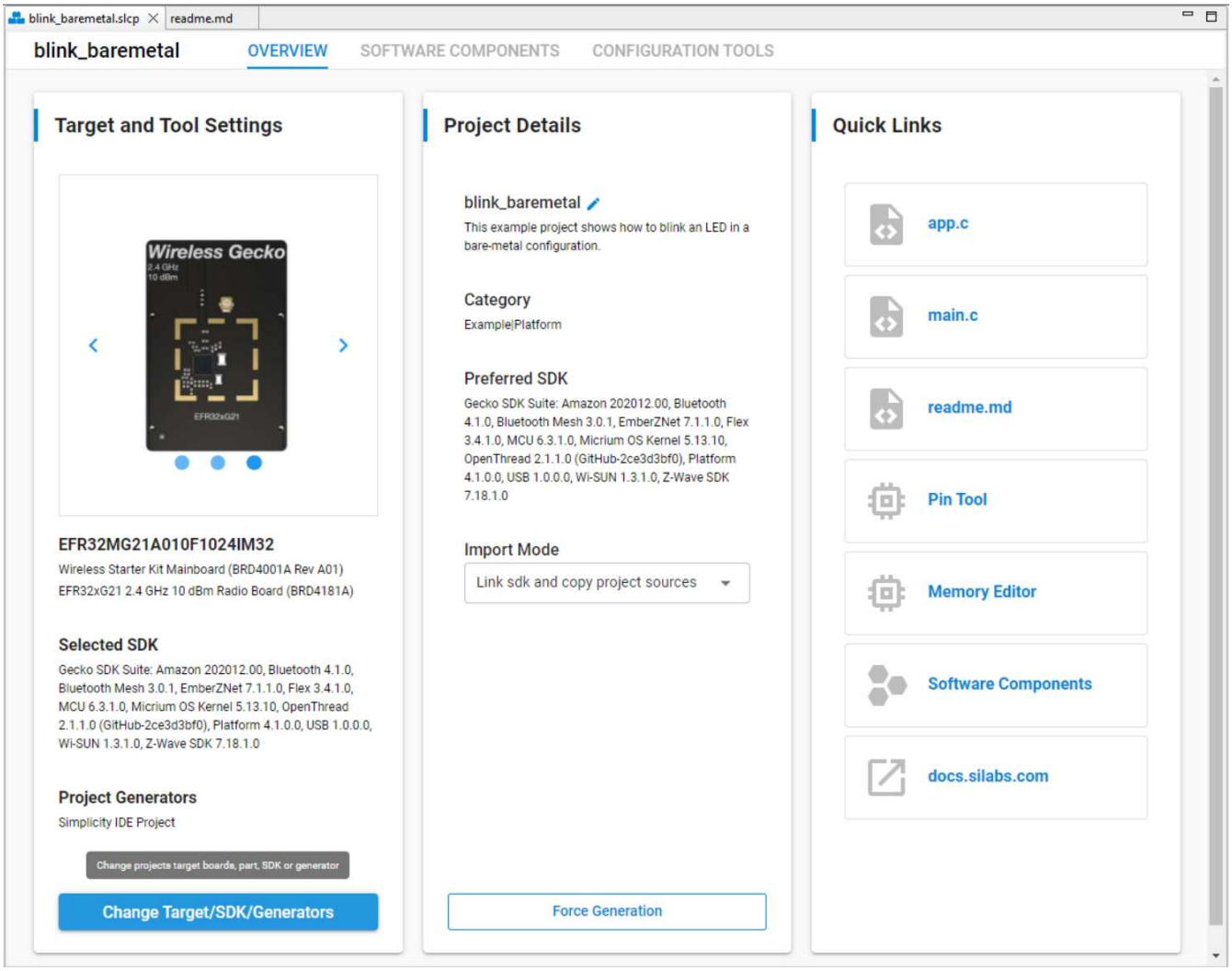


IAR

This section uses Simplicity Studio v5.4.2.0 and IAR v9.20.4. The procedures and pictures may be different for the other versions of Simplicity Studio 5 and IAR.

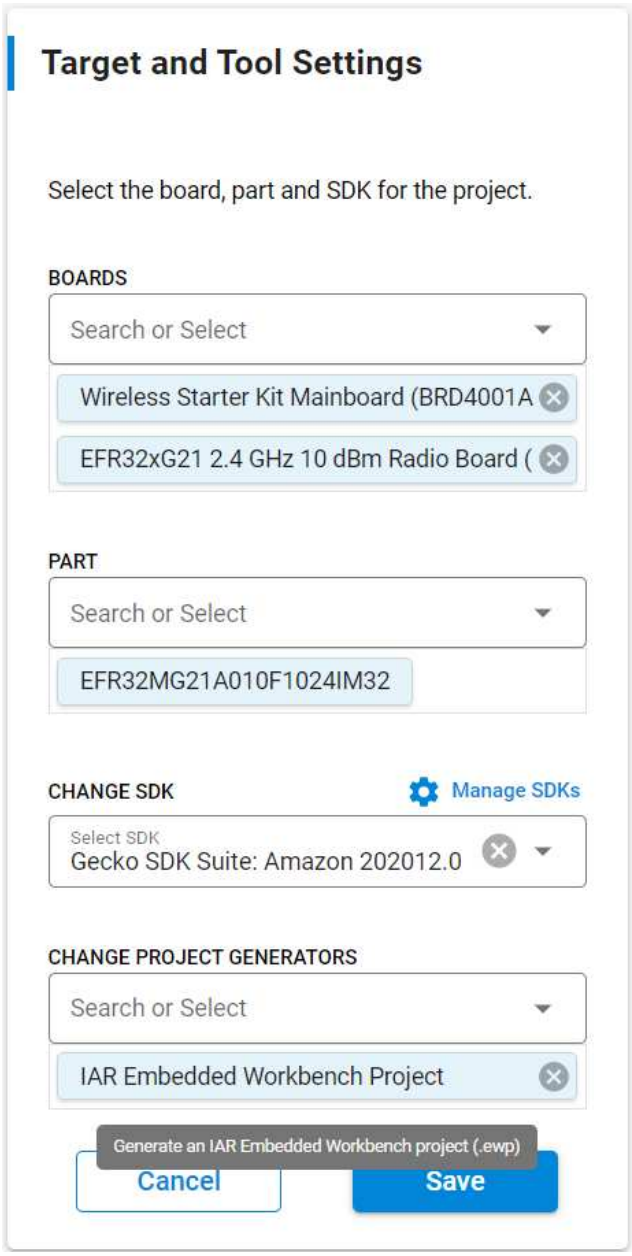
1. The **Overview** tab shows the **Target and Tool Settings** card on the left side. Scroll down if necessary and click **[Change Target/SDK/Generators]**.

Copyright © 2025 Silicon Laboratories. All rights reserved.



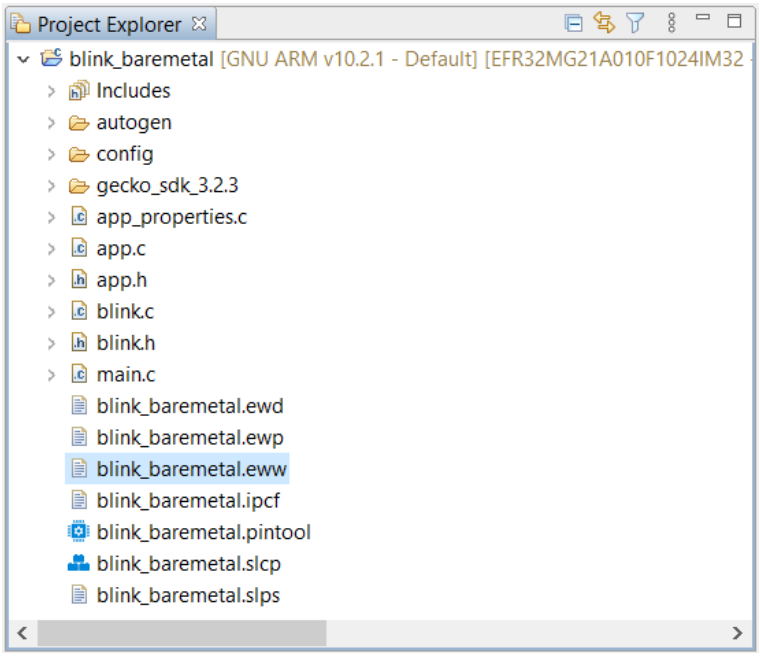
2. Drop down the **CHANGE PROJECT GENERATORS** list and select **IAR Embedded Workbench Project**. Click **[Save]** to generate an IAR project.

Copyright © 2025 Silicon Laboratories. All rights reserved.



3. Double click the IAR workspace file (blink_baremetal.eww) in the **Project Explorer** window to open the IAR project. The IAR creates a folder below (<NAME> is the Windows User Name on PC) in Windows to store the compiled image. C:\Users<NAME>\SimplicityStudio\v5_workspace\blink_baremetal\ewarm-iar\exe

Copyright © 2025 Silicon Laboratories. All rights reserved.

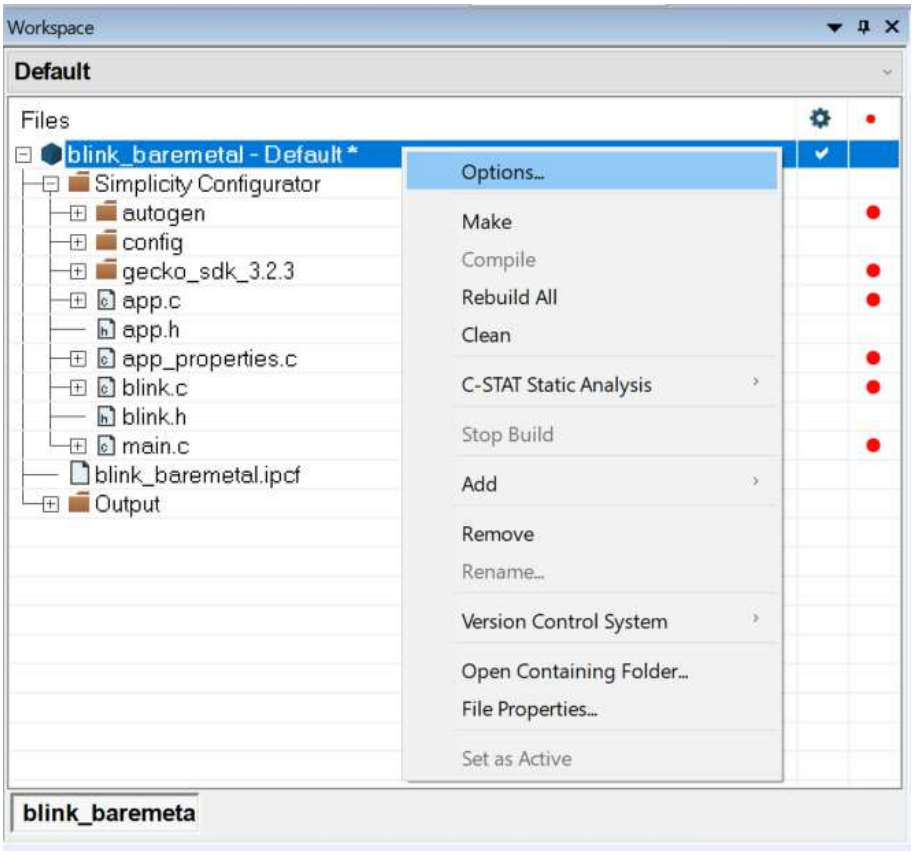


4. Follow the procedures in Signing for ECDSA-P256-SHA256 Secure Boot or Signing for Certificate-Based Secure Boot to create a batch file (Windows) to sign the unsigned application image. This application note uses ECDSA-P256-SHA256 Secure Boot (Using Simplicity Commander) as an example to create a secure_boot_debug.bat file below.

```
cd C:\Users\<NAME>\SimplicityStudio\v5_workspace\blink_baremetal\ewarm-iar\exe
commander convert blink_baremetal.s37 --secureboot --keyfile sign_key.pem --verify sign_pubkey.pem
--outfile blink_baremetal.s37
```C
```

5. Copy the batch file in step 4 and files (sign_key.pem and sign_pubkey.pem in this example) specified in secure_boot_debug.bat to the folder in step 3.
6. Right-click the project in the workspace, and then click **Options...**.

Copyright © 2025 Silicon Laboratories. All rights reserved.
```

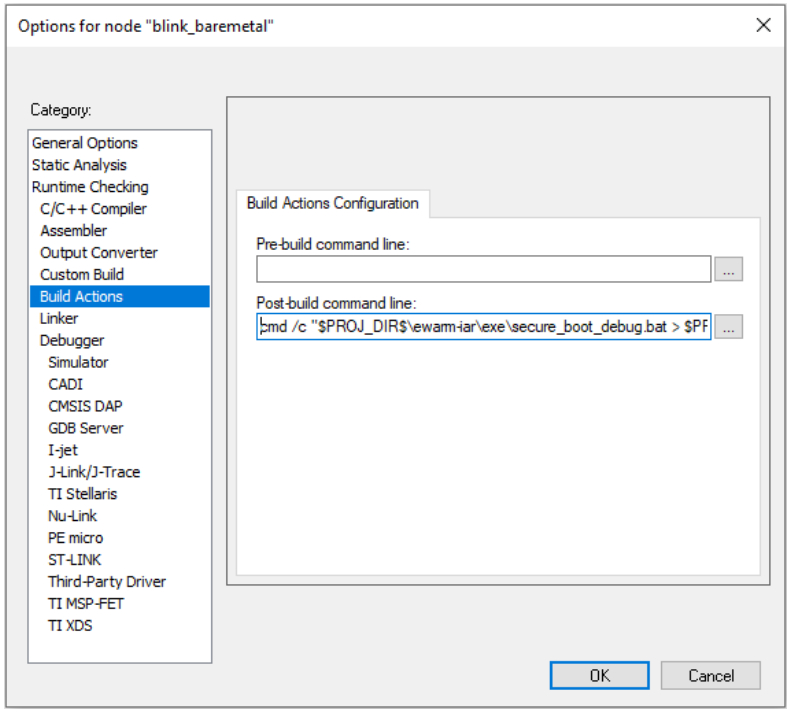


> **Note**: For GSDK v3.2 and lower, the app_properties.c is manually added to the IAR project.

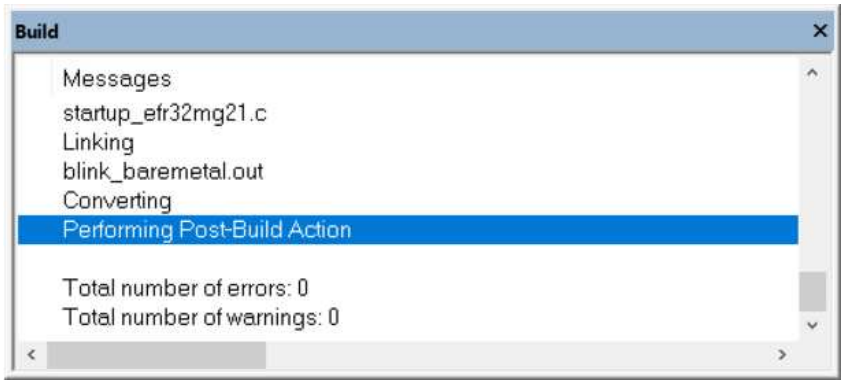
7. Click **Build Actions** to open the **Build Actions Configuration** dialog box. Enter the phrase below to the **Post-build command line:** box to run the batch file as a post-build action. Click **[OK]** to exit.

```
cmd /c "$PROJ_DIR$\ewarm-iar\exe\secure_boot_debug.bat > $PROJ_DIR$\log.txt 2>&1"
```

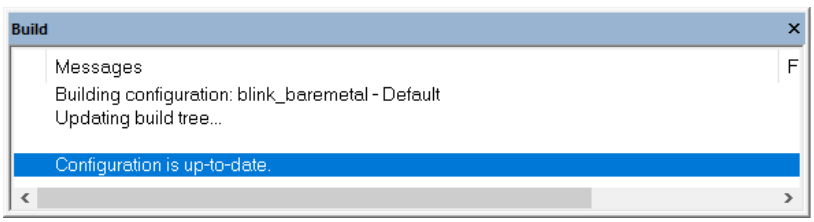
Copyright © 2025 Silicon Laboratories. All rights reserved.



8. After building the project, the batch file in the **Post-build command** overwrites the unsigned application image with the signed application image.



> **Note**: If the project is already up-to-date, it will not invoke the **Post-build command** in step 7 to run the batch file. Use a dummy edit (add space or newline) on one of the source files in the project to trigger the build action.



Copyright © 2025 Silicon Laboratories. All rights reserved.

The > $PROJ_DIR$\log.txt 2>&1 redirects the batch file output to the log.txt file in the IAR project folder.



10. If no error in step 8, click the  icon to start debugging on the signed application image.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Failure Analysis

Failure Analysis

The following table describes the different scenarios when returning a Series 2 device to Silicon Labs for failure analysis.

State	Secure Boot Disabled	Secure Boot Enabled (2)
Standard debug unlock	Device erase is not necessary for failure analysis.	Device erase is not necessary, but a correctly signed image is required to perform failure analysis.
Standard debug lock	Device erase is required to perform failure analysis.	Require device erase and correctly signed image to perform failure analysis.
Permanent debug lock	Cannot perform failure analysis.	Cannot perform failure analysis.
Secure debug lock (1)	Require debug unlock token to perform failure analysis.	Require debug unlock token and correctly signed image to perform failure analysis.

Notes:

1. Follow the procedures in Series 2 Secure Debug section *Secure Debug Unlock and Roll Challenge - Simplicity Commander* to generate a valid debug unlock token for each device returned to Silicon Labs for failure analysis.
2. Secure boot enabled devices, especially with secure boot failure, may limit Silicon Labs' ability to determine the root cause of failure.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Transitioning to the Updated Gecko Bootloader in GSDK 4.0 and Higher

Transitioning to the Updated Gecko Bootloader in GSDK 4.0 and Higher

NOTE: This section replaces *AN1326: Transitioning to the Updated Gecko Bootloader in GSDK 4.0 and Higher*. Further updates to this application note will be provided here

Gecko Bootloader v2.x, introduced in GSDK 4.0, contains a number of changes compared to Gecko Bootloader v1.x. Many of these changes are due to an underlying framework redesign that results in an improved developer experience in Simplicity Studio 5 as well as enhanced compatibility within the GSDK. This document describes the differences between the versions, including how to configure the new Gecko Bootloader within Simplicity Studio 5.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Introduction

Introduction

Gecko Software Development Kit suite (GSDK) version 3 (GSDK v3.0) introduced a new underlying Gecko Platform architecture based on components. Beginning with GSDK 4.0, the Gecko Bootloader now uses this underlying architecture. With Simplicity Studio 5 (SSv5) and GSDK 4.0, developers working with Gecko Bootloader will benefit from the following component-based project configuration features:

- Search and filter to find and discover software components that work with the target device
- Automatically pull in all component dependencies and initialization code
- Configurable software components including peripheral inits, drivers, middleware, and stacks
- All configuration settings are in C header files for usage outside of Simplicity Studio
- Configuration validation to alert developers to errors or issues
- Easily manage all project source via git or other SCM tools
- Managed migration to future component and SDK versions
- Simplified transitions from Silicon Labs development kits to custom hardware

Other features of the SSv5/GSDK 4.x development environment include:

- Project source management options (link to SDK sources or copy all contents to user folder)
- Graphical pin configuration through the Pin Tool
- Redesigned Radio Configurator with a fresh UI that's more intuitive for single- and multi-PHY customization
- Iterative development (configure components, edit sources, compile, debug) using SSv5 configuration tools and third-party IDEs
- GNU makefiles as a build option

This document summarizes the differences between the Gecko Bootloader v2.x in GSDK 4.0 and earlier AppBuilder-based versions. These differences include:

- Differences between AppBuilder and the new component-based Project Configurator
- Comparison of the new Project Configurator components with AppBuilder plugins
- Features of the Pin Tool, now used instead of Hardware Configurator
- Other differences including:
  - Linker file
  - Additional Macros
  - Postbuild steps
  - Main bootloader in main flash
  - Callbacks
  - Storage Slots
  - App Properties

Copyright © 2025 Silicon Laboratories. All rights reserved.

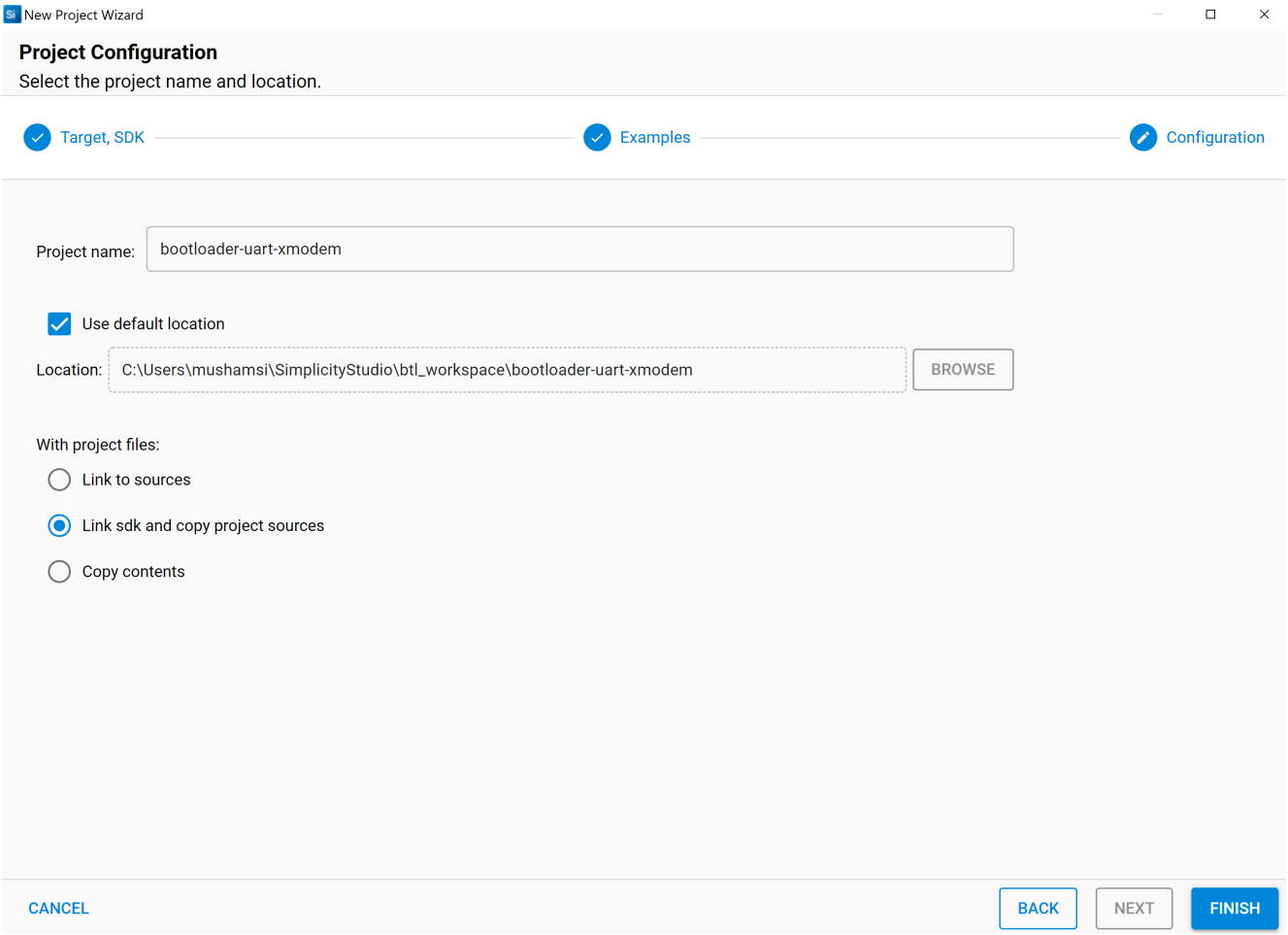
About Projects

About Projects

Creating a Project

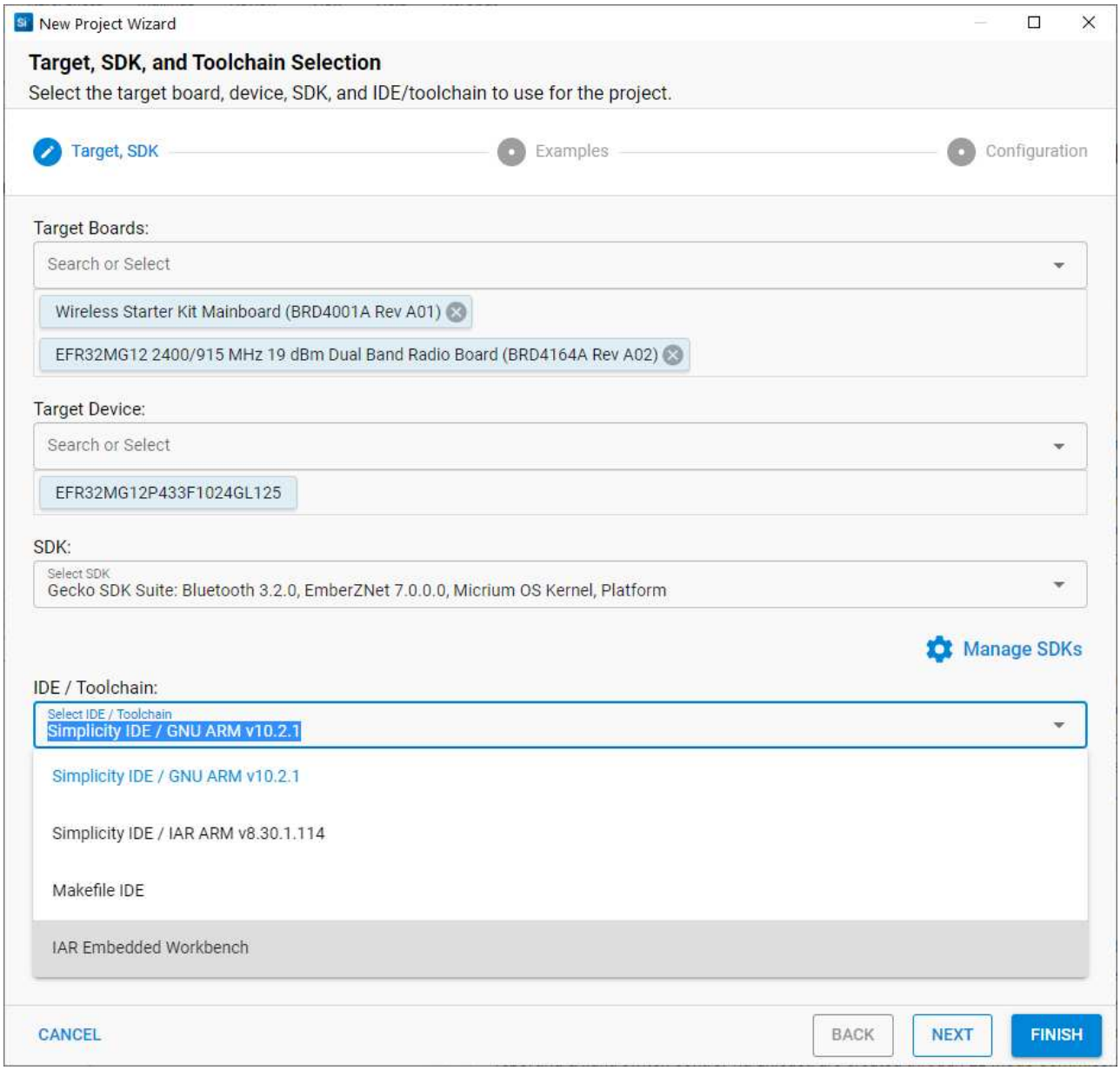
If you have used earlier versions of SSv5, initial project creation is unchanged. If you are unfamiliar with project creation in SSv5, see the Simplicity Studio 5 User's Guide. This section highlights the differences with AppBuilder project creation and configuration.

Once you have selected an example project, an initial configuration dialog opens. Note that the Project Location, which in AppBuilder is shown on the General tab, is defined here.



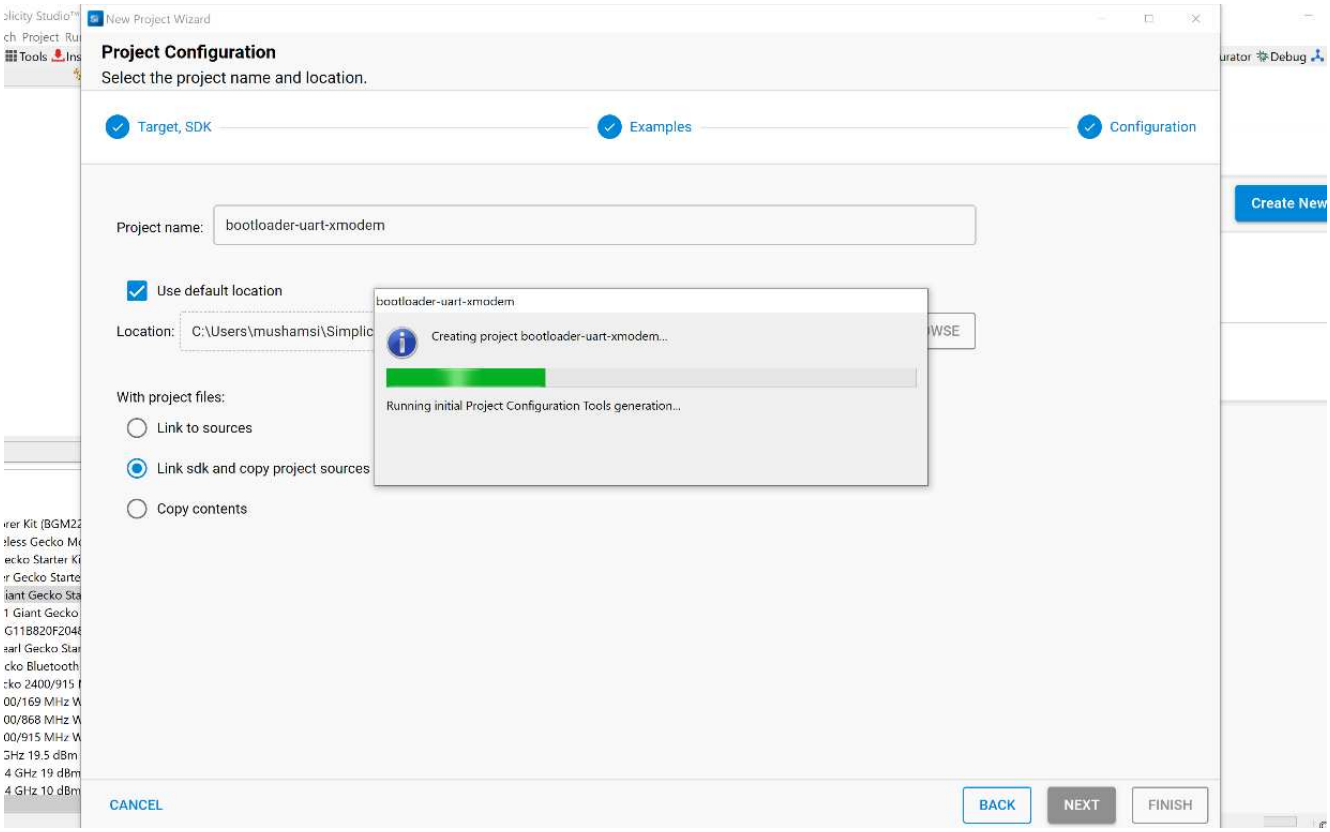
In SSv5, the compiler to use is specified when you create a project and, after that, is difficult to change (see Simplicity Studio 5 User's Guide Tips & Tricks for a procedure to convert an existing project from GCC to IAR). If you do not want to use the default compiler (GCC unless otherwise specified), click **BACK**, and change it on the first Project Configuration dialog.

Copyright © 2025 Silicon Laboratories. All rights reserved.



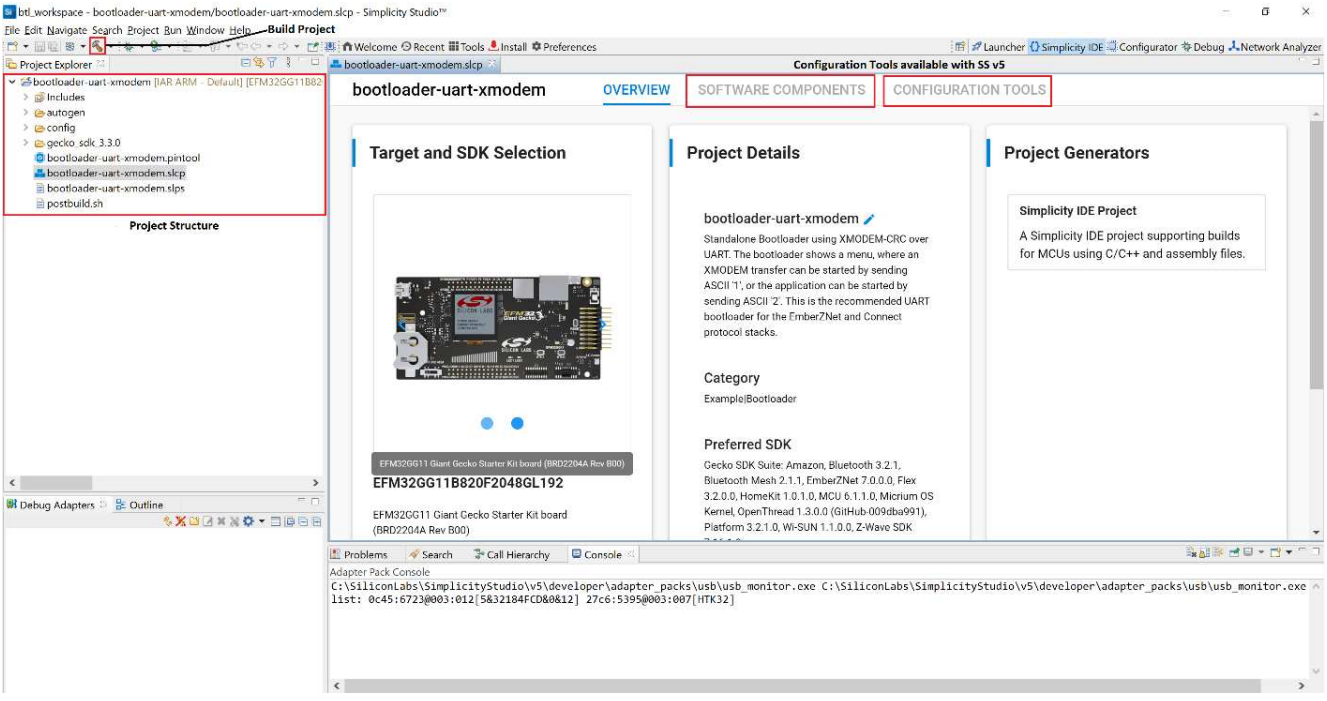
Click **FINISH** and the project will be generated.

Copyright © 2025 Silicon Laboratories. All rights reserved.



Project File Structure

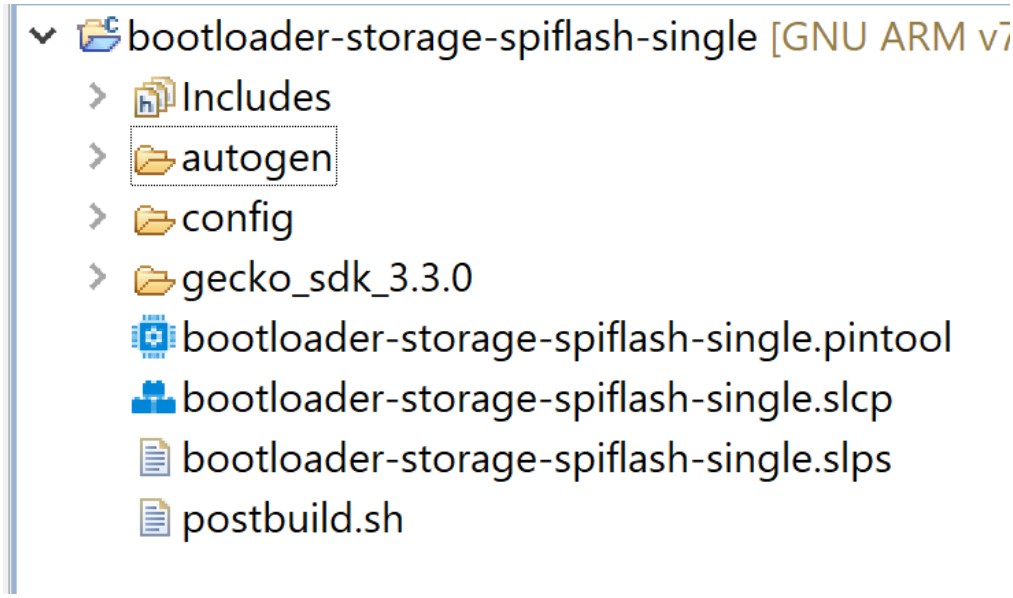
On project generation, the required project files and folders are created.



Projects always include the following parts:

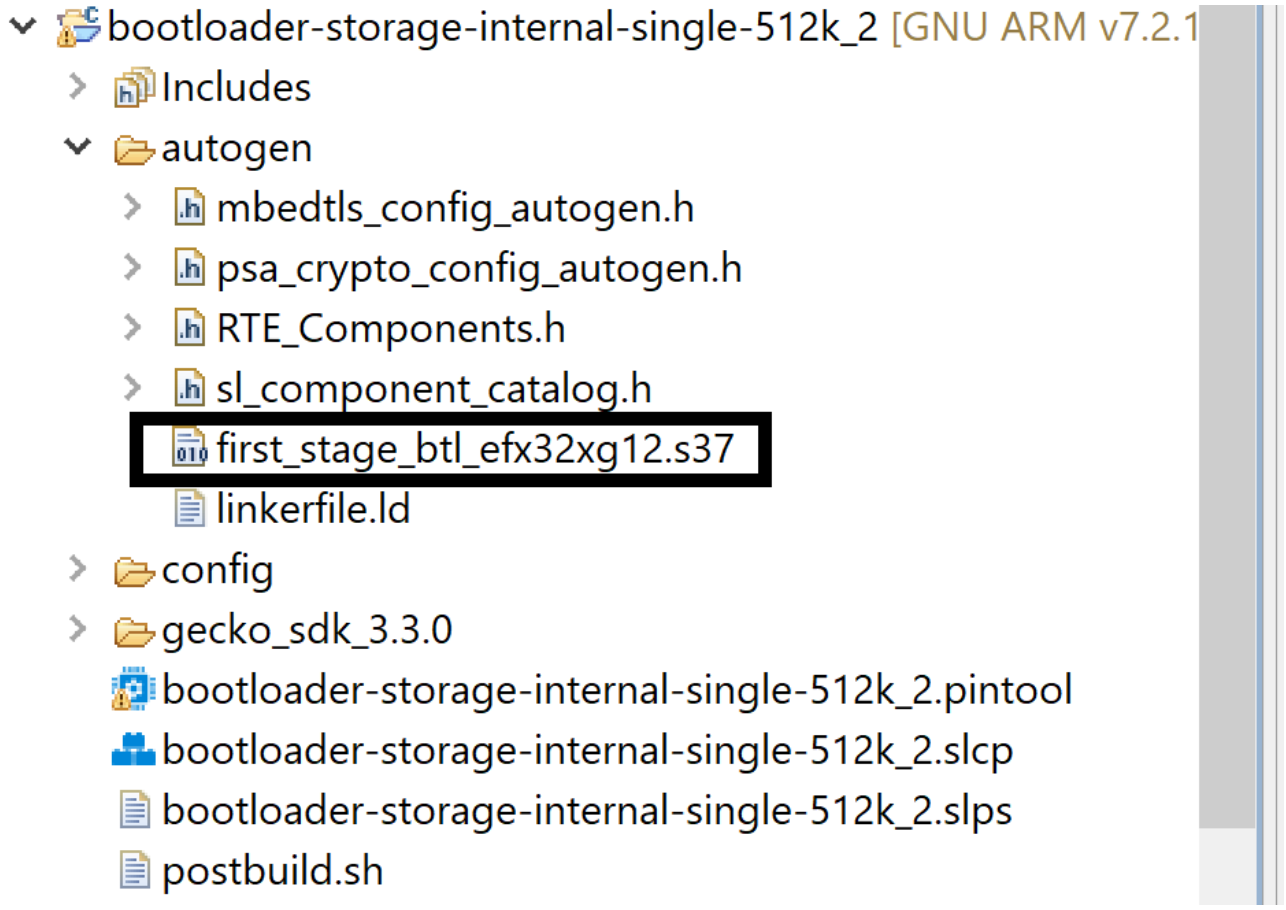
Copyright © 2025 Silicon Laboratories. All rights reserved.

- **autogen** folder: Only the autogen folder includes generated code. It includes configuration (mbedtls_config_autogen.h), init code, the linker script, and other generated code used by components, like the command descriptors for the CLI interface.
- **config** folder: Component configuration headers are located in this folder. These can be edited with the Simplicity IDE Component Editor, but directly editing the header file is also possible. The Component Editor is available through the Project Configurator's **Configure** control, available only for configurable components.
- **gecko_sdk** folder (with version number): Contains source and binary files added by components.
- **files in the root folder**: Only the application specific files should be in the root folder, including source files, the project configurator (*.slcp*) file and the Pin Tool *(.pintool*) file. For more details, see Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher.



The first stage bootloader binary file is copied to the **autogen** folder at the time of project generation. The correct first stage binary file is located in accordance with the target hardware and placed in the **autogen,** which is later used by the postbuild script to generate a combined bootloader binary. The **autogen** folder is shown below with the first stage bootloader binary.

Copyright © 2025 Silicon Laboratories. All rights reserved.



Configuring a Project

Most project configuration can be done through tools in Simplicity Studio. These tools provide the functionality previously provided by AppBuilder.

- **Project Configurator**: the top-level project configuration tool through which you install and uninstall components and change other project parameters. It also provides access to other configuration tools.
- **Component Editor**: provides access to the configurable parameters of an individual component.
- **Pin Tool**: configures peripherals.

Once you create the project, the Project Configurator tabbed interface is presented, including:

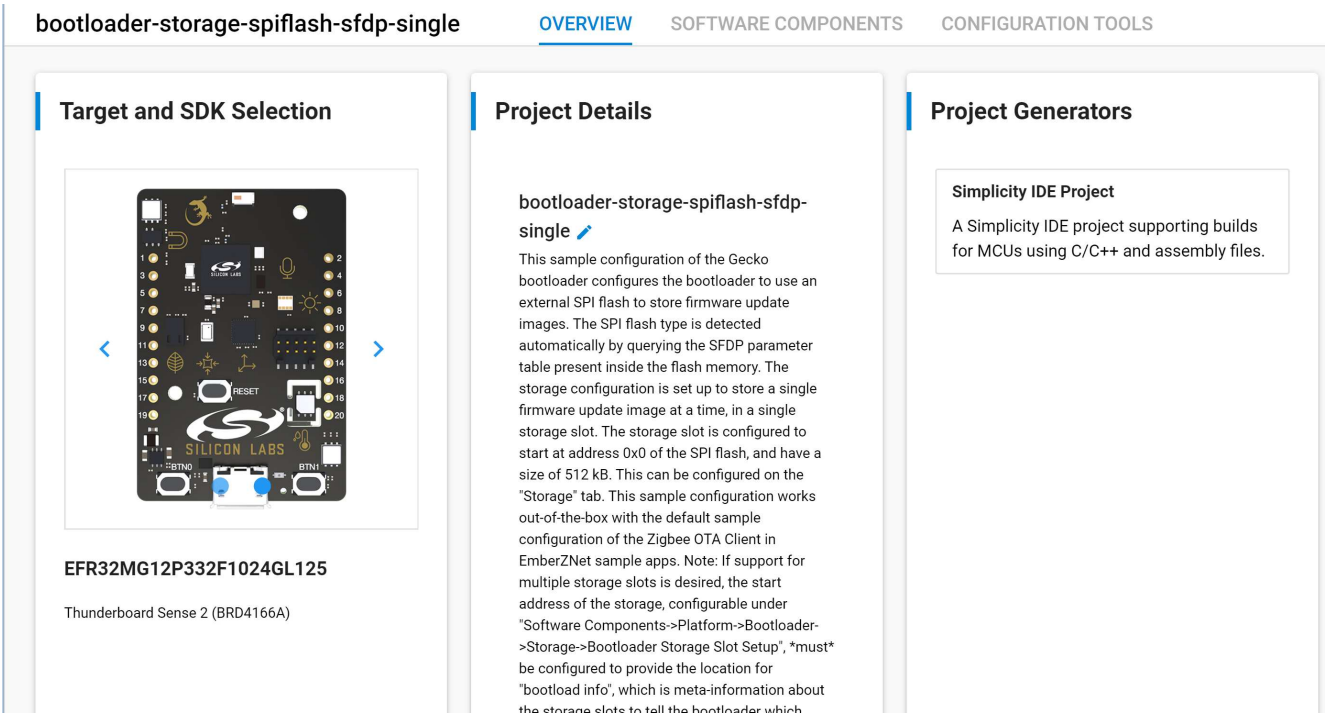
- Overview
- Software Components
- Configuration Tools

You can always re-open the Project Configurator by double-clicking the <projectname>.slcp file. For details on using the Project Configurator and Component Editor, see the Simplicity Studio 5 User's Guide.

OVERVIEW Tab

The OVERVIEW tab, like AppBuilder's General tab, shows the target part information, and a description of the project.

Copyright © 2025 Silicon Laboratories. All rights reserved.

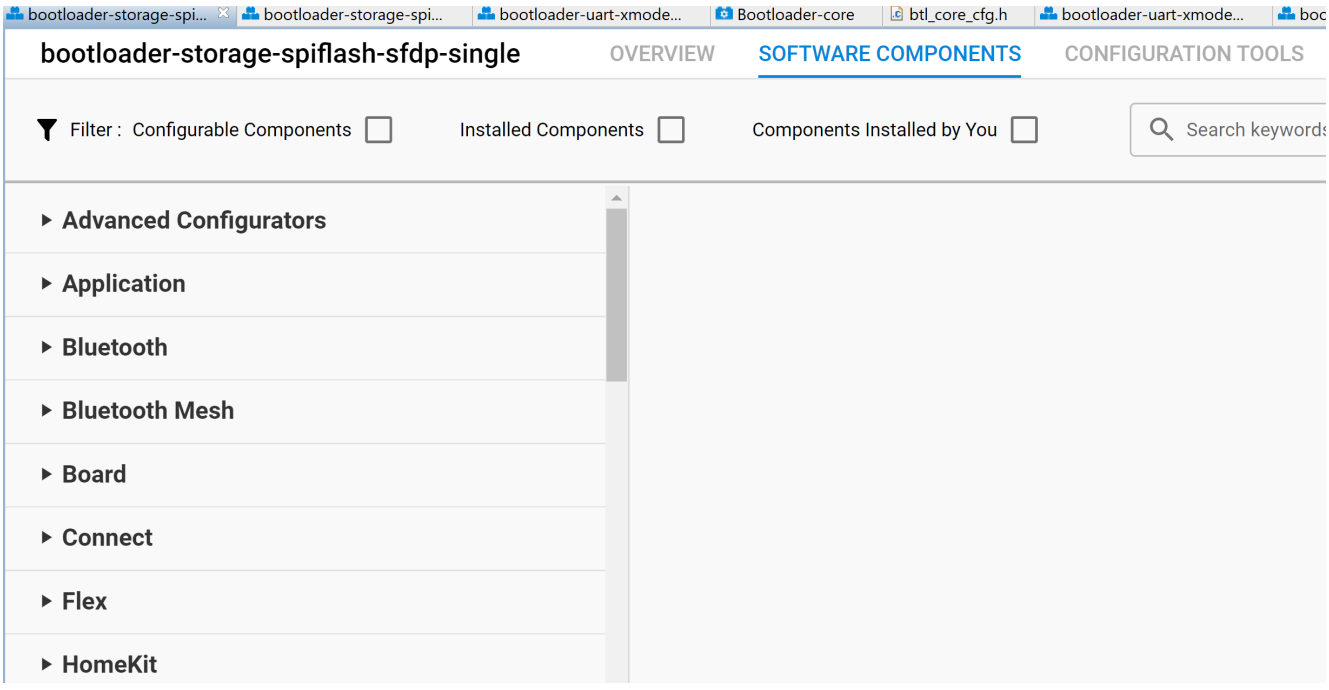


Note that there is no **Generate** control. Changes made through Project Configurator are auto-saved and the project files are auto-generated. The Force Generation control on the Project Details card is provided for use only in the event of a system problem, where auto-generation fails.

The Project Generators interface controls generation of additional files for import into other IDEs. It does not change the compiler used to build the application image.

SOFTWARE COMPONENTS Tab

The SOFTWARE COMPONENTS tab shows the available components and those that are already installed in the example code.

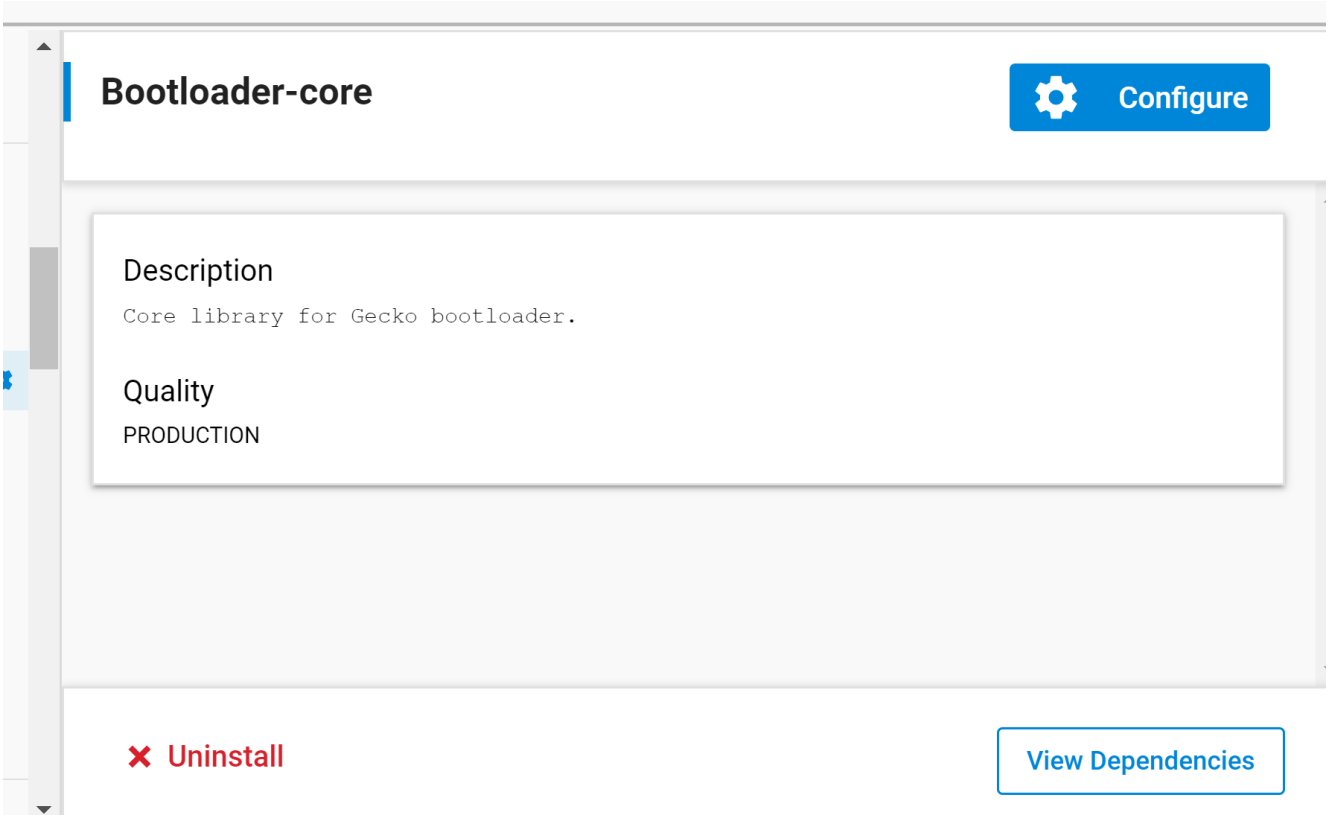


Copyright © 2025 Silicon Laboratories. All rights reserved.

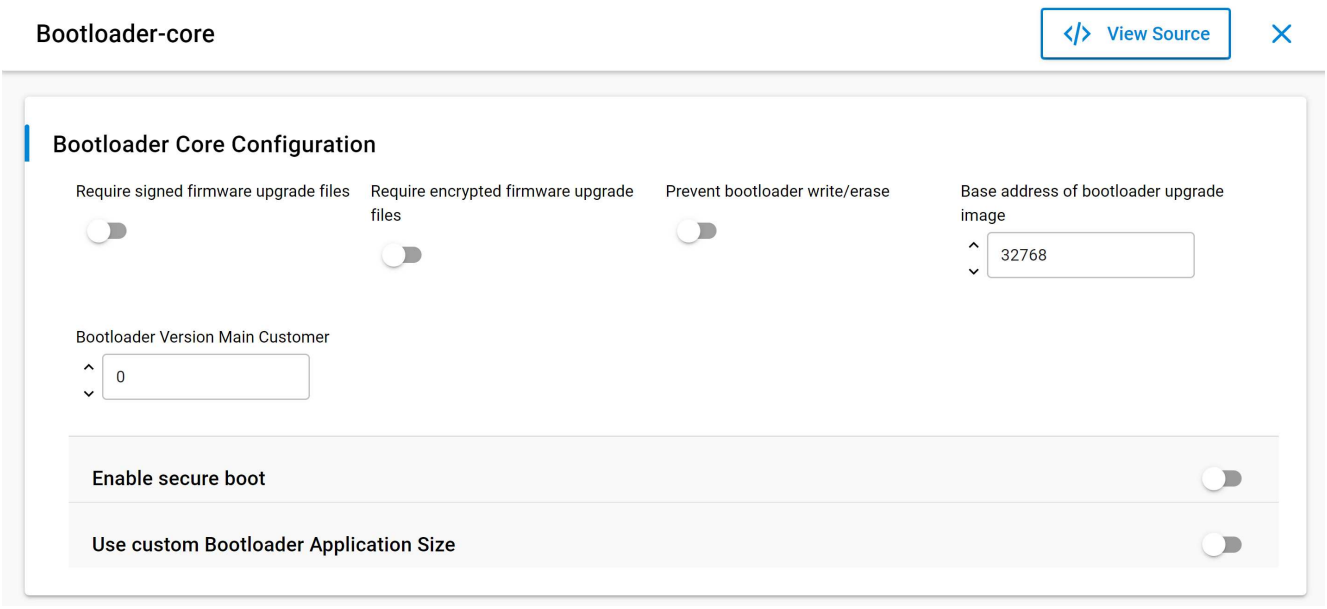
Component groupings are subject to change in SDK releases, as new components are added and organization is updated. The most reliable way to find a specific component is to search for it. You can also filter the displayed components using the checkboxes at the top.

Installed components have a circled checkmark to the left. Configurable components have a gear icon to the right.

Select a component to see details about it. Click **Install** to install the component or **Uninstall** to remove it. If it is configurable, a **Configure** control is shown in the upper right.



Click that control or the gear icon next to an installed component name to open the Component Editor in a new tab.

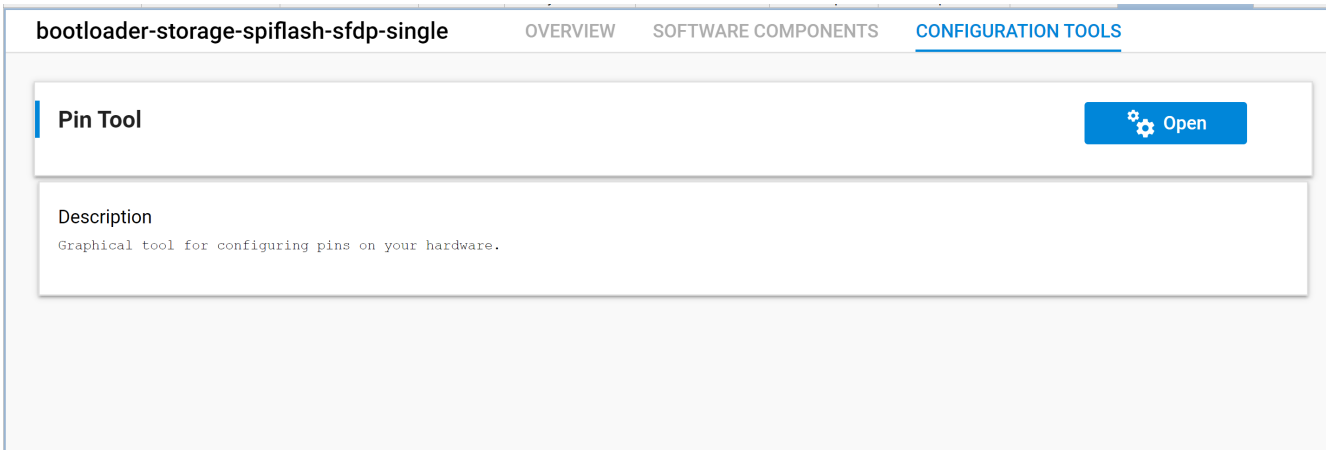


Copyright © 2025 Silicon Laboratories. All rights reserved.

Save any changes to trigger file generation.

CONFIGURATION TOOLS Tab

The CONFIGURATION TOOLS tab is a quick way of opening useful tools for project development.



Copyright © 2025 Silicon Laboratories. All rights reserved.

Components

Components

This section maps the various Gecko Bootloader components to the previously available plugins (in the AppBuilder workflow). Although the basic subgroups might remain the same, specific component locations are subject to change across various GSDK versions. Therefore, it is good practice to search for individual components by their names rather than the specific subgroups that they belong to.

AppBuilder Plugin	Simplicity Studio v5 Component	Additional Comments
Communication		
BGAPI UART DFU	BGAPI UART DFU	
EZSP-SPI	EZSP-SPI	
UART XMODEM	UART XMODEM	
XMODEM Parser	XMODEM Parser	
Core		
Application Upgrade Version Check	Application Upgrade Version Check	
Bootloader Core	Bootloader Core	
GBL Compression (LZ4)	GBL Compression (LZ4)	
GBL Compression (LZMA)	GBL Compression (LZMA)	
Image Parser	Image Parser	
Image Parser with legacy EBL support	Image Parser with legacy EBL support	
Image Parser without encryption support	Image Parser without encryption support	
“	Bootloader Include Parser	New component that includes common Image Parser header files
“	Bootloader in Main Flash	New component that enables the user to place bootloader in main flash for xG13 and xG14 devices
Drivers		
Delay	Bootloader Delay Driver	
SPI Master	Bootloader SPI Controller USART Driver	
SPI Slave	Bootloader SPI Peripheral USART Driver	
UART	Bootloader UART Driver	
EUART	Bootloader EUART Driver	

Copyright © 2025 Silicon Laboratories. All rights reserved.

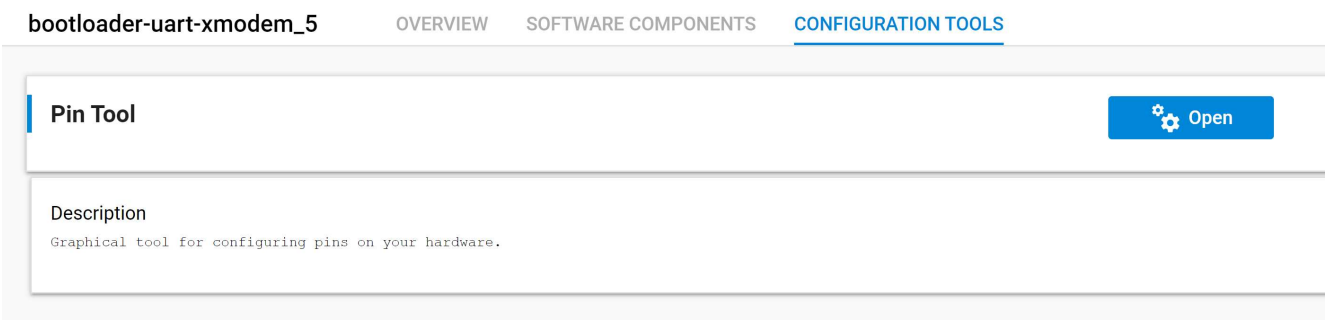
AppBuilder Plugin	Simplicity Studio v5 Component	Additional Comments
“	Bootloader SPI Controller EUSART Driver	New component supporting SPI EUSART driver. This component can be used with devices supporting EUSART interface.
“	Bootloader SPI Peripheral EUSART Driver	New component supporting SPI EUSART driver. This component can be used with devices supporting EUSART interface.
Storage		
Common Storage	Common Storage	
Common Storage (single storage slot only)	Common Storage (single storage slot only)	
Internal Storage	Internal Storage	
SPI Flash Storage	SPI Flash Storage	
“	Bootloader Storage Slot Setup	New component that facilitates configuring storage slots for Internal and SPI Flash based bootloaders
Utils		
Crypto	Crypto	
Cyclic Redundancy Check	Cyclic Redundancy Check	
Debug	Debug	
EMLIB	EMLIB Peripheral HAL	This component is no longer part of the Bootloader module. EMLIB component(s) can be found under Platform > Peripheral
EZSP GPIO Activation	EZSP GPIO Activation	
GPIO Activation	GPIO Activation	
SE Manager	SE Manager	This component is now available under Platform > Security. This component is only applicable for Series 2 devices.
Token Management	Token Management	
mbed TLS	Mbed TLS ...	These components are available under Platform > Security
Bootloader Interface		
	App Properties	A new component to configure application properties (version), which is used during creation of a GBL file using Simplicity Commander.
Bootloader-interface	Bootloader Application Interface	

Copyright © 2025 Silicon Laboratories. All rights reserved.

Pin Tool

Pin Tool

In contrast to the AppBuilder workflow, Simplicity Studio v5 offers the **Pin Tool** configuration tool, which can be used to configure the various pins of the MCU used in the project. Pin Tool can be used in standalone mode through the tile on the **Configuration Tools** tab.



Pin Tool-related configuration is also available through the Component Editor of all applicable configurable components, making it a one-stop solution for configuring pin details and setting user configuration details. Following is the list of configurable components that also include Pin Tool configuration (wherever applicable) in their configuration wizard.

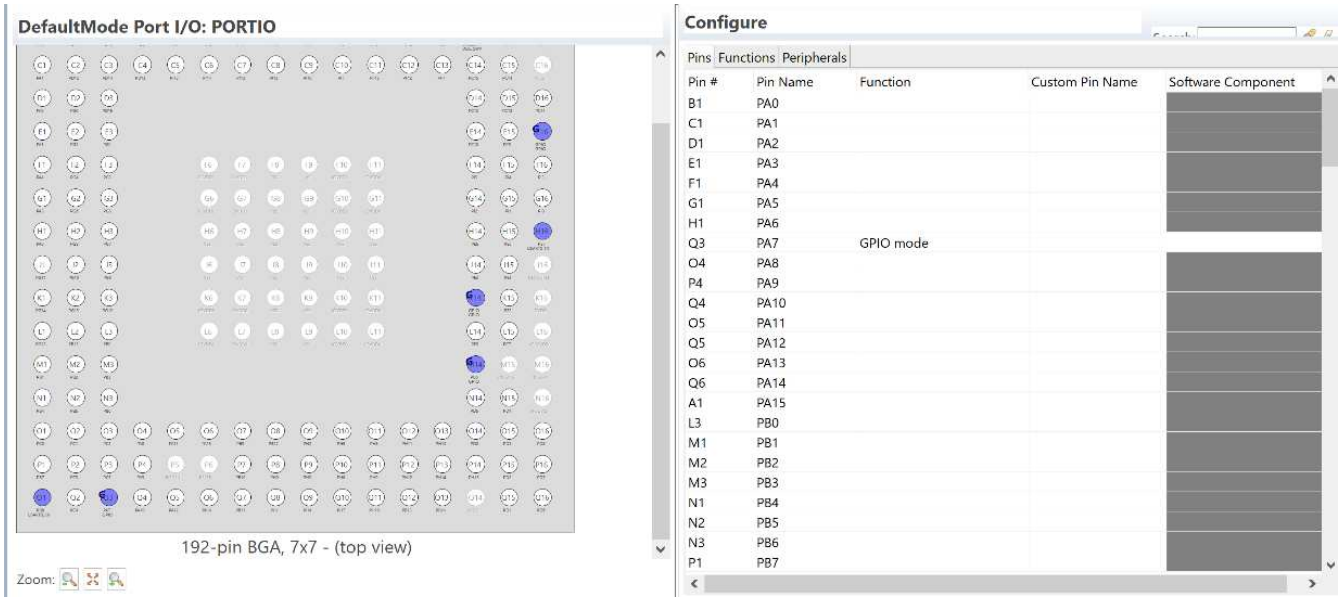
Configurable Component	Configuration Options
UART XMODEM	Menu Idle Timeout
Bootloader Core	All options that were available for the bootloader plugin are now available in the Bootloader Core component. Options include: Enable Secure Boot, Enable certificate support, Prevent bootloader write/erase, etc
Bootloader UART/EUART Driver	Configuration options include: USART settings, USART peripheral, Serial VCOM Enable/Disable
Bootloader SPI Controller USART/EUSART Driver	Configuration options include: Frequency, Peripheral and SPI Port Settings
Bootloader SPI Peripheral USART/EUSART Driver	Configuration options include: Tx Buffer Size, Rx Buffer Size, Peripheral and Port Settings
Bootloader Storage Slot Setup	Supports configuration of storage slot (start addresses and slot lengths) up to a maximum of 3 slots
Common Storage/Common Storage (single storage slot only)	Configuration options include: Start address of bootload info
Internal Storage	Configuration options include: Enable DMA based MSC Write, DMA channel to reserve
SPI Flash Storage	Enables configuration of the list of SPI flash devices that must be enabled in the bootloader image
Debug	Enable/disable Debug prints/asserts
EZSP GPIO Activation	Configuration options include: Properties of SPI NCP, WAKE INT Pin, HOST INT Pin
GPIO Activation	Configuration options include: Properties of Bootloader entry, Button Pin configuration

Copyright © 2025 Silicon Laboratories. All rights reserved.

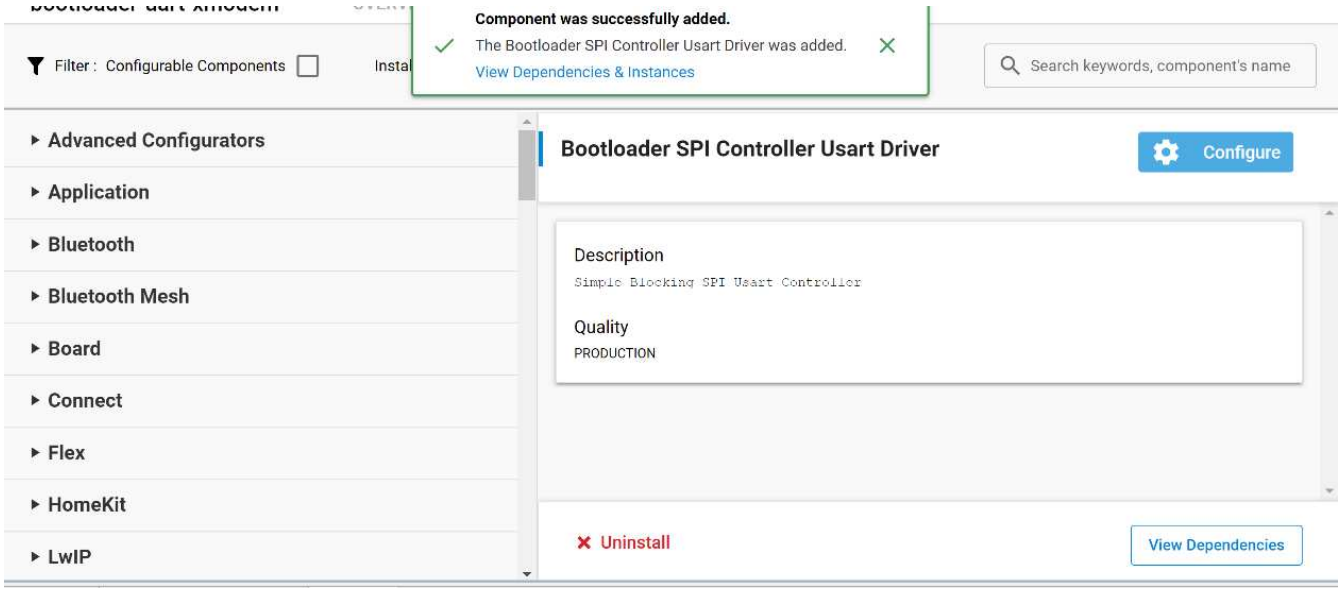
The configuration options of any installed configurable component can be accessed by clicking **Configure**. Refer to section 2 Project Creation for more details on the Component Editor.

As an example, in the following procedure the Bootloader SPI Controller USART Driver is configured using the Component Editor and the pin configuration is verified using the Pin Tool.

1. Pin Tool configuration before installing the SPI Controller component. The device used for this example is the EFM32GG11 Giant Gecko Starter Kit board (BRD2204A Rev B00).

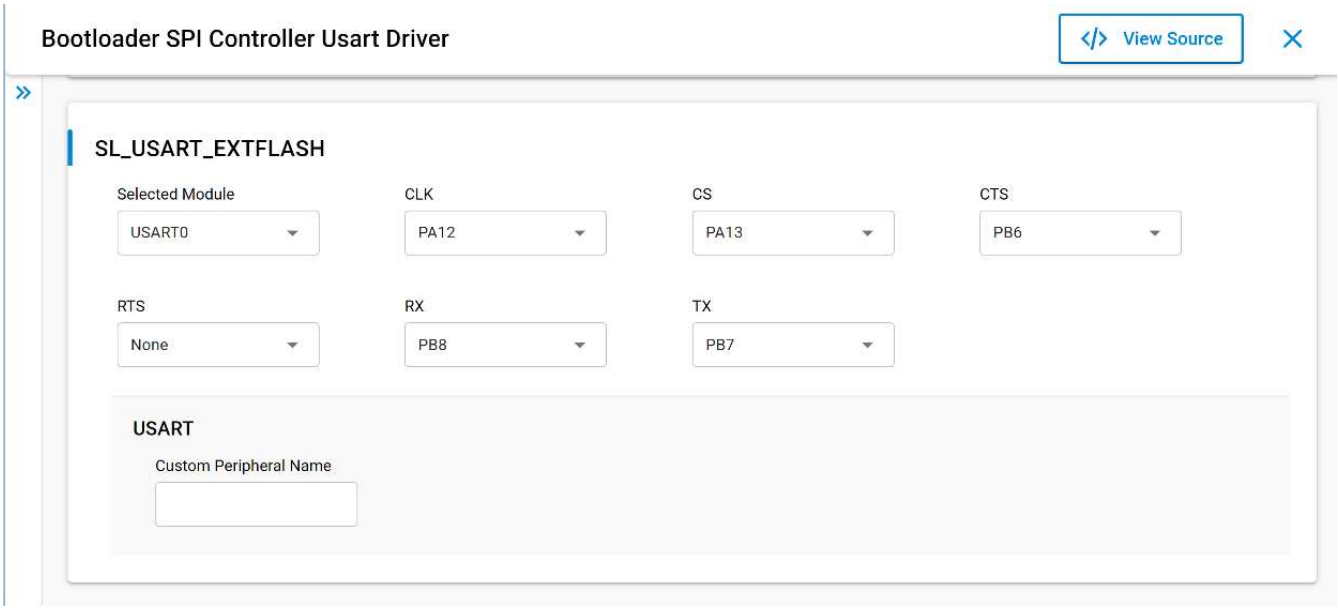


2. Install the Bootloader SPI Controller USART driver component.

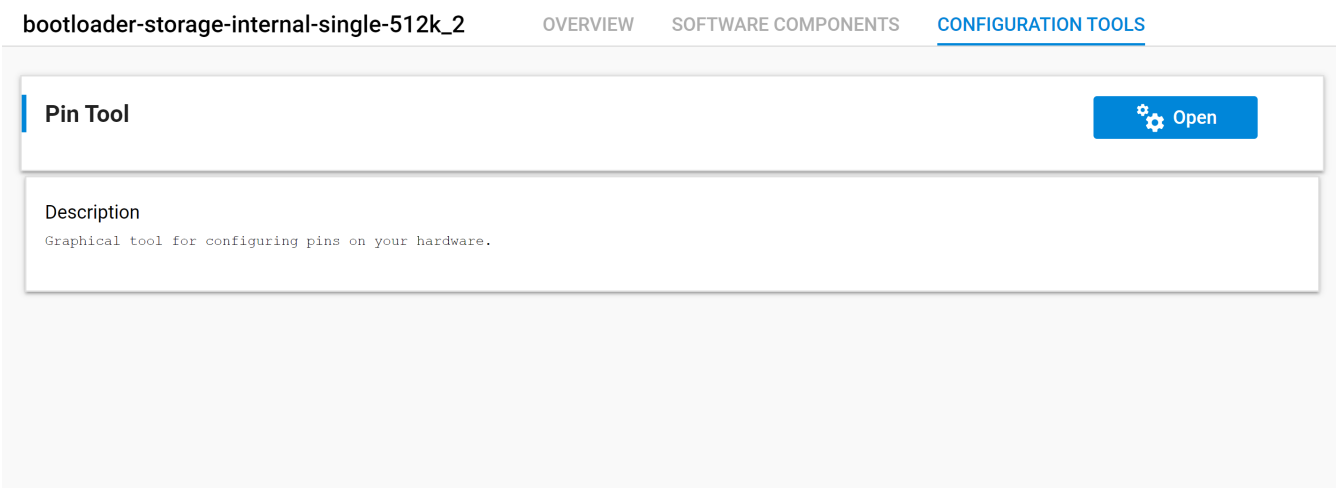


3. Click **Configure** to open the Component Editor and configure the USART peripheral as shown.

Copyright © 2025 Silicon Laboratories. All rights reserved.

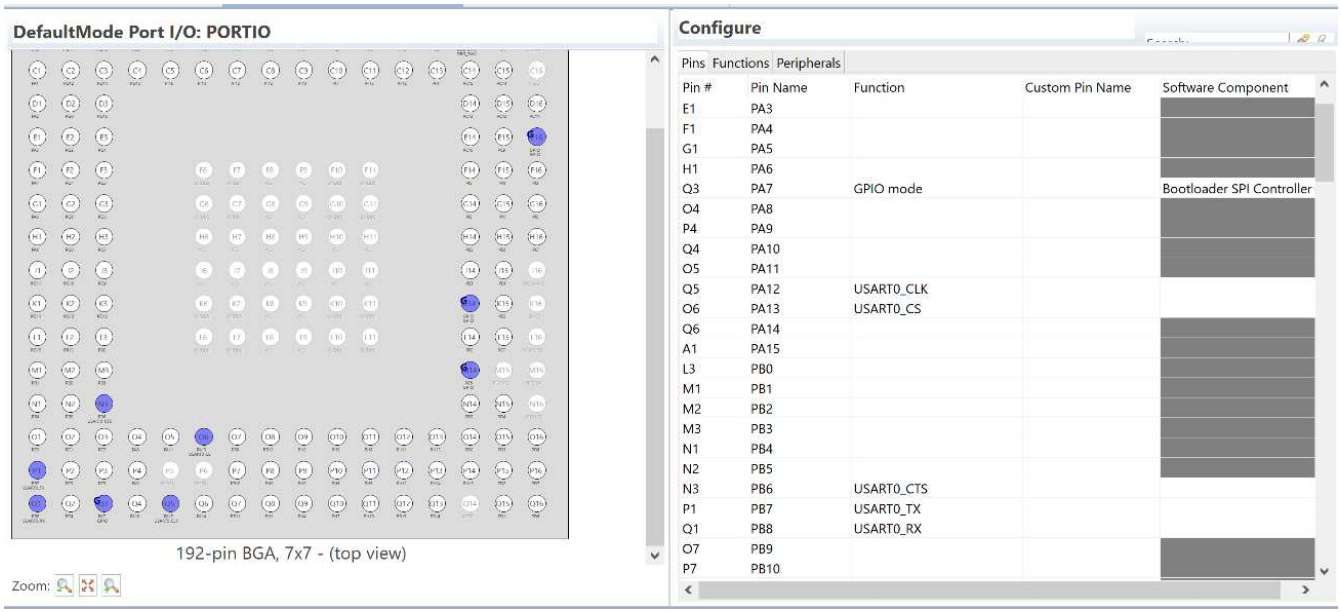


4. Open the Pin Tool.



5. The pin configurations are shown. Notice that pins PA12, PA13, PB6, PB7, and PB8 are reserved for USART0.

Copyright © 2025 Silicon Laboratories. All rights reserved.



Copyright © 2025 Silicon Laboratories. All rights reserved.

Linker File

Linker File

The linker files for GCC and IAR are now generated during the project generation step. The linker files for the bootloader are generated from template files using the Jinja template engine. The template files are included with theGecko SDK Suite (GSDK), which is installed through Simplicity Studio. The jinja template files are located at **<path-to-simplicity-studio-installation>|v5|developer|sdks|gecko_sdk_suite|<gecko_sdk_version>|platform|common|toolchain|**.This path contains folders for both **gcc** and **iar**.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Additional Macros

Additional Macros

In the AppBuilder workflow, the user could define additional compiler macros used during compilation of the bootloader application. In Gecko Bootloader v2.x, these additional macros are now part of the various components. These components have these macros defined internally, and those macros which require user intervention are available to the user as configuration options through the Component Editor.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Postbuild Steps

Postbuild Steps

For Series 1 devices, the main stage bootloader needs to be combined with the first stage bootloader, which results in a combined bootloader binary. To accomplish this, a postbuild script is made available with the GSDK, which combines the appropriate first stage bootloader with the main stage bootloader to produce a combined bootloader binary. A postbuild step must be explicitly added through the Project's settings so that the postbuild script can be run after the bootloader build is complete. For more details, see Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher.

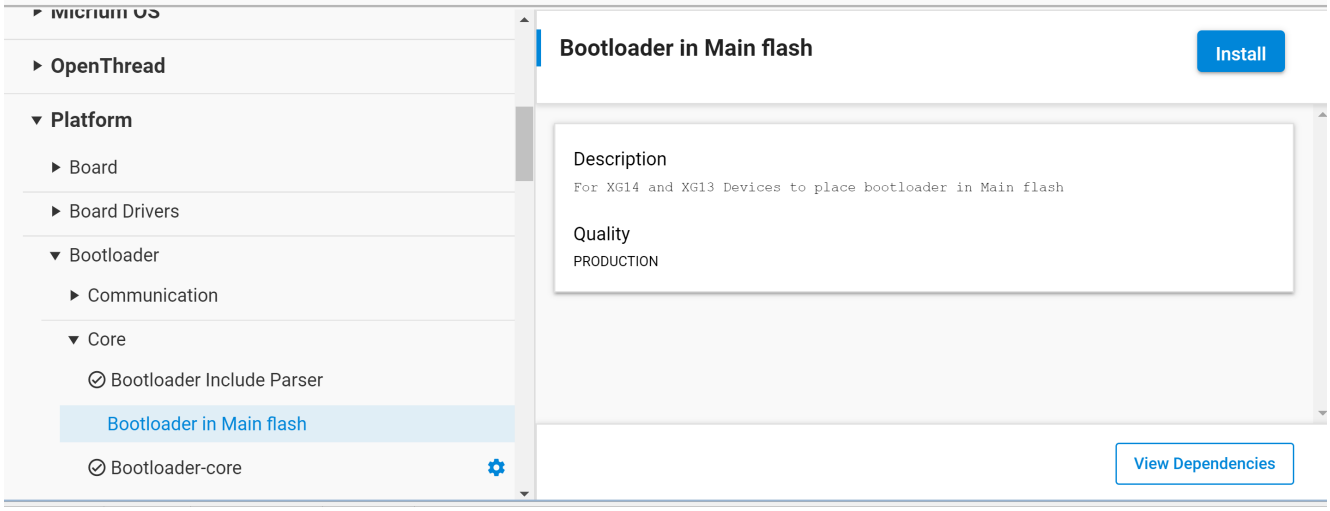
> **Note**: Adding the postbuild steps is mandatory when building bootloader binaries for Series 1 devices. If the postbuild step is not configured correctly, Simplicity Studio cannot build a combined bootloader binary image.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Main Bootloader in Main Flash

Main Bootloader in Main Flash

For xG13 and xG14 devices, the entire main stage bootloader might not fit into the bootloader flash if the user installs some extra components. In such scenarios, the main stage bootloader can be placed in the main flash by installing the **Bootloader in Main Flash** core component in the bootloader project.



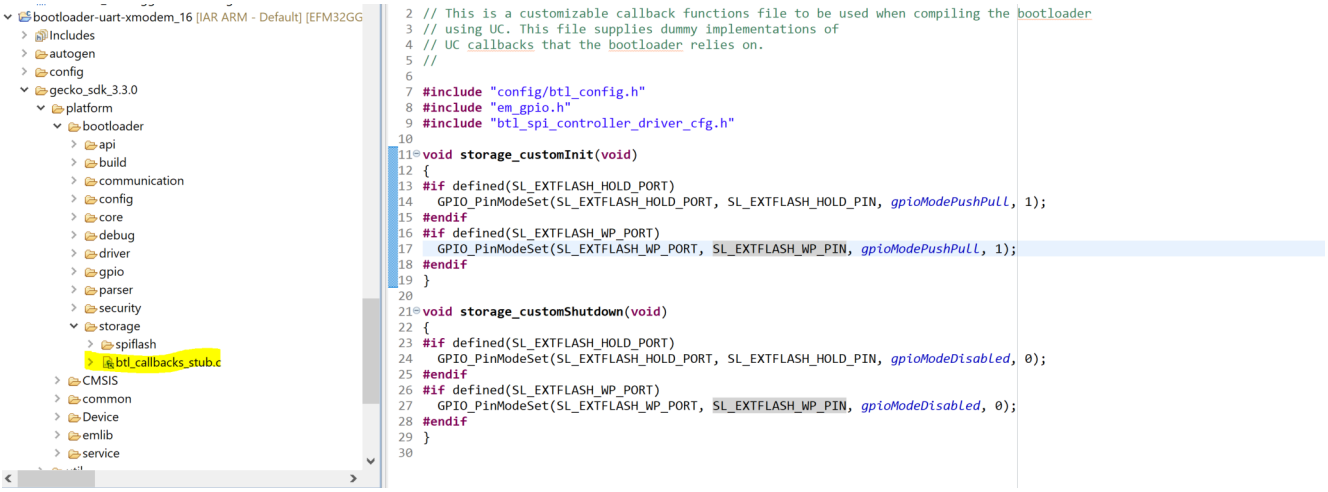
For more information on this component, see Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Callbacks

Callbacks

In contrast to the AppBuilder workflow, callbacks are now part of **btl_callbacks_stub.c**. This file is added to the project when the **SPI Flash Storage** component is installed. This file contains dummy implementation of callbacks that the bootloader relies on. This file can be found in **gecko_sdk_<version> > platform > bootloader > storage > btl_callbacks_stub.c** as shown.

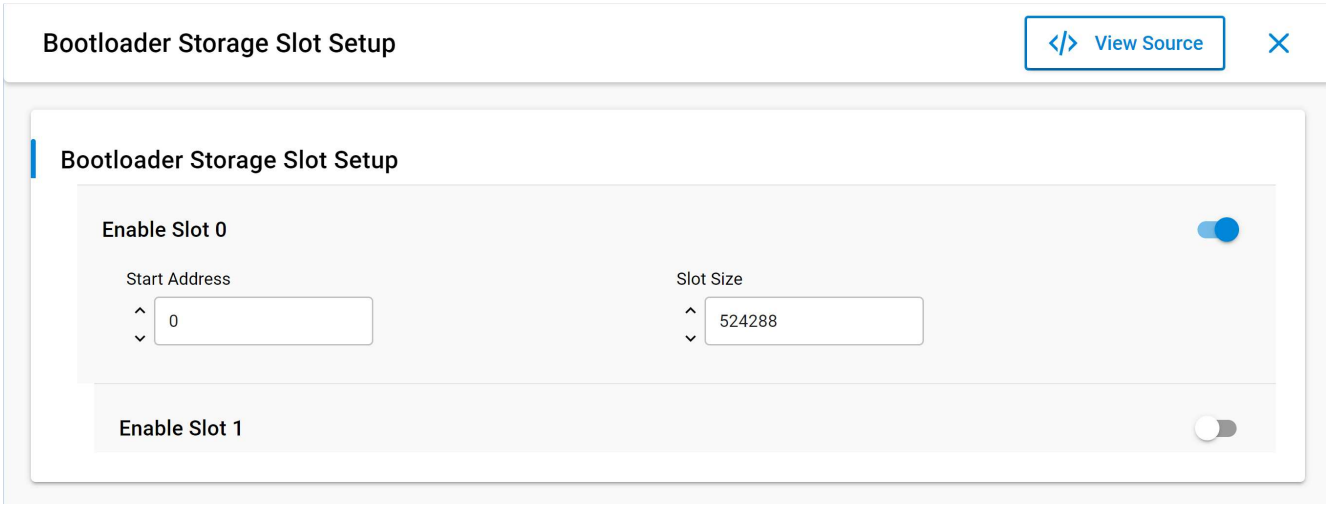


Copyright © 2025 Silicon Laboratories. All rights reserved.

Storage Slots

Storage Slots

Gecko Bootloader supports the Application Bootloader mode. This mode supports single/multiple storage slots that can be configured at compile time. In Gecko Bootloader v2.x, the storage slots can now be configured in the **Bootloader Storage Slot Setup** component using the Component Editor. A maximum of 3 storage slots can be configured.

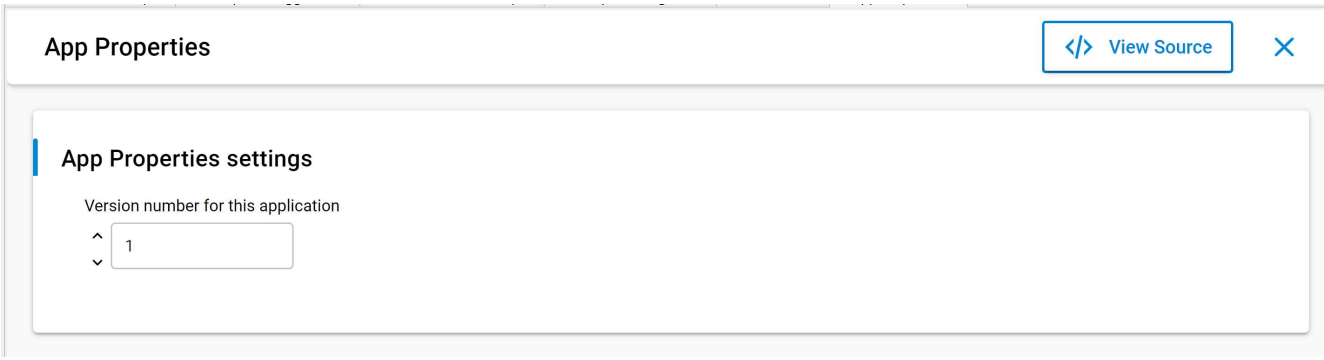


Copyright © 2025 Silicon Laboratories. All rights reserved.

App Properties

App Properties

Previously, in the AppBuilder workflow, a variable of type **AppProperties_t** had to be configured to add application properties to the application. This was then later used by Simplicity Commander during gbl file creation. Beginning with Gecko Bootloader version 2.x, a new configurable component named **App Properties** can be used to configure the application properties. This component can be installed using the Component editor which is available as part of Simplicity Studio. It is also installed automatically as a dependency on installing the **Bootloader Application Interface** component in the project. The component allows the user to configure the application version using Simplicity Studio's Component Editor as shown below.



For more details on how to configure this component, see Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Overview

Multiprotocol

This section provides background information on multiprotocol applications.

- Multiprotocol Fundamentals: Describes the four multiprotocol modes, discusses considerations when selecting protocols for multiprotocol implementations, and reviews the Radio Scheduler, a required component of a dynamic multiprotocol solution.
- Dynamic Multiprotocol User's Guide: Describes how to implement a dynamic multiprotocol solution.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Multiprotocol Fundamentals

Multiprotocol Fundamentals

This section replaces *UG103.16: Multiprotocol Fundamentals*. Further updates to this user guide will be provided here.

Multiprotocol is a way to use more than one protocol on a single chip. Implementing more than one protocol on a single device can improve:

- Cost savings: A single device can perform more than one function.
- Space savings: End user product packaging can be smaller and simpler when protocols can share a single radio.
- Energy savings: The number of devices on a network is reduced.

The following pages describe:

- Different ways to implement multiprotocol devices and the infrastructure requirements for an effective implementation.
- Aspects of protocol operation to be considered when selecting a protocol for a multiprotocol implementation.
- The operation of the Silicon Labs Radio Scheduler, an essential component of a dynamic multiprotocol implementation.

These pages describe the use of multiprotocol on a single chip. For more information about coordinating multiple radio chips, like one from Silicon Labs and an external Wi-Fi radio, see Wi-Fi Coexistence Fundamentals.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Types of Multiprotocol Implementations

Types of Multiprotocol Implementations

Multiple protocols can be implemented on a single device in different ways:

- **Switched**: The application switches between protocols using a bootloader.
- **Dynamic**: The application time-slices between two protocols.
- **Concurrent**: The application is able to receive multiple protocols full time by:
  - Running on the same RF channel of a single radio,
  - Using Silicon Labs Concurrent Listening technology on a single radio, or
  - Using a multi-chip or multi-radio solution.

The following discusses these implementations in more detail.

Switched Multiprotocol

In switched multiprotocol, a device is initially programmed with one protocol, and then uses a bootloader to switch to another protocol at some point in the future. Switched multiprotocol has two primary use cases.

Future Proofing: Device manufacturers may need to sell their devices into different protocol environments, or they may need to plan for an environment changing over time.

Commissioning by Smartphone: Device manufacturers who are committed to a single protocol environment, such as Zigbee Home Automation, may want to make the process of adding a new device more secure and user friendly. In this case, the device is initially programmed with a Bluetooth commissioning application. The Home Gateway/Trust Center is also programmed with a Bluetooth application, and the customer uses a commissioning app on their smartphone. Using the smartphone app they can join a new device onto the network, set up a pairing with other appropriate devices in the network, then switch over to the network protocol. Each wireless network protocol has its own mechanisms for joining and pairing devices, but all can be accommodated with this mechanism.

An effective switched multiprotocol implementation requires:

- A multiprotocol-enabled platform with sufficient memory.
- A consistent API to use the radio.
- A cross-compatible bootloader.

Dynamic Multiprotocol

In a dynamic multiprotocol implementation, two protocols run concurrently, but the application time-slices the radio and rapidly changes radio configurations, such as channel, to enable different wireless protocols to operate reliably at the same time. In time-slicing, the software schedules tasks based around their priority and minimum duration and will default to a background listen task between scheduled tasks. The scheduler can interrupt or delay a lower priority task if a higher priority task is scheduled.

A dynamic multiprotocol implementation allows a device to perform multiple simultaneous functions. For example, an end user can use a smartphone app to connect to the device via Bluetooth to control it or to perform diagnostics, while at the same time the device is connected to the Zigbee network routing packets and acting on Zigbee Cluster Library commands.

In addition to a multiprotocol-enabled platform with sufficient memory, common code infrastructure, common radio interface, and common API to use the radio such as that provided by Silicon Labs RAIL, an effective dynamic multiprotocol implementation requires:

- An RTOS to support task switching and resource sharing.
- A radio scheduler to manage time-slicing. See Radio Scheduler for background on the Silicon Labs radio scheduler.

Copyright © 2025 Silicon Laboratories. All rights reserved.

See Dynamic Multiprotocol User's Guide for additional information about Silicon Labs dynamic multiprotocol implementation and RAIL Fundamentals for more information about Silicon Labs RAIL.

Single Channel Concurrent Multiprotocol

In a single radio, single channel concurrent multiprotocol implementation, two protocols are able to run simultaneously on a single radio by sharing the same radio channel.

For example, a Zigbee/Thread gateway or controller device could manage both a Zigbee-based and a Thread-based network at the same time. In the case of sharing a single channel, the networks must coordinate what channel they use. In addition, they must be able to accommodate reduced bandwidth either by scaling their traffic or by limiting the number of devices on the network.

Concurrent Listening

Concurrent Listening, a Silicon Labs technology, allows the Zigbee and Thread stacks to operate on independent 802.15.4 channels when used with the multiprotocol radio co-processor (RCP). This feature works on EFR32xG21 and EFR32xG24 parts only. For detailed information on the multiprotocol RCP, see Running Zigbee, OpenThread, and Bluetooth Concurrently on a Linux Host with a Multiprotocol Co-Processor.

Concurrent Listening allows a single radio to listen on two channels simultaneously. This is accomplished by extremely rapid switching (on the order of tens of microseconds) that allows the radio to detect preambles for incoming packets on either channel. Once a preamble is detected, the radio stays on that channel until the packet is received, then resumes rapid switching.

Note that this fast switching differs from the time slicing used in dynamic multiprotocol, because the dwell on each channel is just sufficient to detect a preamble and shorter than the full preamble time, so incoming packets are not missed on either channel. In contrast, for dynamic multiprotocol the time slot is large enough to receive scheduled packets, which means that the other channel is not being listened to during that time.

When enabling the concurrent listening feature the PHY performance is slightly degraded. The sensitivity of the IEEE 802.15.4 PHY in this mode drops to around -98 dBm for both the EFR32xG21 and EFR32xG24.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Protocol Considerations

Protocol Considerations

It is important to recognize that a single radio cannot receive or transmit two packets for two different protocols simultaneously. To share a single radio, both protocols have to accept that they will not have 100% use of the radio. They must therefore be able to survive loss of radio without significantly decreasing performance or losing application messages. In this case, *significantly* means that the application is working outside of its intended operational parameters. Examples are higher than acceptable message latency or message loss.

While some protocols and applications will have strict timing requirements for packet transmission, in most cases the critical area of concern with multiprotocol is the receipt of incoming packets from the networks.

For a protocol to be a good candidate for a dynamic or concurrent multiprotocol implementation, the following considerations must apply:

- Neither protocol can take over the radio for long periods (longer than a few milliseconds at a time) under normal operating conditions, excluding commissioning, firmware upgrades, and other related operations.
- One or both protocols must have a robust mechanism for managing loss of incoming packets, such as MAC retries.
- One or both protocols must have very short packets and/or a short time required on the radio.
- For dynamic multiprotocol, one or both protocols must implement strict time slots/connection intervals.

> **Note**: When using dynamic multiprotocol, a radio has to switch between two different PHYs and it 'disappears' from one network or another. For protocols where the device might be the 'parent' of a sleepy end device, if the parent device is not available when the sleepy end device wakes up to send a message, regular dependency on retries will impact the sleepy end device’s battery life.

> **Note**: If you have a protocol that uses multiple radio configurations, you do not need to implement multiprotocol. For example, Silicon Labs' RAIL and the radio configurator support multiple radio configurations in the same protocol. See *AN971: EFR32 Radio Configurator Guide* for more details. Multiprotocol is used if protocols have mostly independent protocol stacks.

Dynamic Multiprotocol Example

Bluetooth LE and Zigbee are suitable protocols for a dynamic multiprotocol implementation. Because of the low duty cycle for Zigbee traffic and the retry mechanisms in the Zigbee networking stack, a Zigbee Router can switch its radio to some other frequency or protocol for short periods without dropping any messages at an application level.

Bluetooth LE radio usage can be predicted and planned in advance. Bluetooth beacons are quite short packets, usually less than 30 bytes. The radio only needs about 1 ms to transmit the beacon, and the interval between beacons is typically no shorter than 100 ms, thus providing a duty cycle of just 1%. This means that the radio can devote at least 99% of its time to the main Zigbee network.

If you are using Silicon Labs products, adding Bluetooth LE to most proprietary networks is also possible because of the short Bluetooth LE packets and the long delay between communications. The proprietary stack must be updated to work with the Bluetooth stack, and all proprietary communications should be examined to determine the priority and required timing accuracy of it. For more details, see Dynamic Multiprotocol User’s Guide and Dynamic Multiprotocol Development with Bluetooth and Proprietary Protocols on RAIL.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Concurrent Multiprotocol Example

Zigbee and Thread are examples of suitable protocols for a concurrent multiprotocol implementation.

The multiprotocol device can control which IEEE 802.15.4 channel it is operating on to connect to other Zigbee and Thread devices. It only has a single IEEE 802.15.4 MAC/PHY to listen to or send on both networks concurrently, meaning that switching radio PHYs is not required.

The multiprotocol device manages incoming packets from either network by filtering both PAN IDs, directing to the appropriate networking stack.

Operational constraints include:

- The multiprotocol device must control selection of the same IEEE 802.15.4 channel for both Zigbee and Thread, which means that it is most likely a Zigbee Coordinator/Thread Leader, in practice the Gateway/Controller in both networks. If using Concurrent Listening on an EFR32xG21 or EFR32xG24, this constraint does not apply.
- The device cannot be required to receive Zigbee and Thread packets simultaneously. However, because of MAC retries, in most cases this should not be a limitation.
- The combined duty cycle of the Zigbee and Thread traffic to or from the device must not exceed what would normally be tolerated by a single Zigbee or Thread device.

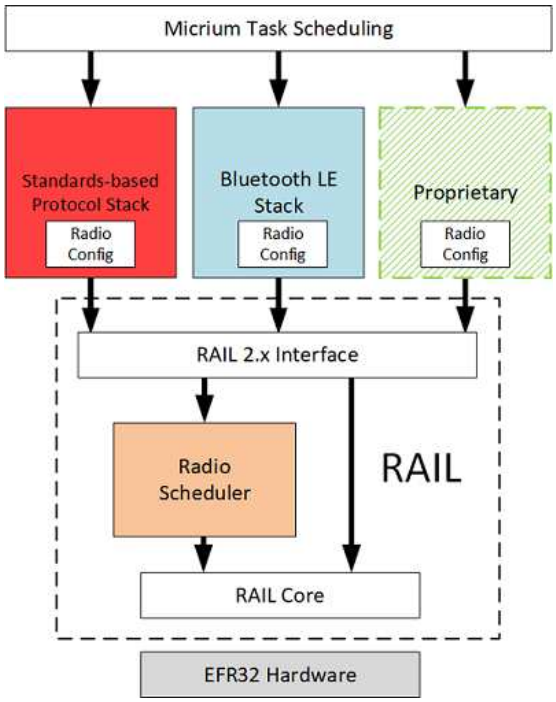
Copyright © 2025 Silicon Laboratories. All rights reserved.

Radio Scheduler

Radio Scheduler

A radio scheduler is an essential component of a dynamic multiprotocol implementation. It is a system for on-demand scheduling of radio tasks as requested by wireless stacks and the manufacturer's applications. This page introduces basic radio scheduler concepts. For details, including examples, of the Silicon Labs radio scheduler operation, see the Dynamic Multiprotocol User's Guide.

The Silicon Labs Radio Scheduler is part of the RAIL library. It operates above the radio hardware and below the RAIL API, as shown in the figure:



Different radio events in each protocol may be more or less important, or more or less time sensitive, depending on the situation. The Radio Scheduler can take those into account when making decisions about conflicts and how to adjudicate them.

Micrium OS is an RTOS that allows stacks and application logic to share CPU execution time.

The Radio Scheduler uses the following concepts.

Radio Operation: A radio operation is a specific action to be scheduled that has both a radio configuration and a priority. Each stack can request that the radio scheduler perform three radio operations:

- Background receive: Continuous receive, intended to be interrupted by other radio operations
- Scheduled receive: Receive at a future time with a minimum duration
- Scheduled transmit: Transmit at a future time with a minimum duration

Radio Config: Radio config determines the state of the hardware that must be used to perform a radio operation.

Priority: Each operation from each stack has a default priority. An application can change default priorities.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Slip Time: Slip time is the maximum time in the future when the operation can be started if it cannot begin at the requested start time.

Yield: A stack must voluntarily yield at the end of an operation or sequence of operations, unless it is performing a background receive. Until the stack yields, the scheduler will not schedule lower priority tasks.

The Radio Scheduler can interrupt a scheduled radio operation if a higher priority task conflicts with it. This could occur in the following two circumstances:

1. A scheduled radio operation takes longer than expected, and the corresponding stack does not yield before the higher priority radio operation must start.
2. A higher priority radio operation has just been scheduled to occur in the future and conflicts with a lower priority operation already scheduled.

Certain long-lived radio operations can have an outsized impact on the correct operation of the product. The application may need to coordinate these tasks between the protocols. If the application does not, then the radio scheduler priorities will take precedence. This can result in the task being interrupted prematurely.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Dynamic Multiprotocol

Dynamic Multiprotocol User's Guide

This section replaces *UG305: Dynamic Multiprotocol User's Guide*. Further updates to user guide will be provided here.

Silicon Labs software is designed to be used by multiple protocols on a single wireless chip. This user's guide provides details about implementing an application using Silicon Labs' Dynamic Multiprotocol solution. Dynamic multiprotocol time-slices the radio and rapidly changes configurations to enable different wireless protocols to operate reliably at the same time.

> **Note**: The Zigbee-specific information in this document applies to version 6.10.x and lower.

Details on specific dynamic multiprotocol implementations are provided in the following application notes:

- Dynamic Multiprotocol Development with Bluetooth and Zigbee EmberZNet SDK 7.x and Higher
- AN1134: Dynamic Multiprotocol Development with Bluetooth and Proprietary Protocols on RAIL in GSDK v2.x
- Dynamic Multiprotocol Development with Bluetooth® and Proprietary Protocols on RAIL in GSDK v3.x and Higher
- AN1209: Dynamic Multiprotocol Development with Bluetooth and Connect
- Dynamic Multiprotocol Development with Bluetooth® and OpenThread in GSDK v3.x

Terminology

The following lists some of the terminology specific to the dynamic multiprotocol implementation.

Radio Abstraction Interface Layer (RAIL): The common API through which higher level code gains access to the EFR32 radio.

Radio Operation: A specific action to be scheduled. A radio operation has both a radio configuration and a priority. Each stack can request that the radio scheduler perform up to two radio operations (background receive and either Scheduled Receive or Scheduled transmit) at a time:

- **Background Receive**: Persistent receive, intended to be interrupted by Scheduled operations, and returned to after their completion.
- **Scheduled Receive**: Receive packets or calculate RSSI at a specified time and duration. (Developers working on RAIL, note that in terms of the RAIL API, “Scheduled Receive” as used in this document refers to any receive operation, other than `RAIL_StartRx`, and is not just limited in scope to `RAIL_ScheduleRx`.)
- **Scheduled Transmit**: Any one of various transmit operations including immediate transmit, scheduled (future) transmit, or CCA-dependent transmit. (Developers working on RAIL, note that in terms of the RAIL API, “Scheduled Transmit” as used in this document refers to any transmit operation, and is not limited in scope to `RAIL_StartScheduledTx`.)

Radio Config: Determines the state of the hardware that must be used to perform a radio operation.

Radio Scheduler: RAIL component that arbitrates between different protocols to determine which will have access to the radio.

Priority: Each operation from each stack has a default priority. An application can change default priorities.

Slip Time: The maximum time in the future when the operation can be started if it cannot begin at the requested start time.

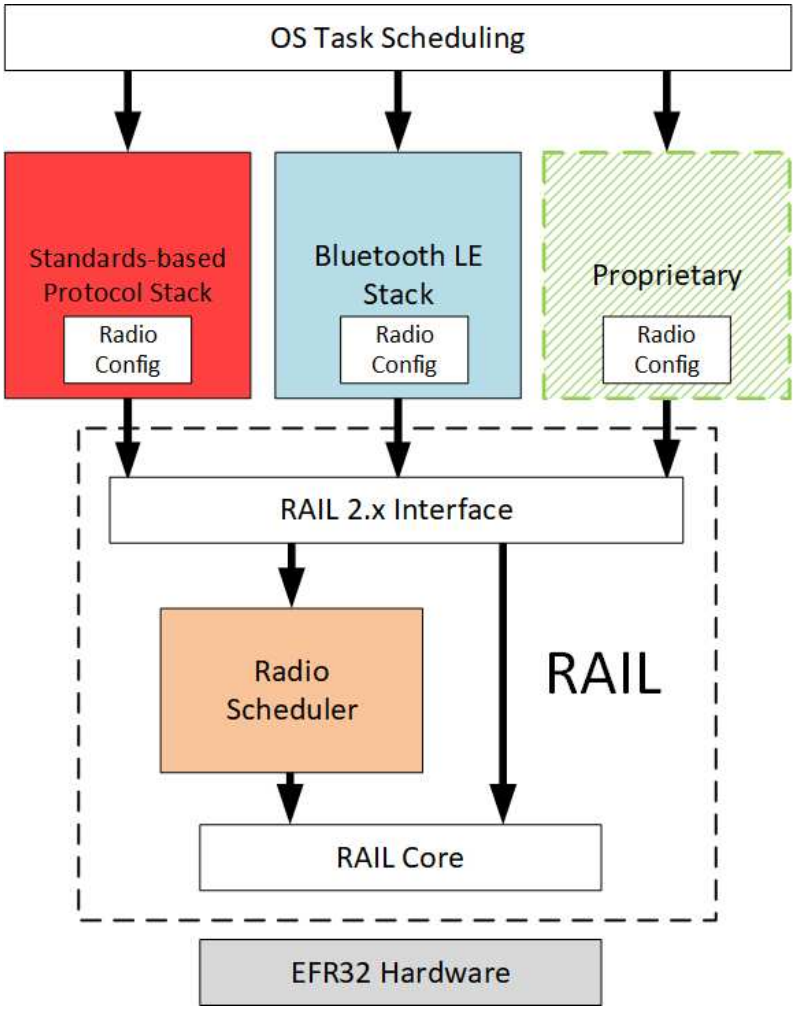
Yield: A stack must voluntarily yield at the end of an operation or sequence of operations, unless it is performing a background receive. Until the stack yields, the scheduler will not scheduler lower priority tasks.

Copyright © 2025 Silicon Laboratories. All rights reserved.

RTOS (Real Time Operating System) Kernel: The part of the operating system that is responsible for task management, and inter-task communication and synchronization.

Architecture

Dynamic Multiprotocol makes use of the EFR32 hardware and the RAIL software as its building blocks. Zigbee, Bluetooth, and/or any other standards-based or proprietary protocols can then be built on top of these foundational layers, using Micrium to manage execution of code between different protocols. The following diagram illustrates the general structure of the software modules.



Beginning with version 2.0, RAIL requires the passing of a radio configuration handle to the RAIL API calls. This configuration describes various PHY parameters that are used by the stack.

The OS is an RTOS that allows stacks and application logic to share CPU execution time.

The Radio scheduler is a software library that intelligently answers requests by the stacks to perform radio operations to maximize reliability and minimize latency. API's provided by RAIL that do not engage the radio bypass the Radio scheduler.

The RAIL core configures the EFR32 hardware in response to instructions from the radio scheduler.

Single Firmware Image

Dynamic Multiprotocol allows a software developer to generate a single monolithic binary that is loaded onto an EFR32. Software updates are done by upgrading the entire binary. This is accomplished using the Gecko Bootloader, the details of which can be found in *UG266: Silicon Labs Gecko Bootloader User's Guide for GSDK 3.2 and Lower*, Silicon Labs Gecko

Copyright © 2025 Silicon Laboratories. All rights reserved.

Bootloader User's Guide for GSDK 4.0 and Higher (series 1 and 2 devices) or Silicon Labs Gecko Bootloader User's Guide for Series 3 and Higher.

Independent Stack Operation

The Silicon Labs stacks still operate independently of one another in a Dynamic Multiprotocol situation. Certain long-lived radio operations will have an impact on another protocol's latency and compliant operation. It is up to the application to determine any special considerations for these events. See The Radio Scheduler for more information.

Copyright © 2025 Silicon Laboratories. All rights reserved.

The Radio Scheduler

The Radio Scheduler

The Radio Scheduler is a component of RAIL (Radio Abstraction Interface Layer). RAIL provides an intuitive, easily-customizable radio interface layer and API, which supports proprietary or standards-based wireless protocols. The Radio Scheduler is designed to allow for radio operations that can be scheduled and prioritized. Different radio operations in each protocol may be more or less important, or more or less time sensitive, depending on the situation. The scheduler can take those into account when making decisions about conflicts and how to adjudicate them.

Unless you are developing applications with a custom protocol on RAIL, most radio scheduler functions are handled automatically by underlying stack and RAIL code. You only need to use the stack through its normal API.

At a high level, the stack sends a radio operation (for example a Scheduled Receive or Scheduled Transmit). The radio operations are queued and then serviced at a future time based upon their parameters. When it is time to start the radio operation the scheduler examines whether a competing event exists and whether the operation can be delayed. If the scheduler cannot run the event, it returns the result to the higher layer, which may retry with new parameters.

Once the radio operation has started, the corresponding stack can send the scheduler additional operations based on the results of the previous operation (for example waiting for an ACK). At the end of each operation or sequence of operations the stack must yield use of the radio.

Radio Operations

Each event in the scheduler is broken up into elements called Radio Operations, which are associated with a radio config and a priority.

Every operation has a priority and is interrupted if the scheduler receives a higher priority operation that overlaps in time. Lower priority radio operations that cannot be run based on their schedule parameters will fail, and it is up to the respective stack to retry them. Once the scheduler actively runs a radio operation from the stack, the stack can continue to send additional radio operations until it voluntarily yields, or until the scheduler receives a higher priority radio operation and preempts it.

- Background Receive
- Scheduled Receive
- Scheduled Transmit

Each stack can ask the Radio Scheduler to perform up to two radio operations (background receive and either Scheduled Receive or Scheduled transmit) at a time.

Each operation has the following parameters:

Parameter	Description
Start Time	An indication at what point in the future this radio operation will run. This could be “run right now” or some value in microseconds in the future.
Priority	A number that indicates the relative priority of the operation. When using the default settings, Bluetooth LE radio operations are almost always higher priority than Zigbee operations.
Slip Time	An amount of time that the event can be delayed beyond its start time and still be acceptable to the stack. This may be 0, in which case the event cannot be slipped.
Transaction Time	The approximate amount of time that it takes to complete the transaction. Transmit events usually have a much more well-defined transaction time, while receive events are often unknown. This is used to help the radio scheduler determine whether an event can be run.

The stack defines these various parameters appropriate to the operation being executed. For example, Bluetooth connection events are often scheduled in the future and have no allowed slip, whereas Zigbee transmit events can often

Copyright © 2025 Silicon Laboratories. All rights reserved.

be delayed a small amount and start later.

From the perspective of the RAIL Radio Scheduler, Scheduled transmit and Scheduled receive are identical. They are both simply operations that require use of the radio, and thus cannot be executed simultaneously. The difference is only apparent at RAIL API layer, where either a TX or RX API is called.

Background Receive

This is a continuous receive mode that is intended to be interrupted by other operations and returned to after their completion. If Background Receive is higher priority than other operations, those radio operations will not be scheduled and will not run. It is up to the stacks or application to change the priority or voluntarily yield. See Examples with Background Receive, Yield Radio and State Transition for examples of how Background receive interacts with Scheduled operations.

Scheduled Receive

This is a receive at a future time with a specified duration. The radio scheduler takes into consideration the radio switching time in deciding whether the operation will be scheduled. If it cannot be scheduled, then the scheduler sends a fail event to the calling stack. The radio operation is automatically extended until the stack voluntarily yields, or the scheduler receives a higher priority operation and interrupts it. Extending the receive allows the stack to continue a radio operation based on the requirements of the higher level protocol, for example transmission of a response based on the received data.

Scheduled Transmit

This is a transmit at a future time with a minimum duration. This minimum duration can include expected follow-on events, for example an ACK to an IEEE 802.15.4 transmit. However, the minimum time for this operation does not have to include unexpected events that may extend the time beyond the minimum duration, for example backoffs due to CCA failures in IEEE 802.15.4. The radio scheduler takes into consideration the radio switching time in deciding whether the operation will be scheduled. If it cannot be scheduled, then the scheduler sends a fail event to the calling stack.

Radio Config

Each radio operation is associated with a predefined radio config that determines the state of the hardware that must be used to perform the operation. The Radio Configs keep track of the stack's current state so that future radio operations will use the same radio parameters. Radio Configs may be active or dormant. If the stack changes an active Radio Config then RAIL makes an immediate change to the hardware configuration as well, for example changing a channel. If the radio config is not currently active then the next scheduled radio operation will use the new radio config.

Priority

Each radio operation has a priority which indicates to the scheduler which operation should be executed if there is a timing overlap between multiple operations. The scheduler treats a priority of 0 as the highest priority and 255 as the lowest priority. The radio scheduler will allow the task with the highest priority to access the physical radio hardware. With most tasks control is returned to the radio scheduler only on completion, but tasks like background receive will be interrupted in case a task with higher priority becomes active.

The stacks each have a default set of priorities based on Silicon Labs' analysis of how best to cooperate to maximize the duty cycle and avoid dropped connections for a generic use case. Specific use cases may have different needs. The priorities are as follows, from highest to lowest:

1. Bluetooth LE Scheduled Transmit
2. Bluetooth LE Scheduled Receive
3. Other protocol Scheduled Transmit
4. Other protocol Background Receive

These priorities may be overridden or changed by the application. It is up to the application to decide under what circumstances to change them. 802.15.4 Rail Priority and Bluetooth Priorities contain more details on priorities for their specific instances.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Slip Time

Every radio operation must have a "slip time," or maximum start time, meaning the furthest time in the future when the operation can be started if it cannot begin at the requested start time. This allows for the scheduler to work around higher priority events that are occurring at the same time, or higher priority events that extend beyond their expected duration. The protocol generally dictates what the slip time can be, but the radio scheduler is capable of handling this on a per-operation basis, allowing a stack to slip some events but not others. In general, IEEE 802.15.4 has longer slip time and Bluetooth LE has a minimal slip time.

Yield

Once a sequence of radio operations is actively being run, the stack may continue to add operations extending the initial operation until the stack has nothing more to do for the particular message exchange. A stack must voluntarily yield unless it is performing a background receive. If a stack does not yield then it will continue to extend its radio operation, and lower priority radio operations will then trigger a failure back to the corresponding stack that requested that radio operation. A higher priority operation cannot interrupt a currently-running, lower priority radio operation that has not yielded. See Examples with Background Receive, Yield Ration and State Transition for examples of situations where explicitly yielding the radio is necessary.

Interrupting a Radio Operation

A scheduled radio operation may be interrupted if a higher priority operation conflicts with it. This could occur in the following two circumstances:

1. A scheduled radio operation takes longer than expected and the corresponding stack does not yield before the higher priority radio operation must start.
2. A higher priority radio operation has just been scheduled to occur in the future and conflicts with a lower priority operation already scheduled.

Long-Lived Radio Operations

Certain long-lived Radio Operations can have an outsized impact on the correct operation of the product. The application may need to coordinate these operations between the protocols. If the application does not, then the radio scheduler priorities take precedence. For example, an IEEE 802.15.4 energy scan can require that the radio stay on to gather sufficient energy readings. If the application does not properly coordinate the operations, the scan could be interrupted prematurely due to a higher priority Bluetooth operation.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Radio Scheduler Examples

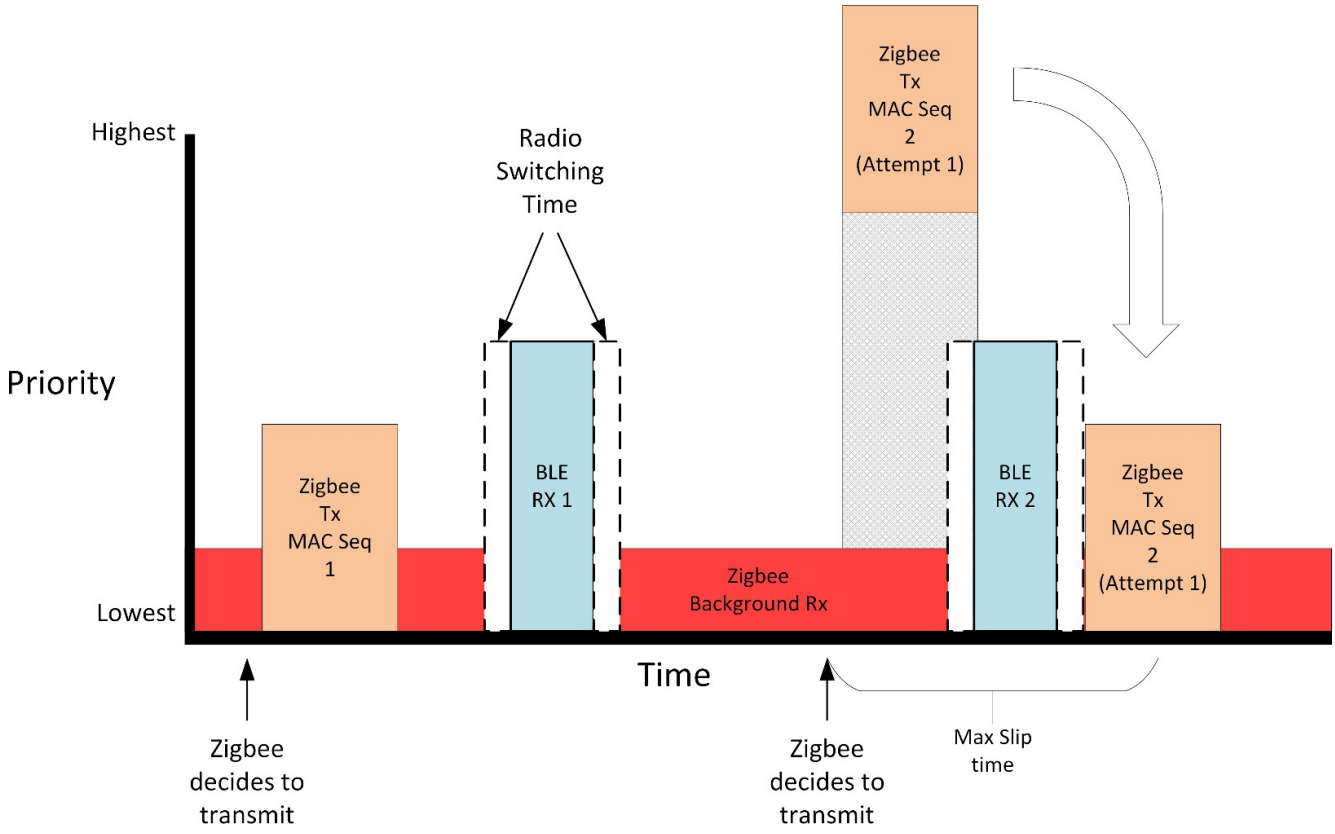
Radio Scheduler Examples

All examples use Bluetooth LE and Zigbee, but the principles apply to other Bluetooth/802.15.4 combinations.

The scheduler starts out by having a low priority Zigbee background receive operation. This represents an always-on router that may need to receive IEEE 802.15.4 packets at unknown times. A Bluetooth LE connection is also active and requires the stack to be ready to receive every 30 ms. The Bluetooth LE stack may schedule this well in advance due to the connection's predictable nature.

Priority Scheduling

This provides a basic example of adjudicating priorities of the different radio operations.



The Zigbee stack decides that it needs to send a packet. It may do this as an on-demand event, meaning the stack decides that it wants to send a packet *now* without informing the scheduler well in advance. This is in contrast to how Bluetooth LE operates, where the scheduled operations are known reasonably far in advance. The scheduler evaluates that it is possible to perform the Zigbee TX 1 radio operation and still service the higher priority Bluetooth LE reception event in the future. Therefore, the scheduler allows the transmit event to occur. The Zigbee stack performs all the pieces of this transmit operation (waiting for a MAC ack) and then voluntarily yields. The estimated transaction time of the Zigbee transmit radio operation does NOT include retries.

In this example, Bluetooth LE is *already* scheduled to receive in the future and the Zigbee stack wants to transmit. For the first Zigbee TX 1 radio operation there is enough time before the Bluetooth LE RX 1 radio operation, so the scheduler allows the stack to perform the operation. Later, when the Zigbee stack tries to schedule Zigbee TX 2 the scheduler determines there is not enough time before the high priority Bluetooth LE RX 2 event. However, the Zigbee stack has indicated that

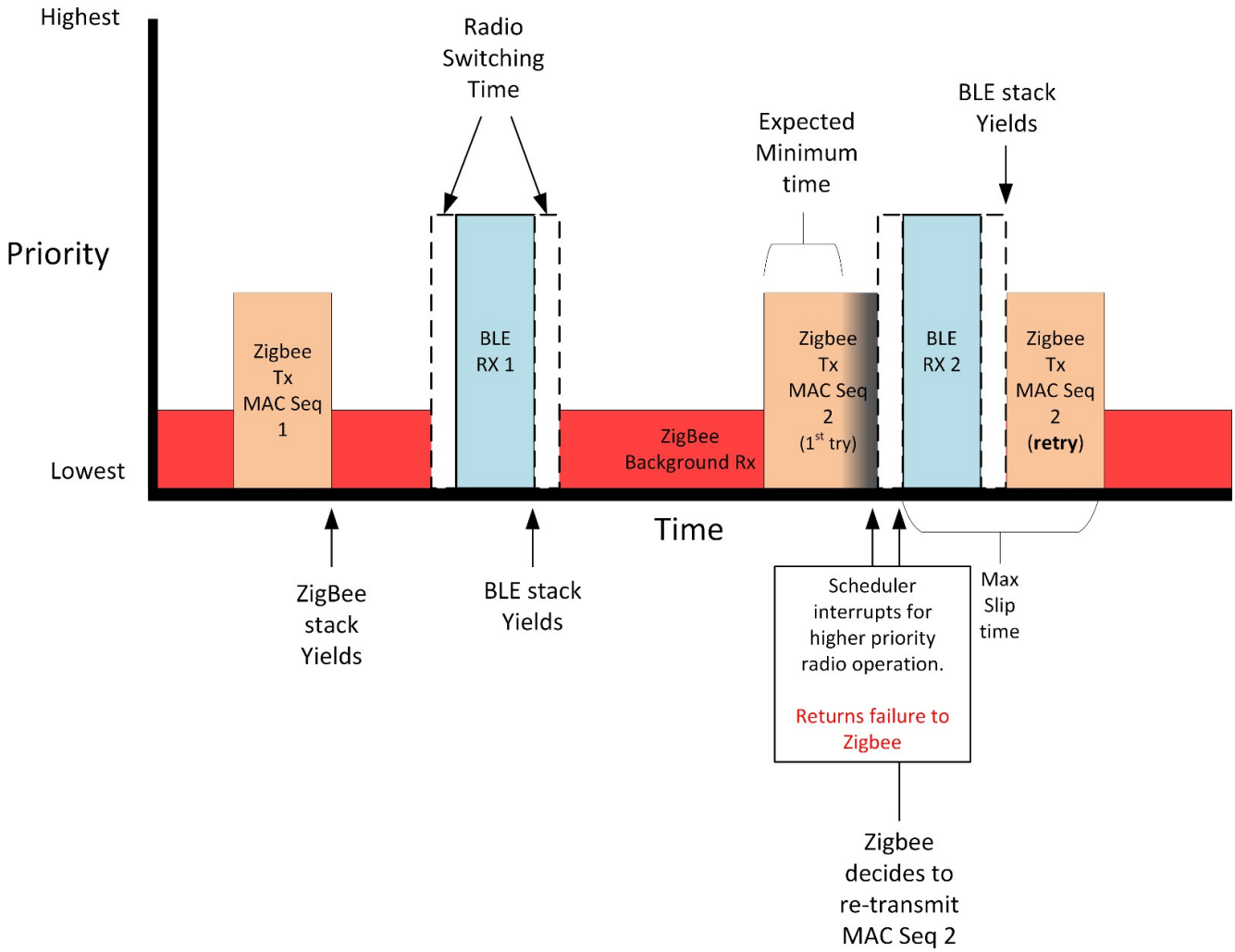
Copyright © 2025 Silicon Laboratories. All rights reserved.

this action may slip its start time. The radio scheduler determines that given the expected duration of the Bluetooth LE radio operation the Zigbee operation can start after that event and still be within the slip time indicated by the Zigbee stack.

If all goes as expected, the Zigbee transmit operation will have its first attempt occur without any failures due to scheduling.

Priority Interruption Example

This example illustrates a higher priority operation interrupting a lower priority one.



This example starts in the same way as the previous example. Zigbee and Bluetooth LE both have a radio operation that is scheduled without any collision.

Later, the Zigbee stack decides it wants to send another packet for the Zigbee TX 2 event. The scheduler determines that it should be possible to schedule this event and service the Bluetooth LE RX 2 event later, based on the minimum time that the Zigbee TX 2 event must take. However, the Zigbee TX 2 event takes longer than expected due to a long random backoff and does not yield in time. This causes the event to collide with a higher priority radio operation, and so the Radio Scheduler interrupts the Zigbee event and returns a failure to the higher-level stack. The Bluetooth LE event occurs normally and when it is complete it voluntarily yields to any lower-priority operations.

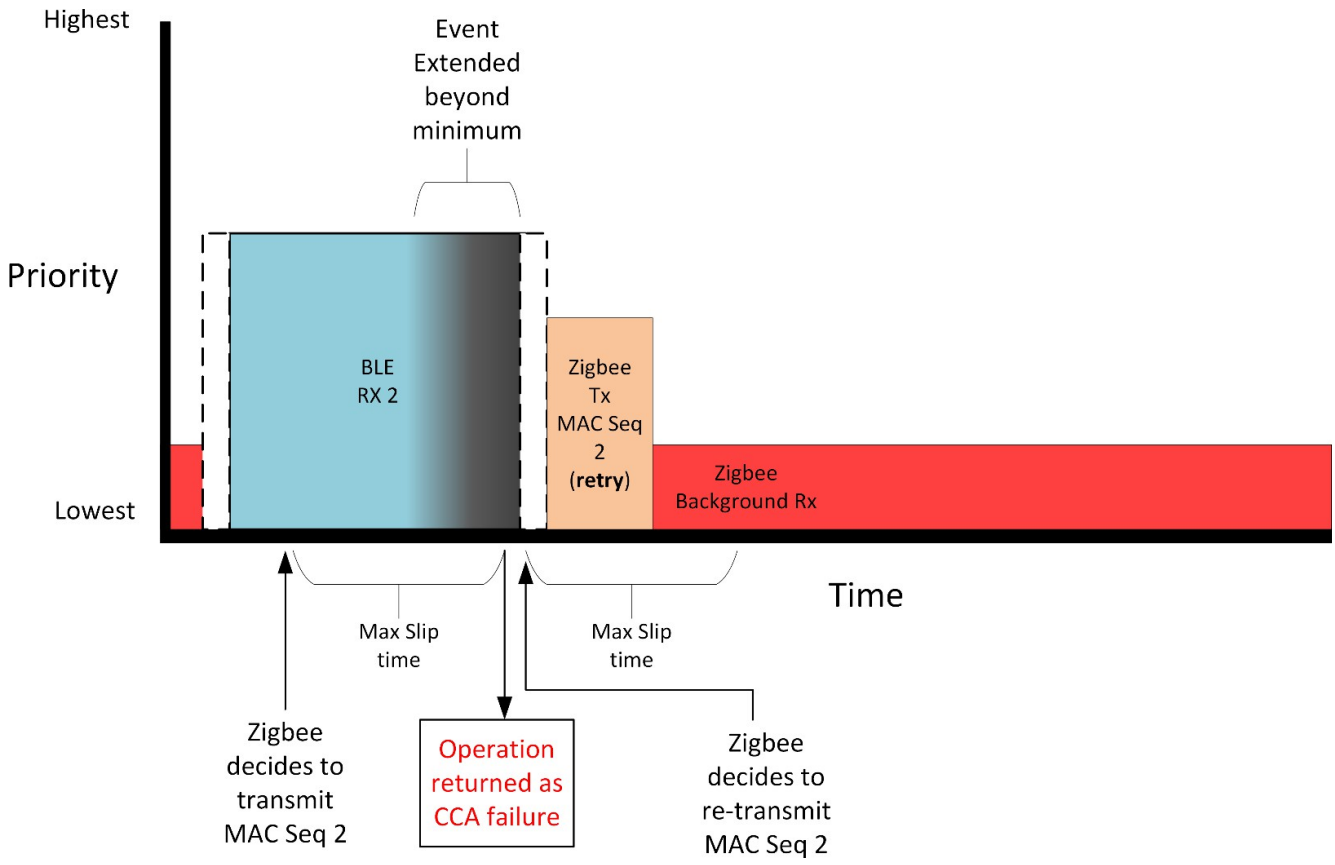
Upon receiving the failure from the radio scheduler, the Zigbee stack immediately attempts to retry the MAC message. It schedules the operation and includes a slip time. At this point the Bluetooth LE stack has priority over the radio and thus the operation cannot be started yet, but the scheduler accepts the new radio operation. The Bluetooth LE stack completes its scheduled receive and yields the radio. The scheduler then triggers the Zigbee transmit operation to occur because it is still

Copyright © 2025 Silicon Laboratories. All rights reserved.

within the slip time of the initial start operation. After the transmit completes the scheduler returns to the background receive operation.

Higher Priority Operation that is Extended

This example shows what happens when a higher priority operation takes longer than originally anticipated and causes a lower priority operation to miss its opportunity.

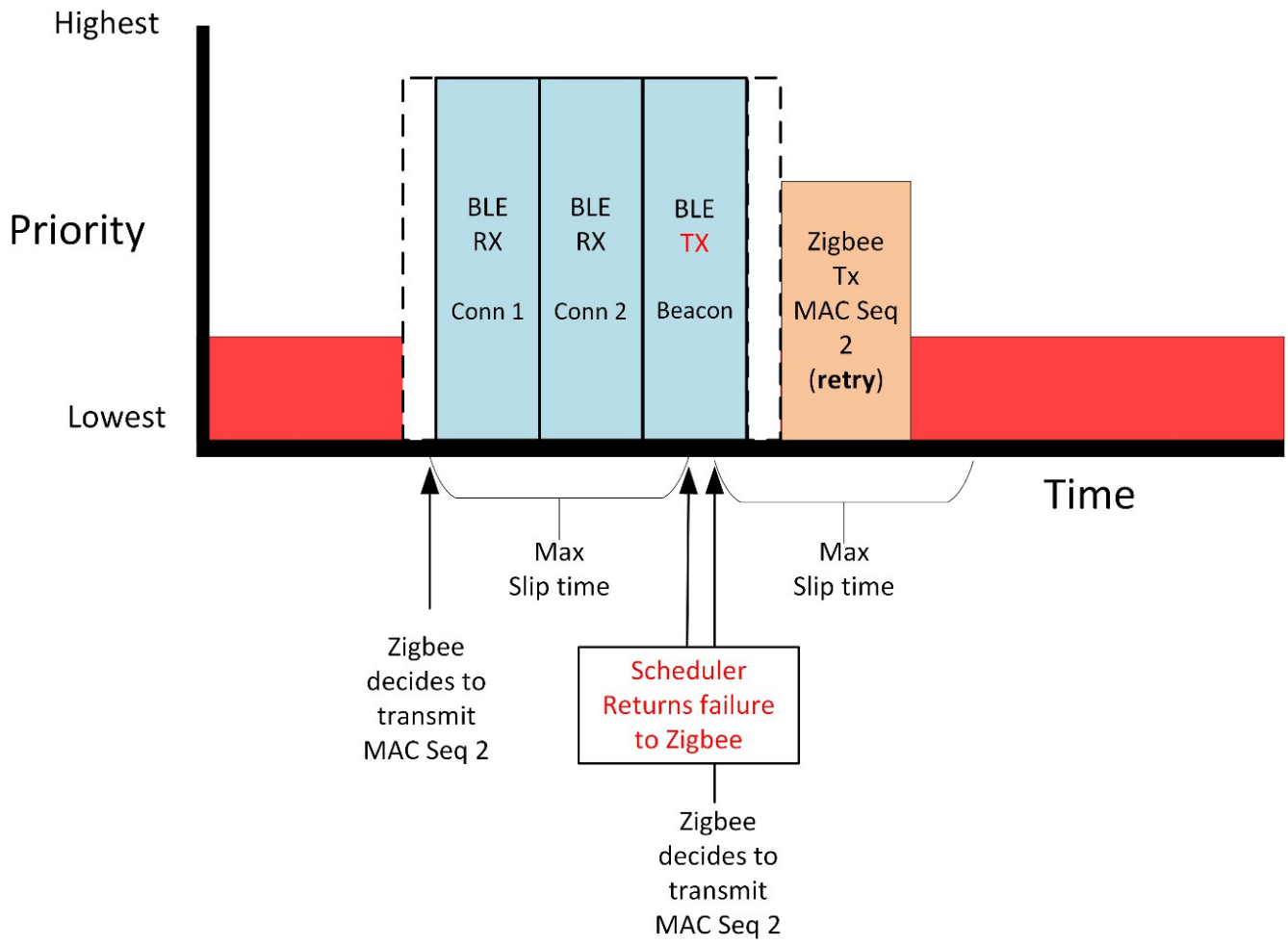


In this case, Bluetooth LE has a Scheduled receive that is currently taking place. Zigbee decides to send a packet but it cannot be run right now. The scheduler accepts the operation under the assumption that the Bluetooth LE event will complete before the end of the slip time of the Zigbee event. However, the Bluetooth LE event extends longer since additional packets are sent between the devices. The Bluetooth LE operation has priority, so the Zigbee operation eventually runs out of slip. An error is returned to the stack. Zigbee decides to re-transmit the packet. Again, the Zigbee stack indicates the operation should start now but may slip into the future. The scheduler is in the middle of changing the radio config so it cannot begin the operation immediately. Instead, it slips the radio operation start time a small amount and then executes the operation.

Higher Priority Operation Without Interruption

In this example the radio scheduler is running on a node acting as a Bluetooth LE peripheral and that node has a number of connections to different central devices. It also has a periodic advertising beacon that is transmitted. The following figure shows a case where these events are occurring virtually back-to-back and do not allow for enough time to switch back to the Zigbee radio config. Therefore it will create a period where the Zigbee stack is unable to transmit even with the slip time.

Copyright © 2025 Silicon Laboratories. All rights reserved.

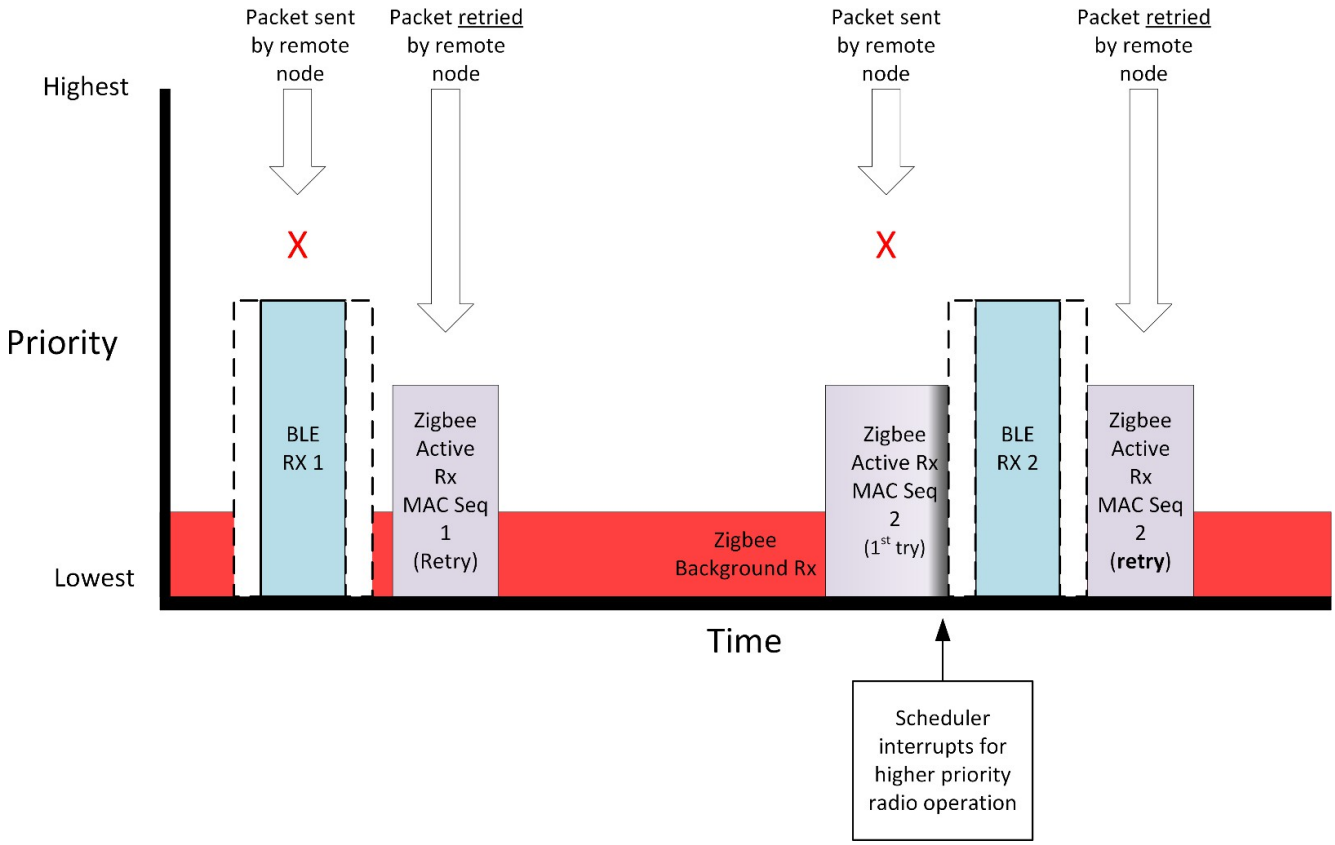


Zigbee asks the scheduler to schedule a transmit radio operation. Even though the scheduler knows that the event will fail due to scheduled higher priority operations, it still accepts the scheduled event. This is done for two reasons. First, circumstances may change and the event can be executed. Second, the stack sitting on top of the radio scheduler may try to retry the action. If the result of the failed scheduling was returned immediately then the stack's attempt to retry would be unlikely to succeed since no time has passed. Instead, by queuing the event and returning the failure *after* the slip time has expired, a retry (with its own slip time) has a better chance of success as the set of upcoming radio operations will be different.

Receive When a Higher Priority Operation is Running

This example illustrates what happens when Bluetooth LE is active and a lower priority operation will be receiving data.

Copyright © 2025 Silicon Laboratories. All rights reserved.



In the first case, when an IEEE 802.15.4 message is sent and the Bluetooth LE stack is utilizing the radio for an active receive the Zigbee stack will not be online to receive the message. However, the Zigbee sender of the message will retry in most cases and with backoffs and other timing alterations is not going to conflict with another higher priority scheduled Bluetooth receive events unlikely to collide. The Zigbee message is received successfully.

The second case shows that, in the case of an active receive, the Zigbee stack may still be interrupted and not receive (or ACK) the message. Successful communication relies on retries at the MAC or higher layer to send this message again and verify the Dynamic Multiprotocol device receives the message.

While there may be considerations for whether or not active receive should be interrupted, it is difficult for the scheduler to make that determination. In general, the robustness of the protocols should allow for messages to be successfully received even with interruptions.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Implementing Multiprotocol with an 802.15.4-Based Stack

Implementing Multiprotocol with an 802.15.4-Based Stack

This section offers general information about implementing an 802.15.4-based stack, such as Zigbee or Connect, as part of a multiprotocol applications. For specifics on how to configure plugins and other details specific to a particular protocol, see one of the following application notes:

- Dynamic Multiprotocol Development with Bluetooth and Zigbee EmberZNet SDK 6.x and Lower
- *AN1209: Dynamic Multiprotocol Development with Bluetooth and Connect*

Wireless Protocol Support

Different wireless protocols have different characteristics that have been leveraged with the design of Dynamic Multiprotocol. For example, Bluetooth Low Energy is very strict and predictable in its schedule of radio operations; advertisement and connection intervals occur at set times. In contrast, an 802.15.4 protocol is more flexible in the timing of many message events; CSMA (carrier sense multiple access) in IEEE 802.15.4 adds random backoffs so that event delays are on the order of milliseconds. This allows IEEE 802.15.4 messages to be sent around the Bluetooth Low Energy events and still be reliably received.

802.15.4 RAIL Priority

802.15.4 protocols currently have three RAIL priorities.

No.	Name	Default Setting	Exit Criterion
1	Active TX	100	MAC ACK received (or not)
2	Active RX	255	Packet filtered or MAC ACK sent
3	Background RX	255	Task with higher Priority present

If an Active TX gets executed the radio will be released at the time the corresponding MAC acknowledgement was received (or a time-out occurred).

Background RX will leave the radio in receive state ready to receive asynchronous messages. If the active RX priority is different than the background RX priority, the receive priority will be raised whenever a sync word is detected and only lowered once that packet is filtered or completed and its ACK is sent if one was requested.

Balancing Priorities

As explained in Implementing Multiprotocol with Bluetooth, by default the Bluetooth priority range is mapped into the RAIL priority range 16 - 32. In general, Bluetooth starts out using low priority (32) and dynamically increases the priority up to the maximum (16) as needed if messages are not succeeding.

As described in Radio Scheduler Examples, an 802.15.4-based stack such as Zigbee or Connect uses default RAIL priority values of 255 for background RX, 255 for active RX, and 100 for active TX.

As a result of these default RAIL priorities, in an 802.15.4 protocol-Bluetooth multiprotocol application, by default Bluetooth traffic will always take priority over 802.15.4 protocol traffic. This is a good choice for many applications, because Bluetooth traffic has stringent timing requirements, unlike 802.15.4 protocols. However, if Bluetooth traffic load is very high (for example, sending lots of data using a very small connection interval), it is possible for 802.15.4 protocol traffic to be completely blocked from access to the radio because of its lower priority and the very small windows of available radio time left by the Bluetooth traffic.

Copyright © 2025 Silicon Laboratories. All rights reserved.

> **Note**: The following information is currently only applicable to the EmberZNet Zigbee stack. Silicon Labs Connect does not yet have the API needed to change the priorities.

If you are developing an 802.15.4-based dynamic multiprotocol application, and it is important for that traffic to succeed in the presence of very high load Bluetooth traffic, you can adjust the default priorities as shown in the table below using the following API:

No.	Name	Default Setting
1	Active TX	23
2	Active RX	24
3	Background RX	255

Because the Bluetooth initially sets its RAIL priority to 32, these 802.15.4 priority settings give 802.15.4 traffic higher priority than Bluetooth initially, which gives the 802.15.4 protocol a chance to transmit or receive traffic successfully even in the presence of a very high load of Bluetooth traffic. On the other hand, Bluetooth will dynamically increase its priority if it is bumped from the scheduler by the 802.15.4 traffic, up to a high priority of 16. Thus, after allowing the 802.15.4 protocol access to the radio initially, Bluetooth will take priority on subsequent retries if necessary.

This approach allows both protocols to compromise on their use of the radio without one being able to completely dominate over the other.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Implementing Multiprotocol with RAIL

Implementing Multiprotocol with RAIL

This section offers more information about RAIL for users who consume the RAIL API directly to develop proprietary protocols. In particular, it offers details on how to work with the RAIL APIs to handle specific radio scheduler cases.

Examples with Background Receive, Yield Radio and State Transition

The fundamentals of the RAIL Multiprotocol priority system are straightforward: a radio event with a higher priority (that is, smaller in number) will always usurp any other radio events with lower priority. However, this topic becomes more complicated when considering state transitions and APIs such as `RAIL_StartRx()`, which put the radio into a certain state for an indefinite amount of time. This section provides some illustrations and examples to demonstrate how these time-unbounded states are handled, and how the application layer can use APIs such as `RAIL_YieldRadio()` to control them. The examples are as follows:

- State Transitions with a Single Protocol
- State Transitions with Two Protocols
- State Transitions with Two Protocols and Monotonically Increasing Priorities

In these examples, `RAIL_StartTx()` is the source of the TX event that interrupts the background RX. Note, however, that these examples are applicable to any radio API except for `RAIL_StartRx()`. In other words, the examples are applicable to any API that starts a radio event that is not a background RX.

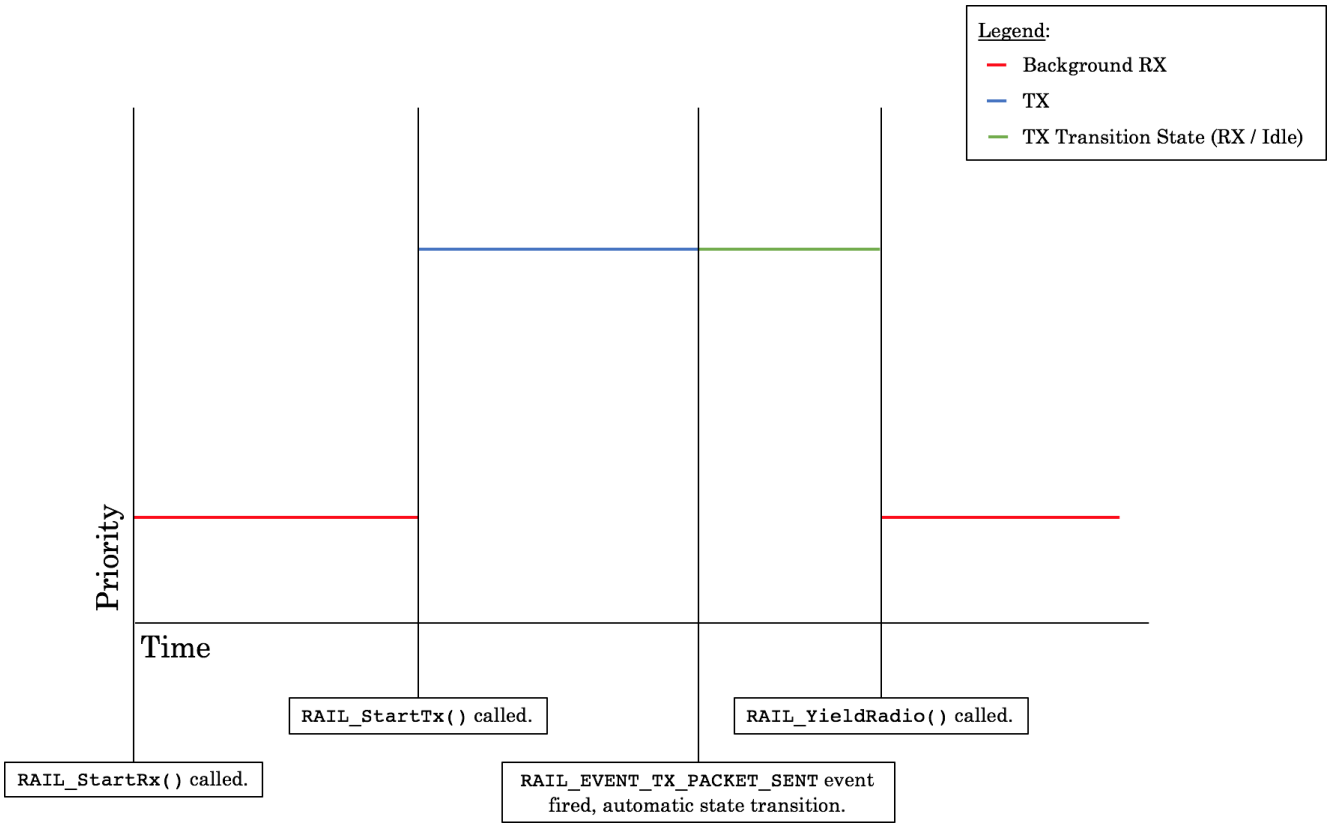
These examples illustrate expected multiprotocol behaviors regarding state transitions. To summarize:

- In a state transition, the new state is treated as an indefinite extension of the originating event at that same priority until `RAIL_YieldRadio()` is called.
- Background RX events are not affected by `RAIL_YieldRadio()`. Only `RAIL_Idle()` can permanently remove a protocol from the background RX state.
- An event with a higher priority will always usurp an event with lower priority, regardless of any other API calls.
- Only `RAIL_StartRx()` receives can be 'returned to' from a higher priority event through `RAIL_YieldRadio()` or `RAIL_Idle()`.
- All radio events other than `RAIL_StartRx()` require `RAIL_YieldRadio()` in order to end and progress to the next event.
- The call to `RAIL_YieldRadio()` cannot be replaced with `RAIL_Idle()`. `RAIL_Idle()` clears out *all* events for the given protocol.

State Transitions with a Single Protocol

This first example examines the behavior of the radio with a single protocol (that is, where the same `RAIL_Handle_t` is used for all radio function calls). The radio starts in RX with an initial call to `RAIL_StartRx()`, then moves into a TX with a higher priority call to `RAIL_StartTx()`. It is important to note that after the transmit is done, the radio transitions to the state specified by `RAIL_SetTxTransitions()`, and it stays in the state indefinitely at the same priority and channel as the TX until `RAIL_YieldRadio()` is called. After that, the radio returns to RX, with the initially specified priority and channel.

Copyright © 2025 Silicon Laboratories. All rights reserved.



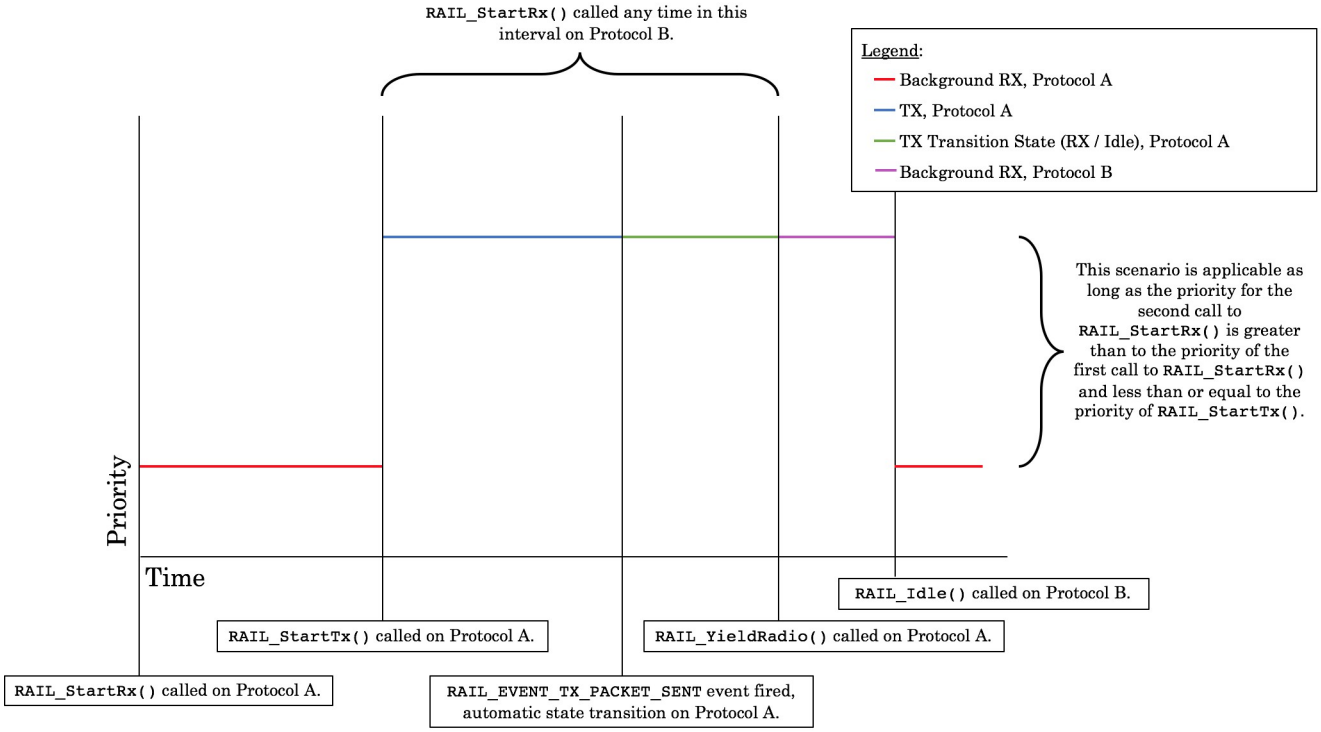
The need to actively yield the radio, and thus the `RAIL_YieldRadio()` API were necessary largely due to ACK'ing. The design philosophy is that, because both a TX and a received ACK are viewed as part of the same transaction, if a node transmits and expects an ACK it should be able to both transition to RX and continue listening for the ACK as part of the same operation (and therefore same priority) as the original TX. In general, however, RAIL on its own cannot know whether or not an ACK is required. This can depend on other factors, such as packet contents, or other application logic, and so cannot be simply determined by checking whether ACK'ing has been configured with `RAIL_ConfigAutoAck()`.Therefore, discretion as to when a radio transaction is complete is left to the application/stack.

In the case that an ACK is not required, Silicon Labs recommends calling `RAIL_YieldRadio()` as part of handling the `RAIL_EVENT_TX_PACKET_SENT` event. Doing this causes the green line in the above figure to be minimized down to the interrupt latency time. If the application does expect an ACK, `RAIL_YieldRadio()` should be called when the ACK is received or has been deemed to time out.

State Transitions with Two Protocols

This scenario is similar to the first scenario regarding state transitions after TX, but introduces another protocol.

Copyright © 2025 Silicon Laboratories. All rights reserved.



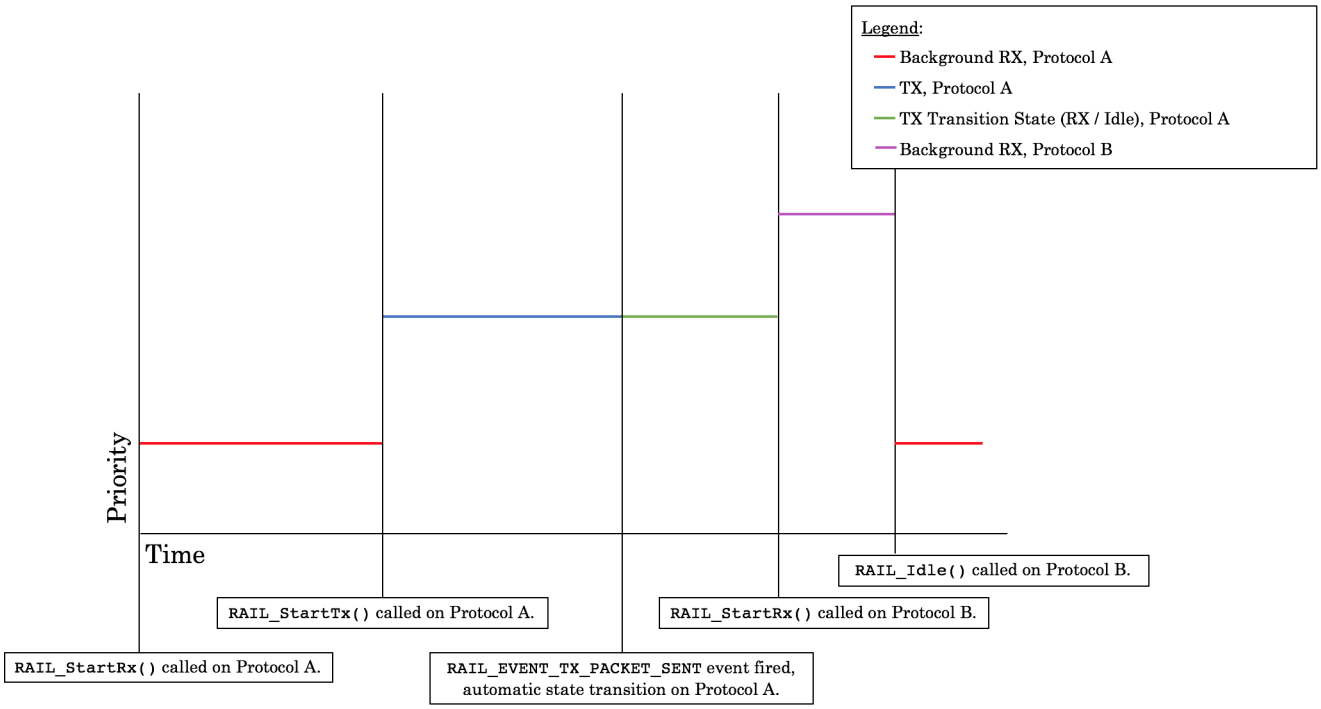
In this situation, it is important to note that `RAIL_StartRx()` can be called at any time during the TX transaction. As long as its priority is less than or equal to the priority of the TX, the RX will not come into effect until the application calls `RAIL_YieldRadio()` on Protocol A. When `RAIL_StartRx()` is called during the TX, the RX is merely added to the queue of events to be handled.

Another key point is that, although `RAIL_YieldRadio()` on Protocol A will transition from TX on Protocol A to RX on Protocol B, a `RAIL_Idle()` on Protocol B is required to transition from the RX on Protocol B to the RX on Protocol A. The philosophy here is that Background RXs can't really be yielded, since the event is never really over. The only way to exit is to stop the Background RX with a call to `RAIL_Idle()`.

State Transitions with Two Protocols and Monotonically Increasing Priorities

The final scenario is nearly identical to the previous one, except the call to `RAIL_StartRx()` on Protocol B is at a higher priority than the call to `RAIL_StartTx()` on Protocol A.

Copyright © 2025 Silicon Laboratories. All rights reserved.



In this case, since the priority of the second `RAIL_StartRx()` is higher than the priority of the call to `RAIL_StartTx()`, a call to `RAIL_YieldRadio()` is no longer necessary. Because the second `RAIL_StartRx()` is at a higher priority, it usurps the `RAIL_StartTx()` event, taking control of the radio and removing the TX event from the state. At any time during that RX on Protocol B, `RAIL_Idle()` can be called to return to the RX on Protocol A, just as in the previous example.

Note here, that when the application calls `RAIL_Idle()` on Protocol B's RX, the application does not return to the TX Transition of Protocol A. Instead, it goes right to the background RX, even though the application never called `RAIL_Idle()` on Protocol A's TX. For Scheduled radio operations (that is, any radio operation started by an API other than `RAIL_StartRx()`), once a radio event is usurped by a higher priority event, it is removed entirely and will not be returned to later. Only Background receives, started by `RAIL_StartRx()`, can be maintained in the background and 'returned to' through a call to `RAIL_YieldRadio()` or `RAIL_Idle()`.

To emphasize the difference between `RAIL_YieldRadio()` and `RAIL_Idle()` it is important to note that, for all these examples, the call to `RAIL_YieldRadio()` cannot be replaced with `RAIL_Idle()`. `RAIL_Idle()` clears out *all* events for the given protocol – both the Background (that is, started by `RAIL_StartRx()`) and Scheduled (that is, started by APIs other than `RAIL_StartRx()`) operations. `RAIL_Idle()` would indeed still cause the application to exit out of the TX transition state, but it would also clear out the Background RX, causing the application to return to idle, not RX.

Copyright © 2025 Silicon Laboratories. All rights reserved.

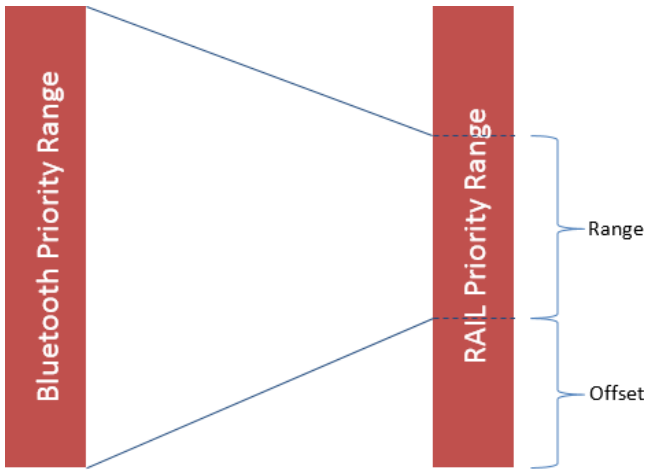
Implementing Multiprotocol with Bluetooth

Implementing Multiprotocol with Bluetooth

For details on how the RAIL/Bluetooth light/switch multiprotocol example was implemented, and for more information on developing a multiprotocol application with your own protocol on RAIL, see Dynamic Multiprotocol Development with Bluetooth and Proprietary Protocols on RAIL in GSDK v3.x and Higher.

Bluetooth Priorities

As opposed to Zigbee with statically defined priorities for different operation types, Bluetooth uses a range and offset approach to assign all tasks to a given area of the priority spectrum.



In this example the Bluetooth priority range, which itself spans from 0 to 255, is mapped to a limited portion of the shared RAIL priority space.

Unlike Zigbee, Bluetooth has much more stringent timing requirements where missing a given slot may result in a connection terminating. Also Bluetooth has a range of different tasks like (potentially multiple) connections, advertisement, scanning, and Periodic Advertising with Responses (PAwR) transmissions and receptions.

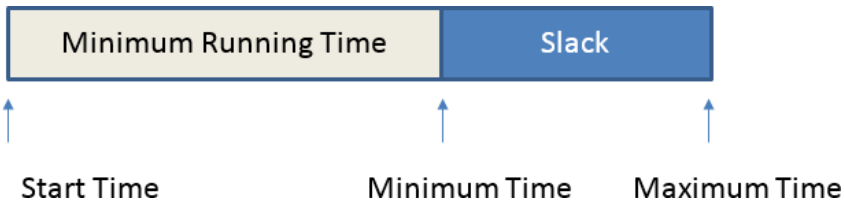
No.	Name	Default Setting	Exit Criterion
1	Connection	135 to 0	Connection Event Ends
2	Connection Initiation	55 to 15	Initiation Window Ends
3	Advertisement	175 to 127	Advertisement Event Ends
4	Scanner	191 to 143	Scan Window Ends
5	PAwR TX	15 to 5	Advertiser: PAwR Response Slot Delay Ends Synchronizer: PAwR Response Slot Ends
6	PAwR RX	20 to 10	Advertiser: PAwR Response Slot Ends Synchronizer: PAwR Response Slot Delay Ends

In order to handle this the Bluetooth scheduler, whose priorities are mapped to the RAIL radio scheduler, takes into account the following parameters for each task:

1. Start Time

Copyright © 2025 Silicon Laboratories. All rights reserved.

Minimum time
3. Maximum time
4. Priority



If the start time is moved the total running time is reduced respectively, that is the slack is reduced. Also priorities can be dynamically adjusted.

Connections

Connections have a relatively high priority. The start time of a connection cannot be moved.

The priority is dynamically increased by the Bluetooth scheduler the closer the connection gets to the supervision timeout, and reaches the maximum priority close to it. A TX packet in the TX queue also increases the priority of a connection.

Connection Initiation

Connection initiation scans advertisements from target device to establish a connection. It has a higher priority compared to a scanner to allow more robust connection establishment.

Advertisements

Advertisements by default have a lower priority and their start point can be moved. Start time and Maximum time are defined by the advertisement interval.

If an advertisement could not be sent out, the priority of advertisements increases slowly and is reset once an advertisement was successfully sent.

Scanner

By default, these tasks have the lowest priority. Start, minimum and maximum time are defined by the scanning interval and window size. Scanning can continue even when interrupted by a higher priority task. If this happens the scan time is accumulated to make sure the desired scan window size is reached at each scanning interval.

As with advertisements, the priority is increased in case the desired scan interval or window size could not be previously met. It is reset back to its initial priority once the scan interval or window size has been met.

Periodic Advertising with Responses (PAwR)

Sending Periodic Advertising with Responses has the highest priority by default over all other Bluetooth tasks, followed by receiving responses in PAwR to maintain synchronization in an electronic shelf label (ESL) network.

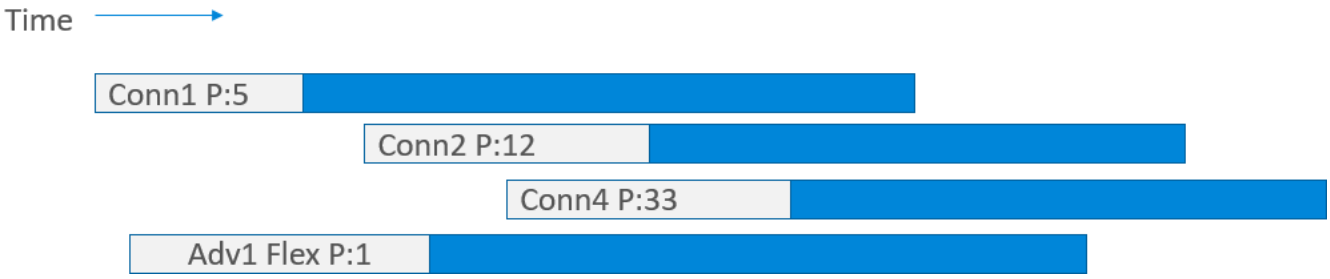
A PAwR task priority is increased if the task scheduling fails twice in a row. The priority is either increased by 1/6th of the priority range, or at least by one until the maximum priority has been reached. The task priority is reset back to the minimum after successful scheduling. The same procedure applies to both PAwR advertiser and synchronizer in both directions.

Example of Bluetooth Scheduler Operation

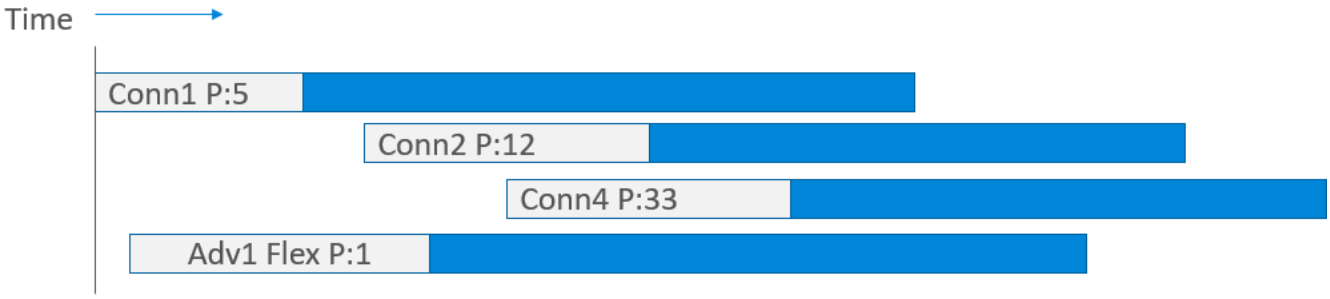
This example illustrates how the Bluetooth scheduler will schedule three connection tasks and one advertisement task, each holding different priorities. In the following figures the gray part indicates the minimum runtime a task requires and the

Copyright © 2025 Silicon Laboratories. All rights reserved.

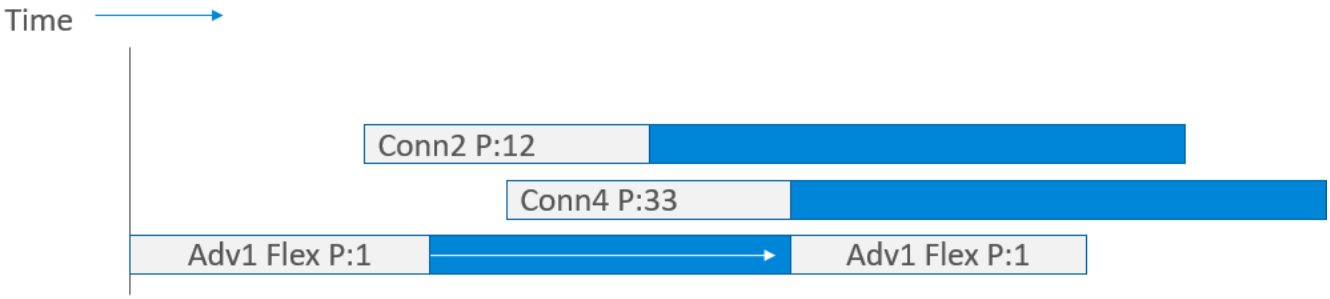
blue part indicates the maximum runtime the task can use and, if flexible, the region where the task can be moved. The following figure shows the initial setup.



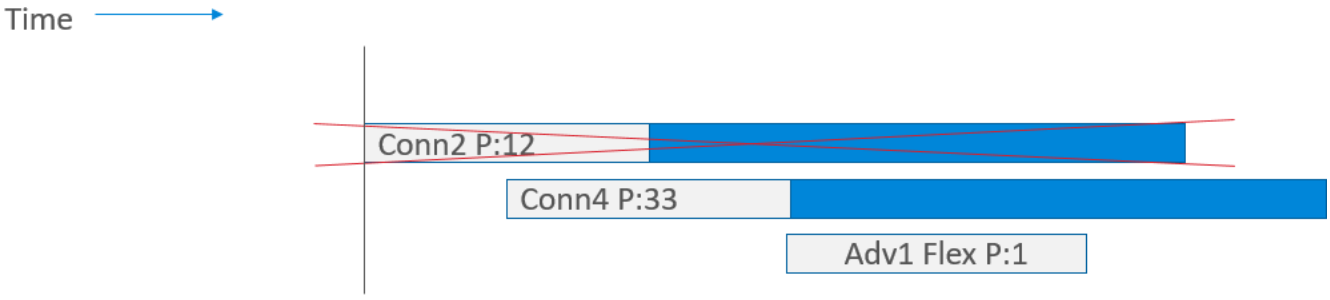
As shown below, Conn1 is the first task to run as it does not overlap with any higher priority task.



Adv1 overlaps with the higher priority Conn2. Adv1 is flexible and therefore gets moved in, as illustrated in the following figure.

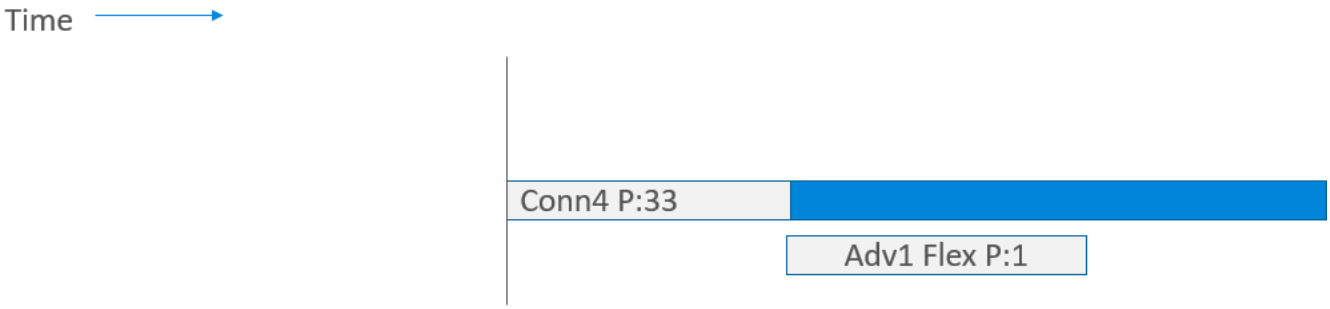


Conn2 overlaps with higher priority task Conn4. As Conn2 is not flexible, the scheduling of Conn2 fails.



Conn4 does not overlap with other tasks, therefore Conn1 end is adjusted to stop before Conn4 starts.

Copyright © 2025 Silicon Laboratories. All rights reserved.



Finally, Adv1 is run. Conn4 is adjusted to end before Adv1 starts.



Modifying Priorities

The "sl_bt_configuration_t" (v3.x)/"gecko_configuration_t" (v2.x) struct defines the sl_bt_stack_config_t struct, which contains the field “bluetooth.linklayer_priorities” that is a pointer to the priority configuration. If the pointer is NULL then the stack uses its default priorities as listed in Bluetooth Priorities, as well as this section.

In case the pointer is not null it must point to a struct of priority settings as defined below:

```
typedef struct {
  uint8_t scan_min;
  uint8_t scan_max;
  uint8_t adv_min;
  uint8_t adv_max;
  uint8_t conn_min;
  uint8_t conn_max;
  uint8_t init_min;
  uint8_t init_max;
  uint8_t rail_mapping_offset;
  uint8_t rail_mapping_range;
  uint8_t _reserved;
  uint8_t adv_step;
  uint8_t scan_step;
  uint8_t pawr_tx_min;
  uint8_t pawr_tx_max;
  uint8_t pawr_rx_min;
  uint8_t pawr_rx_max;
} sl_bt_bluetooth_ll_priorities;
```

The parameters `scan_min`, `can_max`, `adv_min`, `adv_max`, `conn_min`, `conn_max`, `init_min`, and `init_max` define the minimum and maximum priorities for scanning, advertisement, connections, and initiations respectively. The priorities will move between the min and max boundaries as described in:

- Connections
- Connection Initiation
- Advertisements

Copyright © 2025 Silicon Laboratories. All rights reserved.

- Scanner

The RAIL mapping parameters, `rail_mapping_offset` and `rail_mapping_range`, define how the Bluetooth link layer priorities are mapped to the global RAIL radio scheduler priorities. The mapping of these values can be seen in Bluetooth Priorities. The default for both `rail_mapping_offset` and `rail_mapping_range` is 16.

The `adv_step` and `scan_step` parameters define the step size when the priority of scanning and advertising is changed dynamically.

Finally, the parameters `pawr_tx_min`, `pawr_tx_max`, `pawr_rx_min`, and `pawr_rx_max` define the priority range for the PAwR advertiser and synchronizer TX and RX events in each sub-event.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Overview

Non-Volatile Data Storage

This section offers an introduction to non-volatile data storage and describes how to use NVM3 data storage.

- Non-Volatile Data Storage Fundamentals: Introduces non-volatile data storage using flash and the three different storage implementations offered for Silicon Labs microcontrollers and SoCs: Simulated EEPROM, PS Store, and NVM3.
- Using Third Generation Non-Volatile Memory (NVM3) Data Storage: Explains how NVM3 can be used as non-volatile data storage in various protocol implementations.
- Bringing Up Custom Devices for the EFR32MG and EFR32FG Families: Describes how to initialize a piece of custom hardware (a 'device') based on the EFR32MG and EFR32FG families so that it interfaces correctly with a network stack. The same procedures can be used to restore devices whose settings have been corrupted or erased.
- Using Tokens for Non-Volatile Data Storage (PDF): Describes tokens and shows how to use them for non-volatile data storage in EmberZNet PRO and Silicon Labs Flex applications.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Non-Volatile Data Storage Fundamentals

Non-Volatile Data Storage Fundamentals

NOTE: This section replaces *UG103.07: Non-Volatile Data Storage Fundamentals*. Further updates to this user guide will be provided here.

This guide provides a general introduction to non-volatile data storage using flash, with a focus on the three different dynamic data storage implementations offered for Silicon Labs microcontrollers and radio SoCs (Systems on Chip). It offers a comparison of the three implementations and provides recommendations on when to use each. Additional detail on using the various data storage implementations may be found in the following documents:

- *AN1154: Using Tokens for Non-Volatile Data Storage*
- *AN703: Using Simulated EEPROM Version 1 and Version 2 for the EFR32 SoC Series 1 Platform*
- Using Third Generation Non-Volatile Memory (NVM3) Data Storage

Silicon Labs' *Fundamentals* series covers topics that project managers, application designers, and developers should understand before beginning to work on an embedded networking solution using Silicon Labs chips, networking stacks such as EmberZNet PRO or Silicon Labs Bluetooth, and associated development tools. These guides can be used as a starting place for anyone needing an introduction to developing wireless networking applications, or who is new to the Silicon Labs development environment.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Introduction

Introduction

Non-volatile memory (NVM) is memory that persists even when the device is power-cycled. On Silicon Labs microcontrollers and radio SoCs, the NVM is implemented as flash memory. In many applications the flash is not only used to store the application code but also to store data objects that are written and read frequently by the application. As flash memory can only be erased a limited number of times, several methods exist to efficiently read and write non-volatile data without wearing out the flash.

Some data is considered manufacturing data that is written only once at manufacturing time. This document is concerned with dynamic data that changes frequently over the life of the product.

This document provides an introduction to the main design options for dynamic data storage in microcontrollers and radio SoCs, along with guidelines on what factors affect flash lifetime. In addition it introduces the main flash data storage implementations offered by Silicon Labs:

- NVM3
- Simulated EEPROM version 1 (SimEEv1) and version 2 (SimEEv2)
- Persistent Store (PS Store)

Copyright © 2025 Silicon Laboratories. All rights reserved.

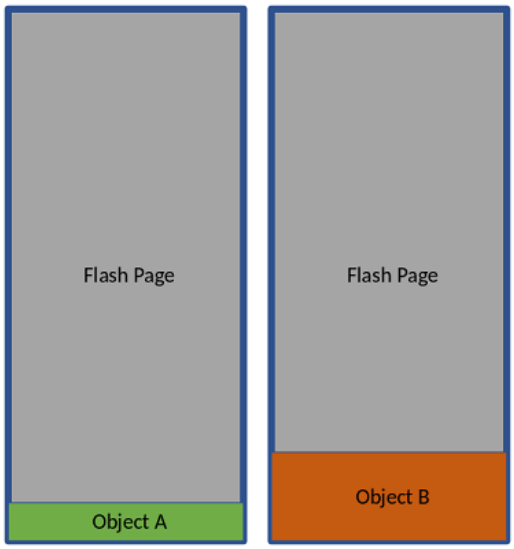
Implementations of Non-Volatile Data Storage

Implementations of Non-Volatile Data Storage

This chapter introduces some of the challenges and design options when implementing non-volatile data storage in flash memory. It describes at a high level how non-volatile data storage is implemented in flash memory in PS Store, SimEEv1/v2, and NVM3.

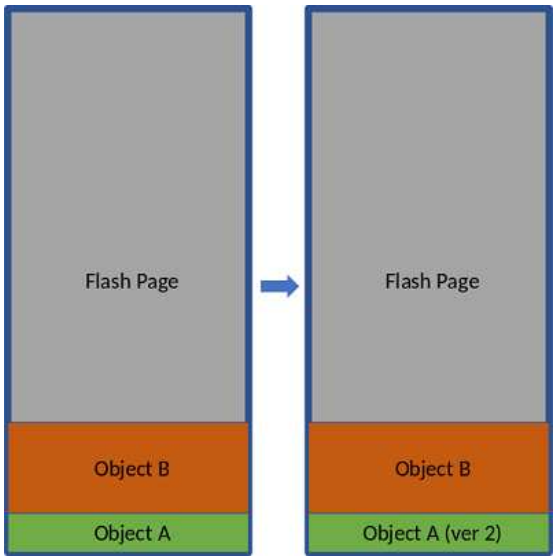
Basic Implementations

One of the characteristics of flash memory is that it is writable in smaller pieces, usually 32-bit words, while it can only be erased in larger chunks, usually pages of several kilobytes. When using flash as data storage, the most straightforward implementation option would be to store each data object in its own flash page, as shown in the following figure. This way each object can be erased and re-written without influencing the other data objects. Usually the data objects are much smaller than the page size, and this solution is not an effective way of using the available flash space.

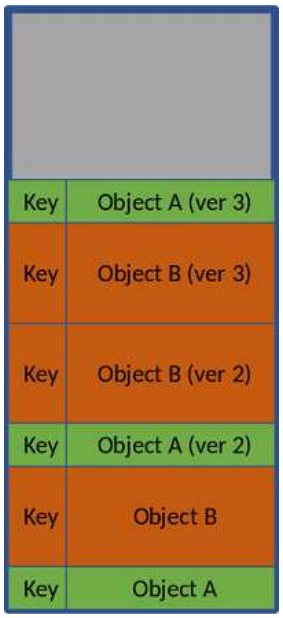


To avoid wasting flash space we can store several data objects in one flash page, as shown in the following figure. This solution then introduces a challenge when we want to write a new value to one of the data objects. In that case the page must be erased before all objects are written back to the page again, including the object we changed. As flash memory can only endure a limited amount of flash erases before the flash cells are worn out, this solution results in a very limited device lifetime.

Copyright © 2025 Silicon Laboratories. All rights reserved.



To avoid erasing the flash page for every object write, we can instead write new versions of each object to new empty locations in the flash page. This is a simple form of wear-levelling that reduces the number of page erases. However, this requires that we store some identification information along with the object data that tells us what object the data belongs to, so we know how to find the latest version of the data object. This is illustrated in the following figure, where a key is added to each version of the object data to identify what object the data belongs to. When accessing an object we then need to search through the flash page for the most recent version of the object. In this case the newest version is the one with the highest address, as we start writing from the lowest address of the page.

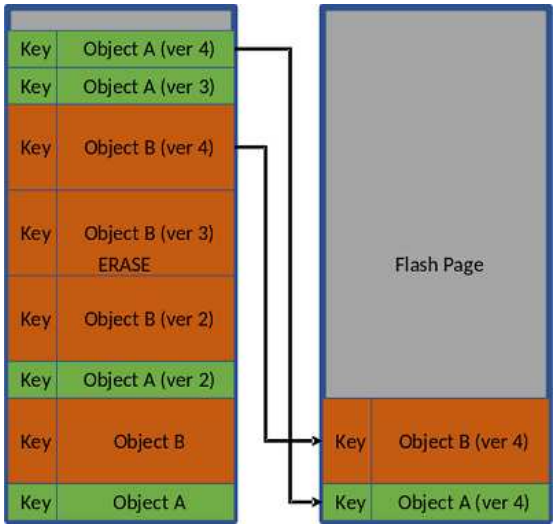


Handling Resets and Power Failures

As we fill up the flash page with new versions of the data objects, eventually no room is left to write new object data. At this point we need to erase the page and start over by writing only the latest versions of each object to the flash page. In many applications, however, power failures or resets can happen at any time, and we should not risk losing data if this occurs. If a reset occurs after the flash page is being erased, but before the data objects are written back, then we will lose this data. To handle this case we introduce a second page, to which we copy the latest version of the data objects, before erasing the original page, as shown in the following figure Then we can start filling the second page with data. When

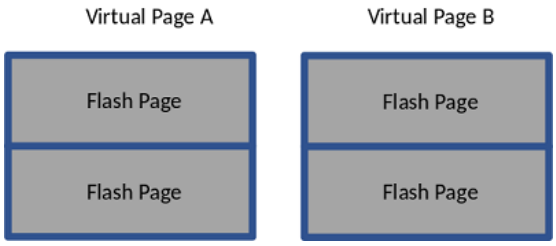
Copyright © 2025 Silicon Laboratories. All rights reserved.

the second page is full we move the latest data back to the first page and so on. This mechanism, where the storage is alternated between two flash pages, is how PS Store operates.



Introducing Virtual Pages

In some applications we write to data objects frequently, and the flash pages therefore also need to be erased frequently. As the data objects in the implementation so far are only spread across two flash pages, each page will frequently get erased and the flash lifetime will be limited. To increase the lifetime we can use more flash pages to store the data objects. In this example, instead of two physical pages, we operate with two virtual pages (A and B) that each consist of several physical flash pages. The virtual pages are erased and written to as if they were one larger flash page. The difference is simply that each virtual page is bigger and we can write more data before we need to erase the virtual page, hence the lifetime is extended. In addition to increasing flash lifetime, using several flash pages per virtual page allows you to store more or larger objects. SimEEv1 uses this design, with each virtual page consisting of two flash pages, A and B, as shown in the following figure.



In some applications the time it takes to write a non-volatile data object must be minimized so as to not interfere with the timing of other critical operations. If an object write is triggered when a virtual page is full, the latest version of all objects must first be copied to the new virtual page before writing the new object version in question to the new page. All objects must be copied over immediately to allow the first page to be quickly erased so we can move data there in case of a failure. Copying all objects at once increases the worst-case object write time.

To reduce the write times, a third virtual page can be introduced that is always kept erased. Instead of copying over the latest version of every object when the first page is full, we can instead copy over only some of the objects. The rest of the objects are copied over as a part of subsequent write operations. This way we spread the copy process to the new page over more write operations, hence each write operation takes less time to complete. With this approach we have live object data spread out across two virtual pages and the third page is always kept erased, so we have somewhere to move the data to in case of a failure. SimEEv2 uses this implementation with 3 virtual pages where each virtual page consists of 6 flash pages.

Copyright © 2025 Silicon Laboratories. All rights reserved.

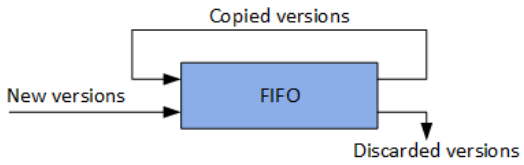
Basic Storage

Basic storage is defined as the size of all the latest version of all objects, including any overhead stored with them. Except for NVM3, every time a flash page or virtual page is erased we must first move the basic storage over to a new page. The basic storage size is important, as it determines how much flash space is left over in a page to store any new versions of the object data. If the basic storage takes up almost the entire page, we can only write a few new object versions before we need to move to a new page and erase the old one. This leads to frequent page erases and a short flash lifetime.

NVM3 only copies unique objects that are stored in the page that will be erased. If newer versions of the object exist in other pages, the object will not be copied.

FIFO Model

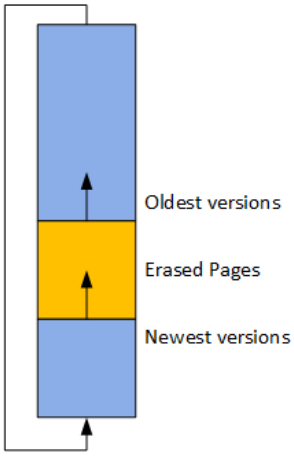
The flash data storage implementations can be modelled as a First-In First-Out (FIFO) buffer (see the following figure), where we write new versions to the input of the FIFO. As the FIFO fills up, we need to free up space by erasing one or more pages at the end of the FIFO. Before we erase a page, we need to copy any object versions for which no newer version of the same object exists in the rest of the flash pages. Other object data can be discarded, because newer versions exist. To maximize flash lifetime we want to copy as few object versions as possible, so that most of the writes to the flash memory are new versions of the objects. If the FIFO is implemented over a large flash space, it is more likely that the new version of the object has been written and that object versions at the end of the FIFO can be discarded. In this case the flash page can be erased with few or no object versions copied.



A drawback of using a few virtual pages is that the available memory for the FIFO is limited to only the virtual pages that can hold live data. For implementations using two virtual pages, like SimEEv1, this means only half the storage space is used for the FIFO, while for SimEEv2 two thirds of the storage space is used for the FIFO.

To allow a higher portion of the storage space to be used for the FIFO, we can instead implement the FIFO as a circular buffer over the entire flash storage space allocated, as shown in the following figure. In this implementation we always need to keep enough erased pages in front of the leading edge of the buffer to write the largest-sized object in case of a failure. When the FIFO fills up to reach the critical number of erased flash pages, we copy over any object versions that have not been superseded and erase the page at the back of the FIFO. This means that, instead of keeping a full virtual page erased, we only need to keep enough space erased to fit the largest-sized object. We can use the rest of the space for the FIFO. NVM3 is implemented as a circular buffer implemented over the entire storage space, thus increasing flash lifetime compared to implementations using smaller virtual pages.

Copyright © 2025 Silicon Laboratories. All rights reserved.



Counter Objects

For some types of data, the storage format can be optimized for the flash medium. EFR32 Series 0 and 1 can write each word two times, allowing for half word updates. Series 2 devices allow just one write per word.

For example, counter values are usually incremented by 1 or some other low value every time they are written. Normally this means that we would have to write the entire counter value in addition to identification bytes every time the counter is incremented. We can optimize this by only storing a start value in addition to the identification value the first time a counter is written. Then we reserve a number of the words following the initial value for writing increments. The flash words in Silicon Labs devices EFR32 Series 0 and 1 can write words twice between each erase by keeping a 1 in the bits that are not to be changed.

As an example, assume we are writing two 16 bit values, 0xAAAA and 0x5555. To safely write them in the same flash word this method can be used:

- Write 0xFFFFAAAA (word in flash becomes 0xFFFFAAAA)
- Write 0x5555FFFF (word in flash becomes 0x5555AAAA)

Note that there is a maximum of two writes to the same word between each erase due to a physical limitation of the flash.

To find the current counter value, we start with the initial value and add the increment values in the halfwords following the initial value. This means that we only need to write one halfword for each increment, instead of the whole counter value and identification value. NVM3 and SimEEv1/v2 support counter objects.

Indexed Objects

When storing data such as arrays in flash, we often only update one index at a time. If we store the whole array as one regular object, we must write the whole array to flash even if just one index is updated. Instead of storing the entire array in one object, we can divide each data array over multiple objects and only update the objects holding the changed array indexes. While it is possible to split arrays into multiple objects manually for all the Silicon Labs storage implementations, SimEEv1/v2 allow all of the indexes to share one object key. The index entry into the array to be looked up is then provided as a separate parameter.

NVM3 does not support reading and writing parts of an object. If the application wants to access indexes individually, each index must be accessed using a unique key.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Dynamic Data Storage Implementations

Dynamic Data Storage Implementations

This chapter introduces some of the dynamic data storage implementations offered by Silicon Labs, including SimEEv1 and SimEEv2, PS Store, and NVM3.

SimEEv1/v2

SimEEv1/v2 can be used with EmberZNet PRO and Silicon Labs Connect (installed with the Flex SDK). SimEEv1 uses two virtual pages with a fixed total size of 8 kB, while SimEEv2 uses three virtual pages with a fixed total size of 36 kB.

One characteristic of SimEEv1/v2 is that all objects are defined with size and type at compile time, hence a new object cannot be created or deleted at runtime.

Silicon Labs provides a plugin for upgrading SimEEv1 data to SimEEv2.

Information about the SimEEv1/v2 implementations is found in *AN703: Using Simulated EEPROM Version 1 and Version 2 for the EFR32 Series 1 SoC Platform*.

PS Store

PS Store can be used with Bluetooth devices on all platforms except for EFR32 Series 2. PS Store API commands are used to manage user data in PS keys in the flash memory of the Bluetooth device. User data stored within the flash memory is persistent across reset and power cycling of the device. The persistent store size is 2048 bytes and uses two flash pages for storage. As Bluetooth bondings are also stored in this area, the space available for user data also depends on the number of bondings the device has at the time. The size of a Bluetooth bonding is around 150 bytes. With its simple implementation and few storage flash pages, PS Store is the smallest of Silicon Labs' non-volatile storage options. PS Store allows objects to be created and deleted at runtime.

Information about the PS Store APIs may be found in the *Bluetooth API Software Reference Manual*.

NVM3

The third generation Non-Volatile Memory (NVM3) data storage driver is an alternative to SimEEv1/v2 and PS Store. The NVM3 driver provides a means to write and read data objects (key/value pairs) stored in flash. Wear-leveling is applied to reduce erase and write cycles and maximize flash lifetime. The driver is resilient to power loss and reset events, ensuring that objects retrieved from the driver are always in a valid state. Because NVM3 can be used with several protocols, such as Bluetooth and EmberZNet PRO, it allows a single data storage instance to be shared in Dynamic Multiprotocol (DMP) applications.

Some of the main features of NVM3 are as follows:

- Key/value pair data storage in flash
- Runtime creation and deletion of objects
- Persistence across power loss and reset events
- Wear-leveling to maximize flash lifetime
- Object sizes from 0 to 4096 bytes
- Configurable flash storage size (minimum 3 flash pages)
- Cache with configurable size for fast object access
- Data and counter object types
- Compatibility layers with token and PS Store APIs provided
- Single shared storage instance in multiprotocol applications
- Repack API to allow application to run clean-up page erases during periods with low CPU load

Detailed information on NVM3 is documented in the EMDRV->NVM3 section of the Gecko HAL & Driver API Reference Guide. Users who are accessing NVM3 through its native API should refer to this API reference guide for information. Users

Copyright © 2025 Silicon Laboratories. All rights reserved.

who are developing dynamic multiprotocol applications should refer to Using Third Generation Non-Volatile Memory (NVM3) Data Storage.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Comparing Non-Volatile Data Storage Implementations

Comparing Non-Volatile Data Storage Implementations

The following table presents an overview of the main features of the various Silicon Labs Non-Volatile Data Storage implementations.

Feature	NVM3	SimEEv1	SimEEv2	PS Store
Compatible devices	EFM32, EFR32	EFR32 Series 1	EFR32 Series 1	EFR32 Series 1
Compatible radio protocols	EmberZNet, Connect, Z-Wave, OpenThread, Bluetooth	EmberZNet, Connect	EmberZNet, Connect	Bluetooth
Flash used for storage	3 or more flash pages	8 KB	36 KB	4 KB
Virtual pages	NA	2	3	NA
Max basic storage (Sum of all object data with overhead)	Variable (see Using Third Generation Non-Volatile Memory (NVM3) Data Storage for details)	2 KB	8 KB	2 KB
Max number of objects	Limited by basic storage	254	254	Limited by basic storage
Max object size (bytes)	Configurable 208-4096	254	254	56
Object creation/deletion	Runtime	Compile time	Compile time	Runtime
Compiled flash size excluding data storage	9.1 KB	3.5 KB	5.4 KB	1.6 KB
Counter object support	Yes	Yes	Yes	No
Indexed object support	Partial (note 1)	Yes	Yes	No
Overhead per object (bytes)	4 (size <= 128 bytes); 8 (size > 128 bytes)	2	6	10
Counter object size including overhead (bytes)	212	60	56	NA
Counter increments before rewrite	100 (EFR32 Series 1); 50 (EFR32 Series 2)	25	25	NA

Note 1: When using NVM3, indexed objects are implemented by storing each index in a separate NVM3 object.

Flash Lifetime

All Silicon Labs Flash Data Storage implementations use some form of wear-levelling to prolong flash lifetime. The effectiveness of the wear-levelling depends on the implementation, the type of data stored, and how often it is updated. The main factors that affect wear-levelling and thereby flash lifetime are:

- Size of flash used for data storage: More flash area gives longer flash lifetime. For NVM3, the number of flash pages used for data storage can be configured, while the rest of the implementations use fixed storage sizes.
- Stored overhead per object: When writing data to the object storage, some overhead bytes are added to identify the data. Implementation with less overhead means the data objects take up less space in flash, and gives longer flash lifetime.
- Alignment to minimum object size: Objects are stored in multiples of the smallest object size. If the data size does not align with this size, padding bytes are added, which adds to the stored data and reduces flash lifetime. For instance, when storing

Copyright © 2025 Silicon Laboratories. All rights reserved.

16-bit objects, NVM3 and PS Store add two extra bytes of padding in addition to the overhead bytes. SimEEv1/v2 are able to store 16-bit data objects without padding.

- Remaining storage after basic storage: For implementations using virtual pages, when switching to a new virtual page one instance of each object is written to the page. The rest of the virtual page can then be used to store new writes of the objects. If a lot of space is used to store one instance of each object, little space is left in the virtual page to use for wear-levelling the subsequent object writes. The flash lifetime will therefore be reduced when the total amount of object data is large relative to the virtual page size. Even for NVM3, where virtual pages are not used, the flash lifetime is limited by the available space of the total NVM3 storage.

To help monitor the actual flash wear, NVM3 and SimEEv1/v2 include function calls for reporting the number of page erases of the data storage flash pages. These erase counters can be read during accelerated lifetime testing of a product to verify if the flash wears at an acceptable rate.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Using Third Generation Non-Volatile Memory (NVM3) Data Storage

Using Third Generation Non-Volatile Memory (NVM3) Data Storage

NOTE: This section replaces *AN1135: Using Third Generation Non-Volatile Memory (NVM3) Data Storage*. Further updates to this application note will be provided here.

The NVM3 driver provides a means to write and read data objects (key/value pairs) stored in flash. Wear-leveling is applied to reduce erase and write cycles and maximize flash lifetime. The driver is resilient to power loss and reset events, ensuring that objects retrieved from the driver are always in a valid state. A single NVM3 instance can be shared among several wireless stacks and application code, making it well-suited for multiprotocol applications. This application note explains how NVM3 can be used as non-volatile data storage in Zigbee (EmberZNet), Open Thread, Z-Wave, Bluetooth, Connect, and WiseConnect applications.

Version 3.x of the Gecko SDK Suite, used with Simplicity Studio 5, introduced a new component-based project architecture that replaced AppBuilder. Eventually all wireless protocols will move to the component-based architecture. This document addresses both this new approach to NVM3 configuration as well as the AppBuilder configuration still in use.

Key Points

- Key/value pair data object storage in flash
- Wear-leveling to maximize flash lifetime
- Resilient to power and reset events
- Shared by Zigbee, Connect, OpenThread, Z-Wave, and Bluetooth stacks
- Compatible with PS Store and Token APIs through wrappers
- Data upgradable from Simulated EEPROM version 2 to NVM3

Copyright © 2025 Silicon Laboratories. All rights reserved.

Overview

Overview

The third generation Non-Volatile Memory (NVM3) data storage driver is for storing persistent data in flash.

NVM3 is designed to work on all Silicon Labs wireless stacks running on EFR32, Series-3, and Si91x devices, as well as MCU applications running on EFM32.

Some of the main features of NVM3 are:

- Key/value pair data storage in flash
- Runtime object creation and deletion
- Persistence across power loss and reset events
- Wear-leveling to maximize flash lifetime
- Object sizes configurable up to 4096 bytes
- Configurable flash storage size (minimum 3 flash pages)
- Cache with configurable size for fast object access
- Data and counter object types
- Compatibility layers included for several Silicon Labs persistent storage APIs
- Single shared storage instance in multiprotocol applications
- Repack API to allow application to run clean-up page erases during periods with low CPU load

NVM3 is described in detail in the NVM3 Documentation on https://docs.silabs.com/. The rest of this document assumes you are familiar with that content.

While the NVM3 API can be used directly, NVM3 is also used as the underlying storage mechanism for several other persistent storage APIs provided by Silicon Labs:

- Token API used in Zigbee EmberZNet and Connect applications
- Persistent Storage API used in Silicon Labs Bluetooth applications

Copyright © 2025 Silicon Laboratories. All rights reserved.

NVM3 Default Instance

NVM3 Default Instance

Several NVM3 instances can be created on a device and live independently of each other, but to save memory it is usually desirable to use only one NVM3 instance as each instance adds some overhead. For applications based on Silicon Labs wireless stacks, a common default instance is used. This allows Dynamic Multiprotocol (DMP) applications that combine several wireless stacks to share the same NVM3 instance.

The number of flash pages used for the NVM3 default instance is configurable, but this setting must match if an application includes several stacks that all use the default instance.

> **IMPORTANT**: When creating an application that includes an NVM3 instance for a device that already contains an NVM3 instance in flash, the number of flash pages configured for the NVM3 instance must match the number of flash pages for the NVM3 instance already found on the device. Therefore it is not possible to change the size of an NVM3 instance once it has been installed on a device, without first erasing the flash pages holding the NVM3 instance and the NVM3 objects stored there.

NVM3 has a cache to speed up access to NVM3 objects. The cache size must be set to a value greater than or equal to the number of objects found in NVM3. This includes the number of NVM3 objects created through the native NVM3 API and any objects created through higher level APIs such as the Token API. The cache must also be large enough to hold any deleted NVM3 objects. The `nvm3_countObjects()` and `nvm3_countDeletedObjects()` functions can be used to find the number of live and deleted objects in NVM3 at any given point. Silicon Labs recommends checking these functions after initialization of all NVM3 objects, both through the native NVM3 API and higher level APIs such as the token API, to figure out the correct size of the NVM3 default cache size.

NVM3 Default Instance Key Space

NVM3 uses a 20-bit key to identify each object. To avoid using the same key for more than one object, the NVM3 key space for the default NVM3 instance has been divided into several domains as outlined in the following table. For example, NVM3 objects defined in the Zigbee EmberZNet stack should use NVM3 keys in the range 0x10000 to 0x1FFFF, while user application tokens should use keys below 0x10000. Note that any user defined NVM3 objects should be placed below 0x10000.

Domain	NVM3 Key
User	0x00000 - 0x0FFFF
Zigbee EmberZNet stack	0x10000 - 0x1FFFF
OpenThread stack	0x20000 - 0x2FFFF
Connect stack	0x30000 - 0x3FFFF
Bluetooth stack	0x40000 - 0x4FFFF
Z-Wave stack	0x50000 - 0x5FFFF
Bluetooth mesh stack	0x60000 - 0x6FFFF
Reserved	0x70000 - 0x7FFFF
Apple HomeKit	0x80000 - 0x80FFF
Zigbee Cluster Library (ZCL)	0x81000 - 0x81FFF
dotdot	0x82000 - 0x82FFF

Copyright © 2025 Silicon Laboratories. All rights reserved.

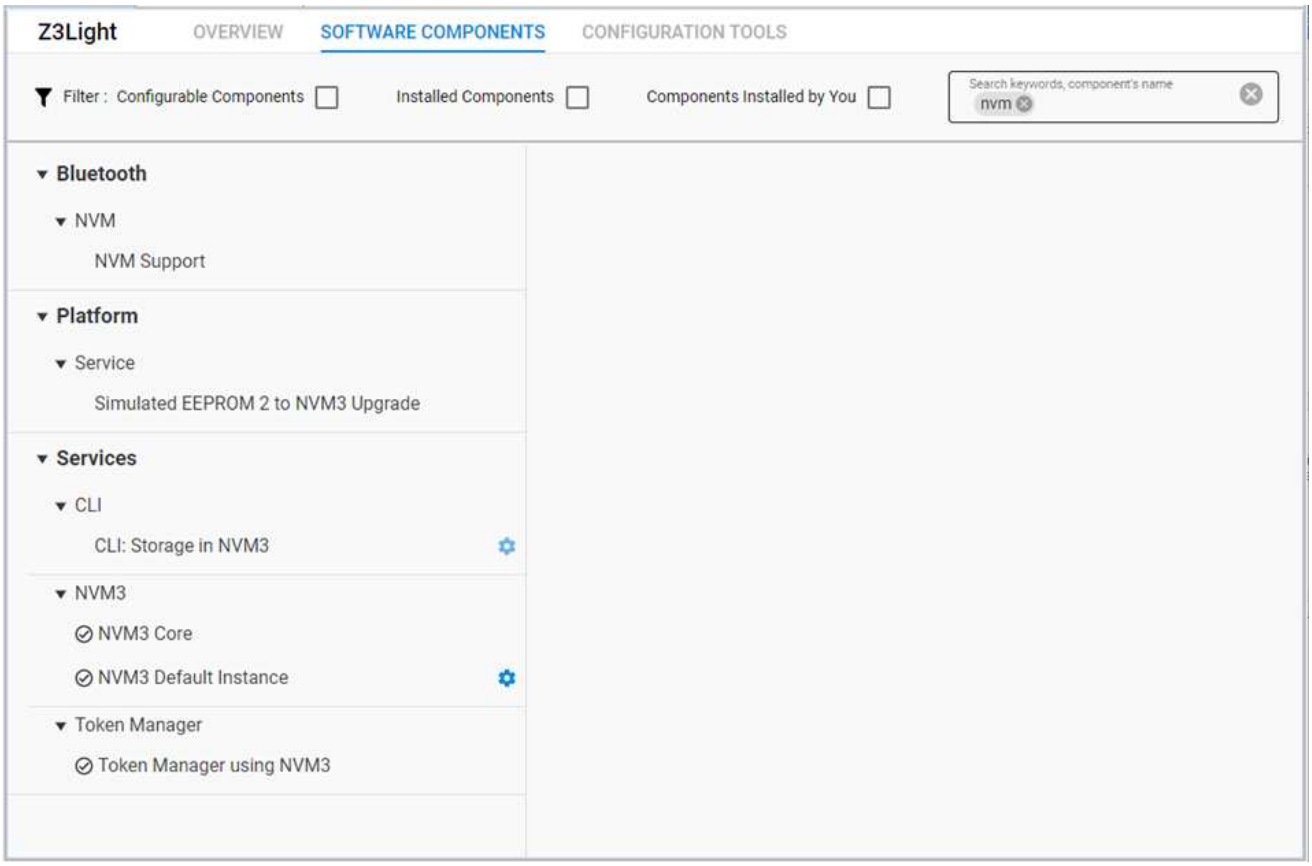
Domain	NVM3 Key
Platform CLI SM	0x83000 - 0x830FF
Platform Crypto	0x83100 - 0x870FF
Bootloader	0x87100 - 0x871FF
Matter	0x87200 - 0x87FFF
Reserved	0x88000 - 0x8DFFF
Static device tokens - override	0x8E000 - 0x8EFFF
Static secure tokens - override	0x8F000 - 0x8FFFF
Wi-SUN	0x90000 - 0x9FFFF
Reserved	0xA0000 - 0xFFFFF

Copyright © 2025 Silicon Laboratories. All rights reserved.

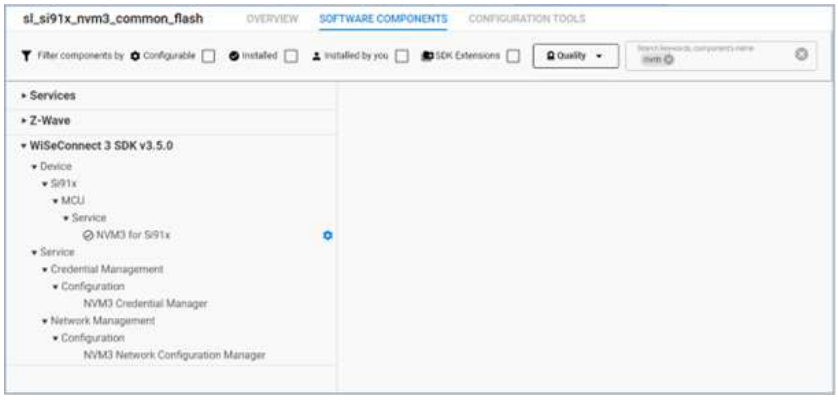
NVM3 in the Simplicity Studio 5 Project Configurator

NVM3 in the Simplicity Studio 5 Project Configurator

The Simplicity Studio 5 Project Configurator, used with Silicon Labs Bluetooth v3.x, Connect v3.x, EmberZNet Zigbee v7.x, and OpenThread SDK applications, includes an NVM3 Core component. A separate component is also provided for the NVM3 default instance, which will initialize this instance.



For Si91x devices, the NVM3 default instance can be configured under NVM3 for Si91x.



Copyright © 2025 Silicon Laboratories. All rights reserved.

The **NVM3 Default Instance** component provides the following configurations:

- **Cache Size**: Number of objects to cache. To reduce access times, this number should be equal to or higher than the number of live and deleted objects stored in NVM3 at any time.
- **Max Object Size**: Size of largest allowed NVM3 object in bytes. Must be between 208 and 4096 bytes.
- **User Repack Headroom**: Headroom determining how many bytes below the forced repack limit the user repack limit is placed. The default value is 0, which means that the forced and the user repack limits are the same.
- **Default Instance Size**: Size of the NVM3 storage region in flash. This must be set to match an integer number of flash pages, 3 pages at minimum.

If the **NVM3 Default Instance** component is included in a project, the default instance will be initialized automatically during `sl\_system\_init()` if the **System Init** component is included in the project.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Using NVM3 with Silicon Labs Connect

Using NVM3 with Silicon Labs Connect

This chapter applies to Connect in the Proprietary Flex SDK v3.x used with the Project Configurator in Simplicity Studio 5. Connect in Proprietary Flex SDK v2.x uses AppBuilder in Simplicity Studio 4. AppBuilder use is described in Using NVM3 in AppBuilder-Based Applications.

For Silicon Labs Connect v3.x applications, a **Token Manager** component is available under Platform > Driver in the Simplicity Studio Project Configurator Software Component view. While the token manager provides the Token API, an additional component must be selected for the token storage backend, either **Token Manager using Sim EEPROM 1**, **Token Manager using Sim EEPROM 2**, or **Token Manager using NVM3.** If NVM3 is chosen as the storage backend, **NVM3** and the **NVM3 Default Instance** components are included in the project.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Using NVM3 with Silicon Labs OpenThread Applications

Using NVM3 with Silicon Labs OpenThread Applications

This section applies to the Silicon Labs OpenThread SDK v1.x used with the Simplicity Project Configurator in Simplicity Studio 5. All OpenThread sample applications in the GSDK by default are configured to use NVM3 to store data in non-volatile memory. When doing so, these applications:

- Use the common default NVM3 instance.
- Include the NVM3 Core and the NVM3 Default Instance components in the project.
- Use the native NVM3 API to access the NVM3 object

The NVM3 key space used by the OpenThread stack is 0x20000 to 0x2FFFF.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Using NVM3 with Zigbee EmberZNet v.7x And Higher

Using NVM3 with Zigbee EmberZNet v7.x and Higher

This chapter applies to the Silicon Labs Zigbee EmberZNet SDK v7.x and higher used with the Simplicity Project Configurator in Simplicity Studio 5. Zigbee EmberZNet 6.x and lower use AppBuilder. AppBuilder use is described in Using NVM3 in AppBuilder-Based Applications.

All Zigbee sample applications in the GSDK by default are configured to use NVM3 to store data in non-volatile memory. When doing so, these applications:

- Use the common default NVM3 instance
- Include the NVM3 Core and the NVM3 Default Instance components in the project
- Use the native NVM3 API to access the NVM3 object

If you have non-volatile data stored in the older SimEE format and need to preserve the data when migrating your Zigbee application to EmberZNet v7.x or higher, then the Token Manager component is available to help manage the data storage. For more information, see the Token Manager component's description.

The NVM3 key space used by the Zigbee stack is 0x10000 to 0x1FFFF.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Using NVM3 with Silicon Labs Bluetooth Applications

Using NVM3 with Silicon Labs Bluetooth Applications

Traditionally the Bluetooth stack uses its own proprietary solution to store data in non-volatile memory, called **Persistent Store (PS Store)**. PS Store stores both data handled by the stack (such as temporary Bluetooth address, bonding keys, and so on) and user data (such as the device state) that has to be preserved on resetting the device. To learn more about PS Store, read the related section of the Bluetooth API Reference Guide.

Some of the sample applications in the Bluetooth SDK are still configured to use PS Store, while others are already configured to use NVM3. This means that:

- On Series 1 devices sample apps are configured to use PS Store, except Bluetooth Mesh NCP sample projects, where NVM3 is used by default.
- On Series 2 devices all sample apps are configured to use NVM3.

While Series 2 devices support NVM3 only, on Series 1 devices both PS Store and NVM3 can be used. This section describes how to configure NVM3 in the Bluetooth SDK, and how to switch between PS Store and NVM3 if needed.

Configuring NVM3 in the Bluetooth SDK

With Simplicity Studio 5 and Bluetooth SDK v3.x

NVM3 can be configured with the Project Configurator as described in NVM3 in the Simplicity Studio 5 Project Configurator.

With Simplicity Studio 4 and Bluetooth SDK v2.x

The Project Configurator is not available to configure NVM3 parameters. Therefore, the parameters have to be defined manually. To overwrite the default parameters:

1. Open the project settings.
2. Find the defined symbols:
    - If you use GCC as your compiler, go to C/C++ Build>Settings>GNU ARM C Compiler>Symbols>Defined Symbols
    - If you use IAR as your compiler, go to C/C++ Build>Settings>IAR C/C++ compiler for ARM>Preprocessor>Defined Symbols
3. Add any of the following defines to overwrite the default parameters:
    - NVM3_DEFAULT_NVM_SIZE
    - NVM3_DEFAULT_CACHE_SIZE
    - NVM3_DEFAULT_MAX_OBJECT_SIZE
    - NVM3_DEFAULT_REPACK_HEADROOM

You can find the description of each parameter in NVM3 in the Simplicity Studio 5 Project Configurator. Be careful when providing the size of NVM3, as it must be a multiple of the flash page size. Note: On Series 1 devices the flash page size is typically 2 kB, while on Series 2 devices the flash page size is typically 8 kB. Check the Wireless Gecko Reference Manual for your device. The NVM size must be at least 3 flash pages.

Switching from PS Store to NVM3

Beginning with Bluetooth SDK v2.13.0, both PS Store and NVM3 are supported as non-volatile memory solutions on Series 1 devices. Most sample applications are configured to use PS Store by default, but for some applications (where larger non-volatile memory is needed) NVM3 may be a better solution.

> **Note**: PS Store and NVM3 are not compatible with each other, therefore upgrading an already existing application from PS Store to NVM3 will result in losing all data stored on the device. If you have an application

Copyright © 2025 Silicon Laboratories. All rights reserved.

> running in the field, it may be wiser to stay with PS Store. If you still want to upgrade, see Using the Gecko Bootloader with the Silicon Labs Bluetooth Application for details.

> **Note**: PS Store uses only 2 flash pages (=4 kB on an EFR32BG1/12/13 device). Therefore, changing to NVM3 will affect the available space in flash. You must be particularly careful when you upgrade the firmware not to overwrite the NVM3 area with the application.

With Simplicity Studio 5 and Bluetooth SDK v3.x

To change your project configuration from PS Store to NVM3, simply install the **NVM3 Default Instance** component in the Project Configurator as discussed in NVM3 in the Simplicity Studio 5 Project Configurator. This will automatically uninstall the (otherwise hidden) PS Store component.

With Simplicity Studio 4 and Bluetooth SDK v2.x

To change your project configuration from PS Store to NVM3, use the following procedure.

1. Copy the following folder with all of its content:
   `C:\SiliconLabs\SimplicityStudio\v4\developer\sdks\gecko_sdk_suite\<version>\platform\emdrv\nvm3` under the `/platform/emdrv` folder of your project.
2. Remove the following files from the project:
   - `/platform/emdrv/nvm3/src/nvm3_hal_extflash.c`
   - `/platform/emdrv/nvm3/src/nvm3_default_extflash.c` (NVM3 use with external flash is deprecated)
3. If you use Apploader in your project, also copy the NVM3 version of Apploader from:
   `C:\SiliconLabs\SimplicityStudio\v4\developer\sdks\gecko_sdk_suite\<version>\protocol\bluetooth\lib\<device>\<compiler>\binapploader_nvm3.o`
   into the `/protocol/bluetooth/lib/<device>/<compiler>` folder of your project.
4. **If you use GCC as a compiler:**
   1. Go to `Project > Properties > C/C++ Build > Settings > GNU ARM C Compiler > Includes`.
   2. Add `${workspace_loc:/${ProjName}/platform/emdrv/nvm3/inc}` to the include directory.
   3. Go to `Project > Properties > C/C++ Build > Settings > GNU ARM C Linker > Miscellaneous`.
   4. Remove `${workspace_loc:/${ProjName}/protocol/bluetooth/lib/<device>/<compiler>/libpsstore.a}`.
   5. Add `${workspace_loc:/${ProjName}/platform/emdrv/nvm3/lib/libnvm3_CM4_gcc.a}`.

   If you use IAR as a compiler:
   1. Go to `Project > Properties > C/C++ Build > Settings > IAR C/C++ Compiler for ARM > Preprocessor`.
   2. Add `${workspace_loc:/${ProjName}/platform/emdrv/nvm3/inc}` to the include directory.
   3. Go to `Project > Properties > C/C++ Build > Settings > IAR Linker for ARM > Library`.
   4. Remove `${workspace_loc:/${ProjName}/protocol/bluetooth/lib/\<device\>/\<compiler\>/libpsstore.a}`.
   5. Add `${workspace_loc:/${ProjName}/platform/emdrv/nvm3/lib/libnvm3_CM4_iar.a}`.
5. If you use Apploader, also modify:
   `${workspace_loc:/${ProjName}/protocol/bluetooth/lib/\<device\>/\<compiler\>/binapploader.o}`
   to
   `${workspace_loc:/${ProjName}/protocol/bluetooth/lib/\<device\>/\<compiler\>/binapploader_nvm3.o}`
6. Configure NVM3 as described in Configuring NVM3 in the Bluetooth SDK.

Switching from NVM3 to PS Store

It may happen that, for a reason such as backward compatibility, you have to change the configuration from NVM3 to PS Store.

With Simplicity Studio 5 and Bluetooth SDK v3.x

Copyright © 2025 Silicon Laboratories. All rights reserved.

To change your project configuration from NVM3 to PS Store, simply uninstall the NVM3 Default Instance component. This will automatically install the (otherwise hidden) PS Store component. Note, that this can only be done on series 1 devices, as series 2 devices do not support PS Store. **Also, it is not possible with Bluetooth Mesh Stack in use**.

With Simplicity Studio 4 and Bluetooth SDK v2.x

To change your project configuration from NVM3 to PS Store, use the following procedure:

1. Remove the /platform/emdrv/nvm3 folder from your project.
2. Copy the PS Store library from:
   `C:\SiliconLabs\SimplicityStudio\v4\developer\sdks\gecko_sdk_suite\<version>\protocol\bluetooth\lib\<device>\<compiler>\libpsstore.a`
   into the `/protocol/bluetooth/lib/<device>/<compiler>` folder of your project.
3. If you use Apploader in your project, also copy the PS Store version of Apploader from:
   `C:\SiliconLabs\SimplicityStudio\v4\developer\sdks\gecko_sdk_suite\<version>\protocol\bluetooth\lib\<device>\<compiler>\binapploader.o`
   into the `/protocol/bluetooth/lib/<device>/<compiler>` folder of your project.

> **Note**: PS Store is not supported on Series 2 devices (EFR32xG2x), therefore there is only an NVM3 version of the Apploader for these devices.

4. **If you use GCC as your compiler**
   1. Go to `Project > Properties > C/C++ Build > Settings > GNU ARM C Compiler > Includes`.
   2. Remove `${workspace_loc:/${ProjName}/platform/emdrv/nvm3/inc}` from the include directory.
   3. Go to `Project > Properties > C/C++ Build > Settings > GNU ARM C Linker > Miscellaneous`.
   4. Add `${workspace_loc:/${ProjName}/protocol/bluetooth/lib/<device>/<compiler>/libpsstore.a}`.
   5. Remove `${workspace_loc:/${ProjName}/platform/emdrv/nvm3/lib/libnvm3_CM4_gcc.a}`.

   If you use IAR as your compiler
   1. Go to `Project > Properties > C/C++ Build > Settings > IAR C/C++ Compiler for ARM > Preprocessor`.
   2. Remove `${workspace_loc:/${ProjName}/platform/emdrv/nvm3/inc}` from the include directories.
   3. Go to `Project > Properties > C/C++ Build > Settings > IAR Linker for ARM > Library`.
   4. Add `${workspace_loc:/${ProjName}/protocol/bluetooth/lib/<device>/<compiler>/libpsstore.a}`.
   5. Remove `${workspace_loc:/${ProjName}/platform/emdrv/nvm3/lib/libnvm3_CM4_iar.a}`.
5. If you use Apploader, also change:
   `${workspace_loc:/${ProjName}/protocol/bluetooth/lib/<device>/<compiler>/binapploader_nvm3.o}`
   to
   `${workspace_loc:/${ProjName}/protocol/bluetooth/lib/<device>/<compiler>/binapploader.o}`.

Copyright © 2025 Silicon Laboratories. All rights reserved.

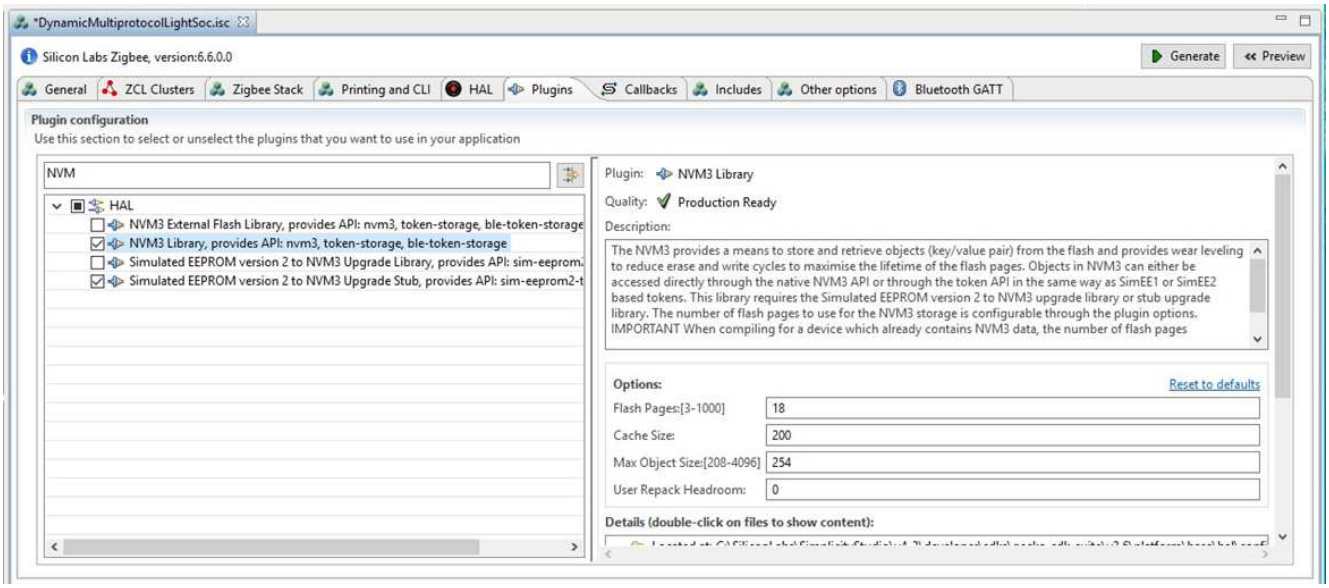
Using NVM3 In Appbuilder Based Applications

Using NVM3 in AppBuilder-Based Applications

This section explains how NVM3 can be used for non-volatile storage in AppBuilder-based applications like EmberZNet 6.10.x and lower, EmberZNet-based DMP applications in GSDK 3.2 and lower, and Connect applications in Gecko SDK Suite 2.7 and lower.

NVM Library Plugin

To use NVM3 with an AppBuilder-based example application, the NVM3 Library plugin should be included in the project. All PS Store and SimEE plugins should be deselected.



The **NVM3 Library** plugin offers four plugin options:

- **Flash Pages**: Number of flash pages to use for NVM3 data storage. Must be 3 or higher. Default is 18 (36KB) for EFR32 Series-1 devices and 4 (32KB) for EFR32 Series-2 devices
- **Cache Size**: Number of objects to cache. To reduce access times, this number should be equal to or higher than the number of objects, including tokens and deleted objects, stored in NVM3 at any time.
- **Max Object Size**: Size of largest allowed NVM3 object in bytes. Must be between 208 and 4096 bytes. Note that the token API can only be used to access objects of 254 bytes or smaller. When accessing larger objects, the native NVM3 API must be used.
- **User Repack Headroom**: Headroom determining how many bytes below the forced repack limit the user repack limit is placed. The default value is 0, which means that the forced and the user repack limits are the same.

When the NVM3 Library plugin is used the **Simulated EEPROM version 2 to NVM3 Upgrade Library** or **Simulated EEPROM version 2 to NVM3 Upgrade Stub Library** must be included, as described in SimEEv2 to NVM3 Upgrade Plugin.

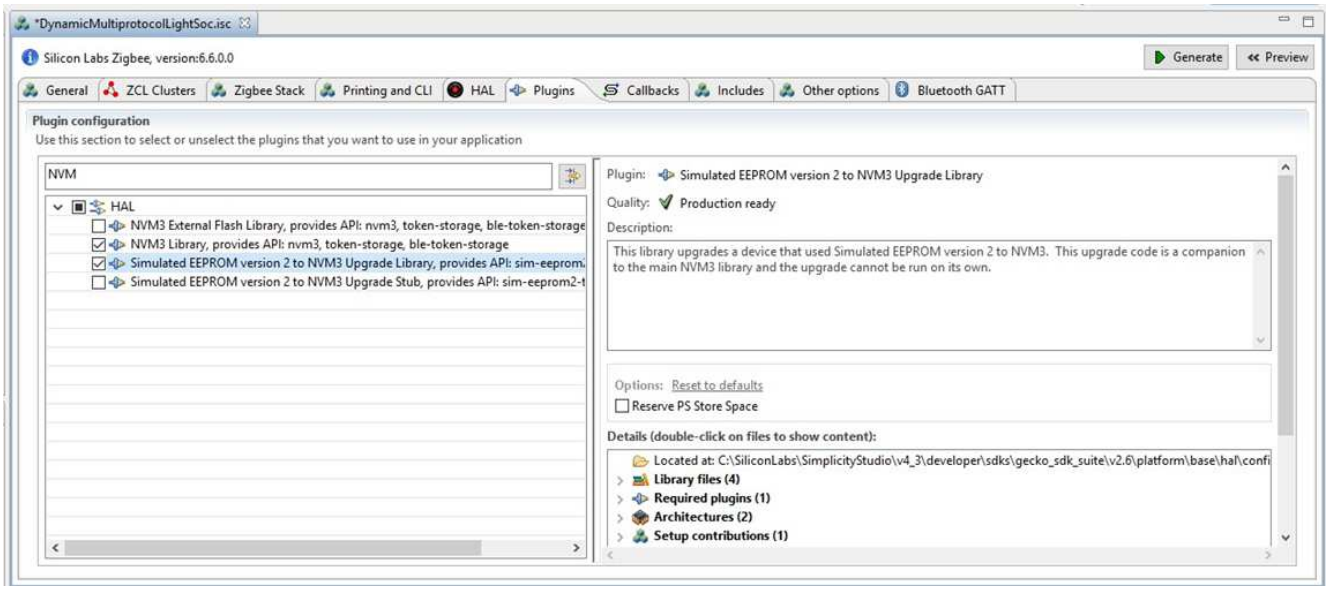
SimEEv2 to NVM3 Upgrade Plugin

An AppBuilder plugin (**Simulated EEPROM version 2 to NVM3 Upgrade Library**) is provided for EmberZNet applications that upgrade tokens stored in SimEEv2 to NVM3. For tokens to be successfully upgraded to NVM3, CREATOR_* and NVM3KEY_* defines must be added for all tokens as described in Token API. The upgrade plugin will replace the SimEEv2 storage in-place with an NVM3 storage instance. The plugin does this by compacting the SimEEv2 storage down to 12 kB,

Copyright © 2025 Silicon Laboratories. All rights reserved.

and then creates an NVM3 instance in the remaining 24 kB of the original 36 kB SimEEv2 storage space. After the token data has been copied over from SimEEv2 to NVM3, the SimEEv2 storage is erased and the NVM3 instance is resized to use the entire 36 kB storage space. Apart from the code space needed for the upgrade library code, the upgrade does not require any additional flash space to the 36 kB storage area. The upgrade plugin requires that the existing SimEEv2 storage space and new NVM3 storage space are located at the same address and have the same size.

The **Simulated EEPROM version 2 to NVM3 Upgrade Library** plugin should be included to enable the upgrade as shown in the figure below. If no SimEEv2 token data is found, the upgrade plugin will look for NVM3 data, and if neither is found it will create a new NVM3 instance with tokens set to their default values. For applications that do not need to upgrade any SimEEv2 tokens, the **Simulated EEPROM version 2 to NVM3 Upgrade Stub** plugin should be included instead.



Copyright © 2025 Silicon Laboratories. All rights reserved.

Using NVM3 with Z Wave Applications

Using NVM3 with Z-Wave Applications

For details on using NVM3 with Z-Wave applications, see https://www.silabs.com/documents/public/user-guides/INS14259.pdf, section 7.4.

Copyright © 2025 Silicon Laboratories. All rights reserved.

NVM3 API Options

NVM3 API Options

This chapter describes the three different APIs available to access NVM3 objects.

- Token API
- Persistent Store API
- Native NVM3 API

Token API

The token API is used to access data stored in SimEEv1 and SimEEv2 with the EmberZNet and Connect stacks, as well as multiprotocol applications. Information on how to define and access tokens can be found in *AN1154: Using Tokens for Non-Volatile Data Storage*, and users should read this document before using the token API. When selecting the NVM3 Library plugin instead of one the SimEE plugins, the NVM3 default instance is used to store the token data instead of SimEE. The same token API can still be used to access tokens stored in NVM3, but the token definition needs some modifications to work with NVM3, as described below.

When defining a token to be used with SimEE, a creator code must be defined as an identifier for the token. Similarly, when defining a token to be used with NVM3, an NVM3 key must be defined for the token. A token definition that is compatible with both NVM3 and SimEE would include both an NVM3 key and a creator code and look like this:

```
#define CREATOR_name 16bit_value
#define NVM3KEY_name 20bit_value
#ifdef DEFINETYPES
  typedef data_type type #endif
#ifdef DEFINETOKENS
DEFINE_*_TOKEN(name, type, ... ,defaults) #endif
```

Select a 20-bit NVM3 key for the token, according to the domains in NVM Default Instance Key Space. Each token must have a unique NVM3 key, except for indexed tokens, where more NVM3 keys must be reserved as outlined in Special Considerations for Indexed Tokens.

Deleting Tokens

As tokens are created at compile time, they cannot be created or deleted at run time. NVM3 objects are, however, created and deleted at run time, and the token initialization function creates NVM3 objects for each defined token if they do not already exist. The token initialization generally does not delete NVM3 objects found that do not have a corresponding token associated with them. Therefore, if a token is no longer included in an application, the application should manually delete the associated NVM3 object by using the NVM3 Native API described in Native NVM3 API. For indexed tokens, however, the token initialization checks if indexed tokens have more or less indexes than the number of NVM3 objects found in the indexed token's NVK3KEY range. If there are fewer indexes, the token initialization deletes the extra NVM3 objects. If the number of indexes has been increased, new NVM3 objects will be created to hold these indexes.

When NVM3 objects are deleted, the actual object data remains in NVM3 but is marked as deleted. The deleted object data remains in NVM3 and consumes cache space until NVM3 repacks have erased the page(s) holding all versions of these objects.

Special Considerations for Indexed Tokens

NVM3 does not have native support for indexed tokens. Therefore an extra requirement is imposed on the NVM3 key selection for indexed tokens. With NVM3, indexed tokens are implemented by storing each index in a separate object, starting with index 0 stored at the defined NVM3KEY_name key value and the last index (127) stored with key NVM3KEY_name + 127. Because of this implementation, 128 NVM3 keys must be reserved for each indexed token. The user still only defines one NVM3KEY_name key value, but no other tokens should be defined with key values in the 127

Copyright © 2025 Silicon Laboratories. All rights reserved.

values following this defined key. Even if the token is defined with fewer than 128 indices, all 128 indices should be reserved as the token might be expanded with more indices later on.

The example below shows two indexed tokens defined in the user key domain:

```
// This key is used for an indexed token and the subsequent 0×7F keys are also reserved #define
NVM3KEY_MY_INDEXED_TOKEN_A 0×00000
// This key is used for an indexed token and the subsequent 0×7F keys are also reserved #define
NVM3KEY_MY_INDEXED_TOKEN_B 0×00080
```

The table below provides indexed token NVM3 key selections in the example.

NVM3KEY	NVM3 Objects Contents
0x00000	Reserved for TOKEN_MY_INDEXED_TOKEN_A index 0
0x00001	Reserved for TOKEN_MY_INDEXED_TOKEN_A index 1
0x00002	Reserved for TOKEN_MY_INDEXED_TOKEN_A index 2
...	
0x0007F	Reserved for TOKEN_MY_INDEXED_TOKEN_A index 127
0x00080	Reserved for TOKEN_MY_INDEXED_TOKEN_B index 0
0x00081	Reserved for TOKEN_MY_INDEXED_TOKEN_B index 1
...	
0x000FF	Reserved for TOKEN_MY_INDEXED_TOKEN_B index 127

Bluetooth NVM API

The Bluetooth NVM API that was originally designed for PS Store can be used in the same way when using NVM3 as when using PS Store. The Bluetooth stack will automatically translate its NVM API calls to PS Store API calls or to NVM3 API calls in the background, depending on what components the project uses. The same applies for Zigbee + Bluetooth DMP projects, where NVM3 is applied as the storage mechanism, but the Bluetooth NVM API can still be used. The Bluetooth API is documented in the Bluetooth API Reference Manual. In Bluetooth SDK v2.x it can be found under the *Flash* class (commands starting with `gecko_cmd_flash_`), while in Bluetooth SDK v3.x it can be found under the *NVM* class (commands starting with `sl_bt_nvm`).

16-bit keys are used with the Bluetooth NVM API, which are then mapped to a 20-bit NVM3 key when NVM3 is used as storage mechanism. The four most significant bits are set to 0x4 to place these objects in the Bluetooth domain of the NVM3 default instance key space. As the Bluetooth NVM API is fixed to use only the Bluetooth domain, any objects to be placed in other domains, for example the User domain, must be created and accessed using the native NVM3 API.

If you want to use the Bluetooth NVM API and the native NVM3 API in the same app, then:

1. Call `gecko_init(pconfig)` to initialize the Bluetooth stack and open its own NVM3 instance. This is done in all Bluetooth sample apps.
2. Open NVM3 by calling the `nvm3_open()` function with the default (!) parameters to open your NVM3 instance: `nvm3_open(nvm3_defaultHandle, nvm3_defaultInit);`

User data can now be saved to:

- PS key range 0x4000 - 0x407F. All other PS keys (0x0000-0xFFFF) are reserved for the stack (for example for storing bonding data).
- NVM3 key range 0x00000-0x0FFFF (NVM3 user data area), and 0x44000-0x4407F (PS Store user data area).

For example, the following API calls will have the same effect (writing to the same area):

- `nvm3_writeData(nvm3_defaultHandle,0x44000,(void\*)data,len);`
- `gecko_cmd_flash_ps_save(0x4000,len,data); //in Bluetooth SDK v2.x`
- `sl_bt_nvm_save(0x4000,len,data); //in Bluetooth SDK v3.x`

Similarly, you can read the same data with the following API calls:

Copyright © 2025 Silicon Laboratories. All rights reserved.

- `nvm3_readData(nvm3_defaultHandle,0x44000,(void\*)read_buffer,maxlen);`
- `gecko_cmd_flash_ps_load(0x4000); //in Bluetooth SDK v2.x`
- `sl_bt_nvm_load(0x4000,maxlen,&read_len,(uint8_t\*)read_buffer); //in Bluetooth SDK v3.x`

Native NVM3 API

For code accessing NVM3 objects that does not need to be compatible with the token or PS Store APIs, using the native NVM3 API to access NVM3 data is recommended to reduce code size and allow using the full feature set of NVM3. Any PS Store object or token can also be accessed through the native NVM3 API using the same NVM3 key. Complete documentation of this API is found online in the Drivers section of the Gecko Platform Developer Documentation. If you are using an earlier version of the GSDK, search that page's version history for the corresponding release.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Simplicity Commander and NVM3

Simplicity Commander and NVM3

Simplicity Commander is a single, all-purpose tool to be used in a production environment. It is invoked using a simple CLI (Command Line Interface) that is also scriptable. Simplicity Commander supports reading out the NVM3 data area from a device and parsing the NVM3 data to extract stored values. This can be useful in a debugging scenario where you may need to find out the stored state of an application that has been running for some time.

More information on how to use the Simplicity Commander with NVM3 can be found in Simplicity Commander Reference Guide.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Securing NVM3 Data on External Flash Devices

Securing NVM3 Data on External Flash Devices

Series-3 devices do not have on-die NVM. Since off-chip memory such as external flash is vulnerable to persistent attacks, data objects are securely stored and retrieved from the external flash using AES-GCM authenticated encryption. On Series-3 devices, the NVM3 AES-128 symmetric key is used to encrypt the data, ensuring it cannot be read in plain text from the external flash. The NVM3 AES-128 symmetric key is a device-generated, non-exportable key stored in the Secure Engine MTP. The NVM3 symmetric key is invalidated or erased during a device erase. Any attempt to load previously recorded NVM3 data (encrypted with the older key) into flash will result in invalid reads and must be avoided. In the case of a device erase, it is essential to reinitialize the NVM3 instance and add the objects again. For details about DFA and DPA countermeasures and the AES engines being used, refer to Secure Key Storage.

Impact of External Flash and Security on Repack Timings

For Series-3 devices using external flash, repack operations may take longer due to the increased latency of accessing external flash over the bus. Additionally, with security features such as authenticated encryption (AES-GCM) enabled, the repack timings may be further impacted due to the cryptographic overhead. It is recommended to schedule repack operations during periods of low CPU and bus activity to minimize the impact on application performance.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Best Practices and Optimizations For NVM3

Best Practices and Optimizations for NVM3

Right Sizing the NVM3 Storage Region in Flash

To minimize repacks and reduce flash wear, it is important to configure the NVM3 region size appropriately. The size of the NVM3 region should be based on:

- The number of objects stored in NVM3.
- The size of each object, including metadata overhead.
- The frequency of object updates and deletions.
- The flash page size of the device.

With the addition of security on Series 3 devices, storing a data object incurs a size overhead of 8 bytes. Ensure that the NVM3 region has sufficient space to accommodate active objects, deleted objects, and spare pages for repack operations. A larger NVM3 region reduces the frequency of repacks, thereby extending the flash lifetime. The "Maximum allowed basic storage" section at https://docs.silabs.com/gecko-platform/latest/platform-driver/nvm3#max-allowed-basic-storage-with-4-k-b-page-size can help with configuring the appropriate number of flash pages and right-sizing the NVM3 storage in flash.

Right Sizing the Cache

The NVM3 cache size should be configured to match the number of live and deleted objects in the NVM3 region. This ensures faster access to objects and reduces the overhead of searching for objects in flash. Users can query the cache size and memory information using the `nvm3_getMemInfo()` API. This API provides details about the current memory state, including available memory, cache status, and additional cache requirements. Properly sizing the cache improves performance and reduces object lookup latency.

Low Memory Notification Callback

Users can use the `nvm3_registerCallback()` API to register a callback function, which is triggered when the NVM3 instance detects low memory conditions or a cache overflow. The callback allows the application to handle conditions such as freeing up memory by triggering repack. It is recommended that the callback be registered before calling any NVM3 APIs. Additionally, users can deregister the callback using the `nvm3_deregisterCallback()` API.

NVM3 Optimizations

Enabling NVM3 optimization improves the NVM3 initialization and object lookup time. NVM3 optimization can be enabled or disabled from Simplicity Studio UC for Series-2 and Series-3 devices. The optimization support has implications on code size. With optimization support enabled, the code size increases by approximately 1248 bytes. NVM3 optimization is enabled by default for Series-3 devices and disabled for Series-2 devices.

By following these best practices and leveraging the available APIs and optimizations, users can maximize the efficiency and reliability of the NVM3 storage system while maintaining high performance for their applications.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Bringing Up Custom Devices for the EFR32MG and EFR32FG Families

Bringing Up Custom Devices for the EFR32MG and EFR32FG Families

NOTE: This section replaces *AN961: Bringing Up Custom Devices for the EFR32MG and EFR32FG Families*. Further updates to this application note will be provided here.

This application note describes how to initialize a piece of custom hardware (a "device") based on the EFR32MG and EFR32FG families so that it interfaces correctly with a network stack. The same procedures can be used to restore devices whose settings have been corrupted or erased. Note, this document is based on Zigbee EmberZNet v7. Following the release of Zigbee EmberZNet v8, included in Simplicity SDK, several API and type definitions have changed. For more information, please visit https://docs.silabs.com/zigbee/8.0.0/zigbee-api-ref-v7-vs-v8/02-renaming-changes-in-zigbee.

Key Points

- Information required before board bring-up
- Setting manufacturing tokens
- Bootloading
- Performing functional testing

Copyright © 2025 Silicon Laboratories. All rights reserved.

Introduction

Introduction

Before an EFR32-based product such as a member of the Wireless Gecko or Flex Gecko families (hereafter EFR32) can be initialized and tested, manufacturing tokens within the EFR32 User Data Page and the Lock Bits Page must be configured. Similarly, before any application specific code can be programmed into the EFR32 flash, a board header needs to be created either manually or from the Application Builder tool set, in EmberZNet 6.10.x and lower, or the Software Components tab in the Project Configurator, in EmberZNet 7.0 and higher. In order to perform these tasks, the product design team must know how the EFR32 is to be used in the wireless system.

In particular, the design team must know the following:

- PCB Manufacturing-specific information (serial number, product numbers, EUI, and so on)
- Bootloader architecture (serial dataflash)
- External components in the RF Path (PA, LNA, and so on)
- 38.4 MHz crystal oscillator specification and a CTUNE value to match your crystal so you hit the center frequency
- Application security tokens (Keys, certificates, and so on)

> **Note**: Even though the EFR32 flash is fully tested during production test, the flash contents in the main flash block are not set to a known state before shipment. The User Data and Lock Bits pages are erased to all 0xFF, except in kits where they might be preloaded with configuration values such as CTUNE.

Copyright © 2025 Silicon Laboratories. All rights reserved.

EFR32 Wireless System

EFR32 Wireless System

Once the hardware design of a custom device has been completed, the assembled product is ready for test and functional validation. Before testing, developers must understand how the device will operate at both the device-level and system-level. The following table describes the different items that developers should know before board bring-up.

Information	Required or Optional	Description
Device Level		
Product Information	Optional	Most products have unique serial numbers as well as generic product codes that might be stored in the EFR32. This information might include items such as where and when the device was assembled, a product serial number, and product name.
Custom EUI	Optional	IEEE 64-bit unique number. Each EFR32 comes with an EUI programmed into the Device Information Page. Customers that have their own IEEE address block can use it in place of the Device Information Page's EUI.
RF Component Information	Required	If the product uses external PAs or LNAs, then developers must know the pin-connections between the off-chip components and the EFR32. They should also understand the LNA Gain as it will be used to offset the clear channel assessment (CCA) threshold.
Protocol-Specific Information	Optional	If developing a product using the Zigbee networking protocol, obtain a Zigbee-assigned manufacturer code before testing the application.
System Level		
Bootloader Option	Required	Silicon Labs offers several bootloading options for different system designs. For more information, see Bootloader Fundamentals.
System Security	Optional	If device-level security is required for the product, determine necessary security settings and keys before setting up the device.

As detailed in the above table, Silicon Labs offers its customers an opportunity to guarantee a device's uniqueness on the network. In addition, it allows customers a way to store product descriptions, manufacturer-specific information and device information.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Setting Manufacturing Tokens

Setting Manufacturing Tokens

EFR32 chips are delivered to customers with all memory erased. Exceptions to this rule are chips that come with kits. Before the EFR32 chips can be used to run applications for a networking stack, the customer or a contract manufacturer/test house must prepare them. Preparation includes programming the proper application and bootloader, if necessary, into the Main flash block, as well as programming customer manufacturing tokens in the User Data block and Lock Bits block.

Manufacturing tokens are values programmed into special, non-volatile storage area of flash. The User Data page and Lock Bits page contain data that manufacturers of EFR32-based devices can program. The Device Information Page also contains manufacturing tokens, but these tokens are fixed values that cannot be modified.

> **Note**: Applications and the stack can read any manufacturing tokens at any time.

Simplicity Commander is a single, all-purpose tool to be used in a production environment. It is invoked using a simple Command-Line Interface (CLI) that is also scriptable. Simplicity Commander enables customers to complete these essential tasks:

- Flash their own applications.
- Configure their own applications.
- Create binaries for production.

For more information, refer to *UG162: Simplicity Commander Reference Guide*.

The following two tables identify the User Data manufacturing tokens that OEMs and CMs may want to program at manufacturing time. Refer to `platform\base\hal\micro\cortexm3\efm32\token-manufacturing.h` for the token definition, because it may differ from these tables depending on the stack release version.

> **Note**: Using tokens to set up manufacturing data in the Silicon Labs Flex SDK only works if you use the Silicon Labs Connect stack. If you're interfacing the RAIL library directly, you can set up similar values using the board-specific header files.

Silicon Labs recommends that User Data and Lock Bits page tokens be written using Simplicity Commander at the same time as programming the main flash. Simplicity Commander also allows for patching and reprogramming the manufacturing blocks as many times as necessary. (See Simplicity Commander Reference Guide for details.) The following two tables describe the location of each manufacturing token as an offset to the starting address of the relevant block. For the most accurate and specific information about where the flash regions begin in the address map of your chip, please consult your IC's Reference Manual or Data Sheet.

Some situations may require that a manufacturing token be programmed at runtime from code running on the chip itself. For wireless mesh network SDKs, the manufacturing token module of the HAL provides a token API to write the manufacturing tokens. However, this API only writes manufacturing tokens that are in a completely erased state. If a manufacturing token must be reprogrammed, you must use an external utility.

- The API on SoC platforms is `halCommonSetMfgToken(token, data)`. The parameters for this API are the same as the API `halCommonSetToken(token, data )`.

Copyright © 2025 Silicon Laboratories. All rights reserved.

- For EmberZNet PRO EZSP NCP platforms, the host API is `ezspSetMfgToken(tokenId, tokenDataLength, tokenData)`. (See *UG100: EZSP Reference Guide* for details.)

In the next table, the Connect stack only uses the following tokens (rows in bold):

- TOKEN_MFG_CUSTOM_EUI_64
- TOKEN_MFG_CTUNE

Table: User Data Manufacturing Tokens for the EFR32

Offset from User Data starting address				
Series 1	Series 2	Size (Bytes)	Name	Description
0x0000	0x000C	2	TOKEN_MFG_CUSTOM_VERSION	Version number to signify which revision of User Data manufacturing tokens you are using. This value should match CURRENT_MFG_CUSTOM_VERSION which is currently set to `0x01FE`. CURRENT_MFG_CUSTOM_VERSION is defined in `\hal\micro\cortexm3\efm32\token-manufacturing.h`. *Usage*: Recommended for all devices using User Data manufacturing tokens.
0x0002	**0x0002**	**8**	**TOKEN_MFG_CUSTOM_EUI_64**	**IEEE 64-bit address, unique for each radio. Entered and stored in little-endian. Setting a value here overrides the EUI64 stored in the Device Information Page. This is for customers who have purchased their own address block from IEEE. WARNING: If this value is set "live," it may not propagate to all levels of the stack and some outgoing messages may continue to use the default value. Silicon Labs recommends forcing a reset after setting EUI64 and before starting the networking stack. *Usage:* Optionally set by device manufacturer if using custom EUI64 address block.**
0x0080	0x008C	16	TOKEN_MFG_SERIAL_NUMBER	Serial Number value of the entire product in binary representation. *Usage*: Optionally set by device manufacturer.
0x001A	0x0010	16	TOKEN_MFG_STRING	Optional device-specific string, for example, the serial number. *Usage:* Optionally set by device manufacturer to identify device.
0x002A	0x0020	16	TOKEN_MFG_BOARD_NAME	Optional string identifying the board name or hardware model. *Usage:* Optionally set by device manufacturer to identify device.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Offset from User Data starting address				
Series 1	Series 2	Size (Bytes)	Name	Description
0x003A	0x0030	2	TOKEN_MFG_MANUF_ID	16-bit ID denoting the manufacturer of this device. When you are programming with EmberZNet PRO, Silicon Labs recommends setting this value to match your Zigbee-assigned manufacturer code, such as in the stack's emberSetManufacturerCode()``API call. *Usage:* Recommended for EmberZNet PRO devices utilizing the stand-alone bootloader.
0x003C	0x0034	2	TOKEN_MFG_PHY_CONFIG	Reserved for future use; should be left un-programmed (0xFFFF).
0x0104	0x009C	1	TOKEN_MFG_LFXO_TUNE	Manufacturing token space for Low Frequency XTAL tune value.
0x0108	0x0104	4	TOKEN_MFG_KIT_SIGNATURE	Manufacturing token space for Kit Signature. *Usage*: Optionally set by device manufacturer for kit identification.
0x003E	0x0038	40	TOKEN_MFG_ASH_CONFIG	ASH configuration information. *Usage:* Optional for EmberZNet PRO devices acting as network coprocessors (NCPs) for EZSP-UART. Not used by any SoC use cases.
0x00F0	0x0060	2	TOKEN_MFG_SYNTH_FREQ_OFFSET	Reserved for future use; should be left un-programmed (0xFFFF). Radio synthesizer frequency adjustments should be made using the `TOKEN_MFG_CTUNE` token instead.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Offset from User Data starting address				
Series 1	Series 2	Size (Bytes)	Name	Description
0x00F6	0x0064	2	TOKEN_MFG_CCA_THRESHOLD	Threshold(s) used for energy detection Clear Channel Assessment (CCA). You may want to override the default CCA threshold(s) by setting this token if your design uses a Low-Noise Amplifier (LNA). An LNA changes the gain on the radio input which results in the radio "seeing" a different energy level than if no LNA was used. Bits 0-7: Set to the two's complement representation of the 2.4GHz band CCA threshold in dBm below which a 2.4GHz channel will be considered clear. Valid values are -128 through +126, inclusive. +127 is **not** valid and **must not** be used. Bit 8: Set to 0 if the threshold in Bits 0-7 is valid and should be used. Set to 1 (the erased state) if those bits are invalid or have not been set; in this case, the default 2.4GHz band threshold of -75 dBm will be used. Bits 9-15: Applies only to SubGHz-capable PHYs. These bits can be used to override the default CCA threshold that is applied to channels in the SubGHz band (for example, in GB868, the default SubGHz CCA threshold is -87 dBm). The interpretation of the SubGHz CCA threshold value in bits 9..15 is as a 7-bit unsigned value in the range 1..126 which is negated to be the CCA threshold to apply (that is, token values 1..126 in bits 15..9 map to SubGHz CCA thresholds of -1..-126 dBm respectively). The values 0 and 127 (the 7 bits all 0s or all 1s) are interpreted as no override—the default SubGHz CCA threshold will be used.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Offset from User Data starting address				
Series 1	Series 2	Size (Bytes)	Name	Description
0x00F6 (Cont.)	0x0064 (Cont.)		TOKEN_MFG_CCA_THRESHOLD (Cont.)	**Example 1:** A 2.4GHz design uses an LNA that provides a gain of +12 dB. Add that gain to the 2.4GHz default threshold of -75 dBm to get the dBm value for the token: -75 + 12 = -63 dBm. The two's complement signed representation of -63 is 0xC1 so the complete token value to be programmed is 0xFEC1. **Example 2:** A GB868 SubGHz design uses an LNA that provides a gain of +12 dB. Add that gain to the GB868 SubGHz default threshold of -87 dBm to get the negated dBm value for the token: -87 + 12 = -75 dBm. Negating, the value 75 or 0x4B goes into bits 9..15 of the token so the complete token value to be programmed is 0x97FF. **Example 3:** A dual-PHY design has an LNA like Example 1 for 2.4GHz and an LNA as in Example 2 for GB868 SubGHz. The token values in those examples combine to be 0x96C1 for this design.
0x00F8	0x0068	8	TOKEN_MFG_EZSP_STORAGE	An 8-byte, general-purpose token that can be set at manufacturing time and read by the host microcontroller via EZSP's getMfgToken command frame. *Usage:* Not required. Device manufacturer may populate or leave empty as desired.
0x0100	**0x0100**	**2**	**TOKEN_MFG_CTUNE**	**This token is for tuning the EFR32 system XTAL and consequently also tunes the radio synthesizer frequency. *Usage:* Optional. Only necessary if additional crystal tuning is desired to reach optimal timing and frequency output.**
0x0102	0x0070	2	TOKEN_MFG_XO_TUNE	This token is for tuning an attached Si446x transceiver's system crystal and consequently also tunes its radio synthesizer frequency. A value in the least-significant byte in the range 0x00 to 0x7F will be applied to the Si446x's GLOBAL_XO_TUNE property. All other values are currently ignored and reserved for future use.

In the next table, the Connect stack only uses the following tokens for the Gecko Bootloader (rows in bold):

- TOKEN_MFG_SECURE_BOOTLOADER KEY
- TOKEN_MFG_SIGNED_BOOTLOADER KEY_X
- TOKEN_MFG_SECURE_BOOTLOADER KEY_Y

Table: Lock Bits Manufacturing Tokens for the EFR32

Copyright © 2025 Silicon Laboratories. All rights reserved.

Offset from Lock Bits starting address				
(Series 1)	(Series 2)	Size (Bytes)	Name	Description
0x0204	0x032C	16	TOKEN_MFG_BOOTLOAD_AES_KEY	Reserved for future use; should be left un-programmed (all 0xFF bytes).
0x0214	0x0204	92	TOKEN_MFG_CBKE_DATA	Defines the security data necessary for Smart Energy devices. It is used for Certificate Based Key Exchange to authenticate a device on a Smart Energy network. The first 48 bytes are the device's implicit certificate, the next 22 bytes are the Root Certificate Authority's Public Key, the next 21 bytes are the device's private key (the other half of the public/private key pair stored in the certificate), and the last byte is a flags field. The flags field should be set to 0x00 to indicate that the security data is initialized. For more information on the Smart Energy tokens, see document *AN708: Setting Smart Energy Certificates for Zigbee Device**s. Usage:* Required by Smart Energy Profile certified devices.
0x0270	0x0270	20	TOKEN_MFG_INSTALLATION_CODE	Defines the installation code for Zigbee devices. The installation code is used to create a pre-configured link key for initially joining a Zigbee 3.0 or Zigbee Smart Energy (ZSE) network. The first 2 bytes are a flags field, the next 16 bytes are the installation code, and the last 2 bytes are a CRC. Valid installation code sizes for ZSE devices are 6, 8, 12, or 16 bytes in length, but Zigbee 3.0 devices are required to use 16-byte installation codes. All unused bytes should be 0xFF. The flags field should be set as follows depending on the size of the install code: 6 bytes = 0x0000 8 bytes = 0x0002 12 bytes = 0x0004 16 bytes = 0x0006 For more information on working with Zigbee installation codes, see document Using Installation Codes with Zigbee Devices. For more information on the Smart Energy tokens, see document *AN708: Setting Smart Energy Certificates for Zigbee Devices. Usage*: Required by Zigbee 3.0 and Zigbee Smart Energy compliant devices.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Offset from Lock Bits starting address				
(Series 1)	(Series 2)	Size (Bytes)	Name	Description
0x0284	0x0260	2	TOKEN_MFG_SECURITY_CONFIG	(EmberZNet PRO only) Defines the security policy for application calls into the stack to retrieve key values. The API calls emberGetKey() and emberGetKeyTableEntry() are affected by this setting. This prevents a running application from reading the actual encryption key values from the stack. This token may also be set at runtime with emberSetMfgSecurityConfig() (see that API for more information). The stack utilizes the emberGetMfgSecurityConfig() to determine the current security policy for encryption keys. The token values are mapped to the EmberKeySettings stack data type (defined in ember-types.h). See the following table for the mapping of token values to the stack values. *Usage:* Optional for EmberZNet NCP devices wishing to limit access to security data over the serial interface.
0x0286	0x0286	16	TOKEN_MFG_SECURE_BOOTLOADER_KEY	This token holds the 128 bit key used by the secure bootloader to decrypt encrypted Ember Bootloader (EBL) and Gecko Bootloader (GBL) files. A value of all F's is considered an invalid key and will not be used by the secure bootloader. *Usage:* Required only if using one of the Ember bootloaders with the "secure" prefix, or the Gecko Bootloader with support for encrypted GBL files enabled. See *UG266: Gecko Bootloader User Guide for GSDK 3.2 and Lower* or Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher, for more information on the Gecko Bootloader.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Offset from Lock Bits starting address				
(Series 1)	(Series 2)	Size (Bytes)	Name	Description
0x0296	0x0298	148	TOKEN_MFG_CBKE_283K1_DATA	Defines the security data necessary for Smart Energy 1.2 devices using the ECC 283k1 curve. It is used for Certificate Based Key Exchange to authenticate a device on a Smart Energy network. The first 74 bytes are the device's implicit certificate, the next 37 bytes are the Root Certificate Authority's Public Key, the next 36 bytes are the device's private key (the other half of the public/private key pair stored in the certificate), and the last byte is a flags field. The flags field should be set to 0x00 to indicate that the security data is initialized. For more information on the Smart Energy tokens, see document *AN708: Setting Smart Energy Certificates for Zigbee Devices*. *Usage:* Required for Zigbee Smart Energy 1.2 devices.
0x34A	0x034C	32	TOKEN_MFG_SIGNED_BOOTLOADER_KEY_X	These tokens hold the 256-bit X and Y components of the ECDSA P-256 public key used by the Gecko bootloader to perform cryptographic signature verification of signed GBL files ("secure bootload") and signed application images ("secure boot"). A value of all F's is considered an invalid key. See *UG266: Gecko Bootloader User Guide for GSDK 3.2 and Lower* or Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher, for more information on the Gecko Bootloader. *Usage*: Required only if using the Gecko bootloader with secure boot or support for signed GBL files is enabled.
0x36A	0x036C	32	TOKEN_MFG_SIGNED_BOOTLOADER_KEY_Y	

Copyright © 2025 Silicon Laboratories. All rights reserved.

Offset from Lock Bits starting address				
(Series 1)	(Series 2)	Size (Bytes)	Name	Description
0x038A	0x038C	34	TOKEN_MFG_THREAD_JOIN_KEY	This token is for saving the Join Key. The Join Key has a max length of 32 Bytes. *Usage*: Optional. Use with the Silicon Labs Thread Stack only.
0x0448	0x03B0	16	TOKEN_MFG_NVM3_CRYPTO_KEY	Manufacturer Token space for Storing NVM3 Crypto Key.

Table: Mapping of EmberKeySettings to `TOKEN_MFG_SECURITY_CONFIG`

EmberKeySettings Value	TOKEN_MFG_SECURITY_CONFIG Value
`EMBER_KEY_PERMISSIONS_NONE`	0x0000
`EMBER_KEY_PERMISSIONS_READING_ALLOWED`	0x00FF
`EMBER_KEY_PERMISSIONS_HASHING_ALLOWED`	0xFF00

Copyright © 2025 Silicon Laboratories. All rights reserved.

Testing

Testing

At this point, you may want to perform a simple send/receive test on the new device to determine its range and generally test its radio functionality.

> **Note**: Zigbee EmberZNet SDK 7.0 introduced a new component-based architecture, along with a Project Configurator and other tools to replace AppBuilder and plugin configuration. In general, the new software components are comparable to the plugins. For more information, see AN1301: Transitioning from Zigbee EmberZNet SDK 6.x to SDK 7.x.

For customers who are using mature applications with the EmberZNet PRO SDK, Silicon Labs recommends using the manufacturing test library (also known as "mfglib"), a set of APIs or NCP serial commands provided for IEEE 802.15.4-based functional testing of the radio from within a normal EmberZNet PRO application for an SoC or host platform. This library is typically used in the low-volume and high-volume phases of testing and manufacturing, where additional programming steps for a dedicated RF test application add unnecessary test time. For SoC applications based on EmberZNet PRO, as well as for custom Zigbee NCP builds through the NCP Framework, a "Manufacturing Library" component/plugin is provided as part of the corresponding application framework to enable this functionality in the build. The user must then add code to the application to invoke these functions (described in stack/include/mfglib.h as well as *UG100: EZSP Reference Guide*) in the desired context. A serial command line-driven interface to these functions is available in EmberZNet PRO SoC and Host applications via the Manufacturing Library CLI component/plugin, and an over-the-air (OTA)-driven interface is available in EmberZNet PRO SoC applications via the Manufacturing Library OTA component/plugin.

If the new device fails to successfully transmit or receive packets with the known good device, you may want to attach the new device to a signal generator or network analyzer to verify that generated packets on the target frequency can be received and that the new device can transmit accurately at the center frequency of the selected channel. Other tests may be required for FCC or CE compliance testing.

> **Note**: The manufacturing test library is not available with Silicon Labs Connect.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Using Tokens for Non-Volatile Data Storage

Using Tokens for Non-Volatile Data Storage

> NOTE: This section replaces *AN1154: Using Tokens for Non-Volatile Data Storage*. Further updates to this application note will be provided here.

This application note describes tokens and shows how to use them for non-volatile data storage in EmberZNet PRO and Silicon Labs Connect (part of the Flex SDK).

Key Points

- Defines tokens.
- Discusses dynamic token access.
- Explains manufacturing tokens.
- Describes default tokens.
- Details how to add a custom application dynamic token.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Introduction

Introduction

This application note explains how to use tokens for non-volatile data storage. First, the concept of tokens, different types of tokens, and how to categorize them from multiple perspectives is introduced. Then, practical instructions on how to create and access (that is, read and write) tokens from the application are provided.

After reading this application note you should expect to understand what tokens are, why and when you would use them, and how to create and access them from certain types of applications, currently EmberZNet PRO and Silicon Labs Connect (part of the Flex SDK).

A new infrastructure was introduced with Gecko SDK Suite (GSDK) v3.x. As part of that infrastructure update, the HAL APIs used to access and manage tokens have been replaced with Token Manager APIs. Variances are noted throughout.

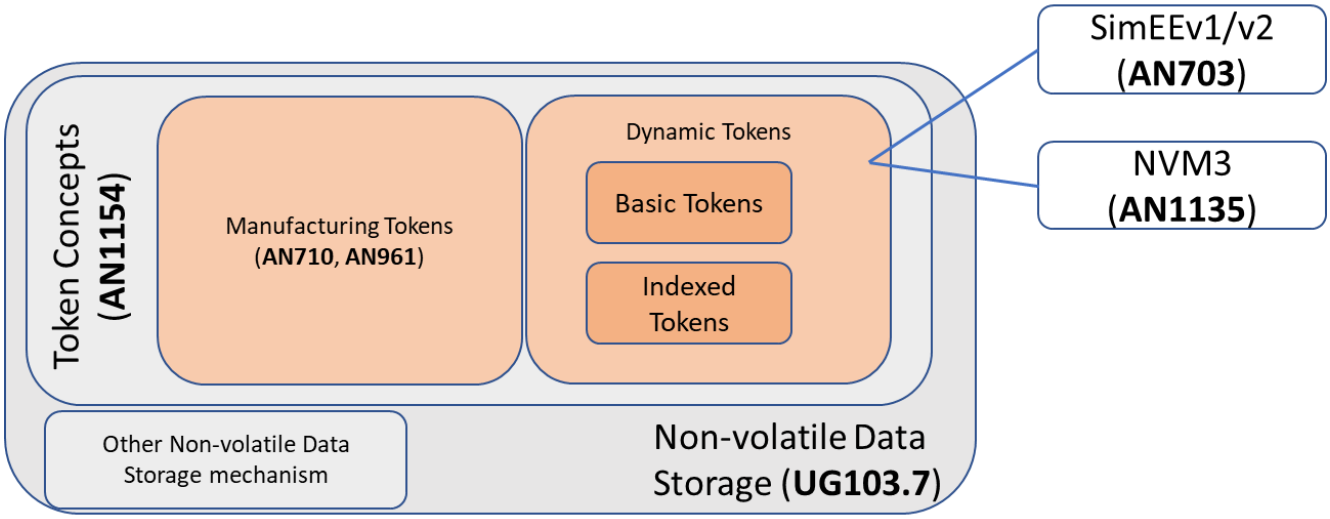
Copyright © 2025 Silicon Laboratories. All rights reserved.

About Tokens

Concept of Tokens

A token is an abstract data constant that has special persistent meaning for an application. Tokens are used to preserve certain important data across reboots and during power loss. These tokens are stored in non-volatile memory. A token has two parts: a token key and token data. The token key is a unique identifier that is used to store and retrieve the token data. In many cases, the word "token" is used quite loosely to mean the token key, the token data, or the combination of key and data. Usually it is clear from the context which meaning to use. In this application note, "token" always refers to the key + data pair.

The rest of this section examines the categorization of tokens from a few different angles. The following figure is a conceptual illustration of the "universe" of tokens, and how this application note relates to other documents on non-volatile data storage. Understanding the types of tokens is relevant to understanding how to create and use them, as described in Section Creating and Accessing Tokens.



Types of Tokens: Manufacturing Tokens and Dynamic Tokens

Tokens are written differently depending on how they are going to be used.

Manufacturing tokens are written either only once or very infrequently during the lifetime of the chip, and they are stored at absolute addresses of the flash. For more information about manufacturing and token programming, refer to document Bringing up Custom Devices for the EFR32MG and EFR32FG Families.

Dynamic tokens can be accessed (both read and written) frequently. They are stored in a dedicated area of the flash which uses a memory-rotation algorithm to prevent flash overuse. Silicon Labs offers three different dynamic token implementations: Simulated EEPROM Version 1 (SimEEv1), Simulated EEPROM Version 2 (SimEEv2), and Third Generation Non-Volatile Memory (NVM3). For an overview of non-volatile data storage concepts and a description of the three implementations, see *UG103.7: Non-Volatile Data Storage Fundamentals*.

The fundamental purpose of the dynamic token system as compared to generic RAM usage is to allow the token data to persist across reboots and during power loss. By using the token key to identify the proper data, the application requesting

Copyright © 2025 Silicon Laboratories. All rights reserved.

the token data does not need to know the exact storage location of the data. This simplifies application design and code reuse.

Because EFR32 process technology does not offer an internal EEPROM, the storage mechanism for dynamic tokens is implemented to use a section of internal flash memory for stack and application token storage. For SimEEv1, the EFR32 Series 1 uses 8 kB of upper flash memory for non-volatile data storage. SimEEv2 requires 36 kB of upper flash storage. Using SimEEv2 requires a special key from Silicon Labs. The purpose of this is to prevent an unintended upgrade from Version 1 to Version 2. The only way to downgrade requires full data loss and upgrading might not retain every token. With NVM3, storage size is configurable from 3 flash pages on up.

Parts that use dynamic tokens to store non-volatile data have different levels of flash performance with respect to guaranteed erase cycles. EFR32 flash cells are qualified for a guaranteed 10,000 erase cycles. See the datasheet for your specific part in order to determine the number of guaranteed erase cycles across voltage and temperature. Due to the limited erase cycles, the storage mechanism for dynamic tokens implements a wear-leveling algorithm that effectively extends the number of erase cycles for individual tokens.

Silicon Labs recommends that application designers familiarize themselves with the different dynamic token storage mechanisms, so that they design the application's use of tokens for optimal flash erase cycles. Refer to document *AN703: Using Simulated EEPROM Version 1 and Version 2 for the EFR32 Series 1 SoC Platforms*, for more information about SimEEv1/v2. Note that SimEEv1/v2 are not implemented on EFR32 Series 2. Refer to Using Third Generation Non-Volatile Memory (NVM3) Data Storage for more information about NVM3. If you are unsure where to start and are developing for EFR32, Silicon Labs recommends NVM3. NVM3 is more configurable (allows for better balance of token capacity versus reserved flash) and is compatible with DMP in case the application needs to go that way in the future. Here are some very general guidelines on the mechanisms' distinguishing features:

- Simulated EEPROM version1: low data requirement, single protocol, EFR32 Series 1.
- Simulated EEPROM version2: high data requirement, single protocol, EFR32 Series 1.
- NVM3: high data requirement, dynamic multi-protocol, EFR32 Series 2.

Types of Dynamic Tokens: Basic Tokens and Indexed Tokens

There are two types of dynamic tokens, with the types distinguished by their format:

- **Non-indexed or basic** dynamic tokens. These can be thought of as a simple char variable type. They can be used to store an array, but if one element changes the entire array must be rewritten.
  - A counter token is a special type of non-indexed dynamic token meant to store a number that increments by 1 at a time.
- **Indexed** dynamic tokens can be considered as a linked array of char variables where each element is expected to change independently of the others and therefore is stored internally as an independent token and accessed explicitly through the token API.

More information on basic and indexed tokens can be found in *AN703: Using Simulated EEPROM Version 1 and Version 2 for the EFR32 Series 1 SoC Platforms* and Using Third Generation Non-Volatile Memory (NVM3) Data Storage.

Types of Tokens: Default Tokens and Custom Tokens

Depending on whether a token is provided by Silicon Labs as part of a networking protocol stack or created by a user, the tokens can also be categorized as default tokens or custom tokens.

Default Tokens

The networking stack contains default tokens that are grouped by their software purpose:

- **Manufacturing Tokens** are set at manufacturing time and cannot be changed by the application.
- **Stack Tokens** are runtime configuration options set by the stack. These dynamic tokens should not be changed by the application.
- **Application Framework Tokens** are application tokens used by the Application Framework and generated by AppBuilder in Connect v2.7.x and EmberZNet SDK v6.7.x/v6.8.x. These dynamic tokens should not be changed by the application after project generation. Examples of these are ZCL attribute tokens and plugin tokens.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Custom Tokens

In addition to default tokens, users can add these types of tokens specific to their application:

- **Custom Manufacturing Tokens** are defined by the user and set at manufacturing time.
- **Custom Application Tokens** are additional application tokens users may add to meet their unique application needs for non-volatile data storage.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Creating and Accessing Tokens

Creating and Accessing Tokens

Now that you understand what tokens are and a few ways they may be categorized depending on how you view them, you can learn how to use tokens. This includes knowing where to find default tokens, how to create new tokens, and how to read and potentially modify tokens. Keep in mind that methods may vary depending on the type of tokens involved.

Token Header Files

A token header file is simply a .h file that contains token definitions. Manufacturing tokens and dynamic tokens have separate token header files. There may be more than one header file in an application for dynamic tokens: one for stack tokens, variable number of application framework tokens, and possibly one for custom application tokens.

Creating Dynamic Tokens

Adding a dynamic token to the header file involves three steps:

1. Define the token name.
2. Add any typedef needed for the token, if it is using an application-defined type.
3. Define the token storage.

The rest of this section looks at each of these steps one at a time.

Define the Token Name

When defining the name, do not prepend the word `TOKEN`. For SimEEv1/v2 dynamic tokens, use the word `CREATOR`:

```
/**
* Custom Application Tokens
*/
// Define token names here
#define CREATOR_DEVICE_INSTALL_DATA (0x000A)
#define CREATOR_HOURLY_TEMPERATURES (0x000B)
#define CREATOR_LIFETIME_HEAT_CYCLES (0x000C)
```

For NVM3 dynamic tokens, use the word `NVM3KEY`. Note that the example below assumes a Zigbee application. For a different stack, the NVM3 domain would be different. Note also that the `NVM3KEY` value for `HOURLY_TEMPERATURES` is set to a value where the subsequent 0x7F values are unused because this is an indexed token. Refer to Using Third Generation Non-Volatile Memory (NVM3) Data Storage for more information on NVM3 default instance key spaces and restrictions on selecting `NVM3KEY` values for indexed tokens.

```
/**
* Custom Zigbee Application Tokens
*/
// Define token names here
#define NVM3KEY_DEVICE_INSTALL_DATA (NVM3KEY_DOMAIN_USER | 0x000A)
// This key is used for an indexed token and the subsequent 0x7F keys are also reserved
#define NVM3KEY_HOURLY_TEMPERATURES (NVM3KEY_DOMAIN_USER | 0x1000)
#define NVM3KEY_LIFETIME_HEAT_CYCLES (NVM3KEY_DOMAIN_USER | 0x000C)
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

These examples define the token key and link it to a programmatic variable. The token names are actually `DEVICE_INSTALL_DATA, HOURLY_TEMPERATURES` and `LIFETIME_HEAT_CYCLES`, with different tags prepended to the beginning depending on the usage. Thus, they are referred to in the example code as `TOKEN_DEVICE_INSTALL_DATA`, and so on.

The token key values must be unique within this device. The token key is critical to linking application usage with the proper data. As such, a unique key should always be used when defining a new token or even changing the structure of an existing token. Always using a unique key guarantees a proper link between application and data. `CREATOR` code values are 16-bit and `NVM3KEY` code values are 20-bit. For SimEEv1/v2, the first bit is reserved for manufacturing tokens, stack tokens, and those application tokens defined by the application framework, so all custom tokens should have a token key less than 0x8000.

For NVM3, custom application tokens should use the `NVM3KEY_DOMAIN_USER` range so they do not collide with the stack tokens in the `NVM3KEY_DOMAIN_ZIGBEE` range.

Define the Token Type

The token type can be either a built-in C data type or defined as a custom data structure using `typedef`. Note that the token type must be defined only in one place, as the compiler will complain if the same data structure is defined twice.

Each token in the code examples in Section Creating Dynamic Tokens is a different type; the `HOURLY_TEMPERATURES` and `LIFETIME_HEAT_CYCLES` types are built-in types in C, while the `DEVICE_INSTALL_DATA` type is a custom data structure:

```
#ifdef DEFINETYPES
// Include or define any typedef for tokens here
typedef struct {
int8u install_date[11] /** YYYY-mm-dd + NULL */
int8u room_number; /** The room where this device is installed */
} InstallationData_t;
#endif //DEFINETYPES
```

Define the Token Storage

After any custom types are defined, the token storage is defined. This informs the token management software about the tokens being defined. Each token, whether custom or default, gets its own entry in this part:

```
#ifdef DEFINETOKENS
// Define the actual token storage information here
DEFINE_BASIC_TOKEN(DEVICE_INSTALL_DATA,
InstallationData_t,
{0, {0,...}})
DEFINE_INDEXED_TOKEN(HOURLY_TEMPERATURES, int16u, HOURS_IN_DAY, {0,...})
DEFINE_COUNTER_TOKEN(LIFETIME_HEAT_CYCLES, int32u, 0}
#endif //DEFINETOKENS
```

The following expands on each step in this process.

```
DEFINE_BASIC_TOKEN(DEVICE_INSTALL_DATA,
InstallationData_t,
{0, {0,...}})
```

`DEFINE_BASIC_TOKEN` takes three arguments: the name (`DEVICE_INSTALL_DATA`), the data type (`InstallationData_t`), and the default value of the token if it has never been written by the application (`{0, {0,...}}`).

The default value takes the same syntax as C default initializers. In this case, the first value (`room_number`) is initialized to 0, and the next value (`install_date`) is set to all 0s because the {0,...} syntax fills the remainder of the array with 0.

The syntax of `DEFINE_COUNTER_TOKEN` is identical to `DEFINE_BASIC_TOKEN`.

Copyright © 2025 Silicon Laboratories. All rights reserved.

`DEFINE_INDEXED_TOKEN` requires a length of the array—in this case, `HOURS_IN_DAY` or 24. Its final argument is the default value of every element in the array. Again, in this case it is initialized to all 0s.

Accessing Dynamic Tokens With HAL APIs

Refer to this information if you are developing with any EmberZNet or Connect in GSDK v2.7x, and with EmberZNet in GSDK v3.0.

The networking stack provides a simple set of APIs for accessing token data. The APIs differ slightly depending on the type of the tokens. The rest of this section discusses how to access tokens depending on the types. You can find the full documentation in the stack API Reference.

Accessing Basic (Non-indexed) Tokens

The non-indexed/basic token API functions include:

```
void halCommonGetToken(data, token)
void halCommonSetToken(token, data)
```

In this case, 'token' is the token key, and 'data' is the token data. Note that `halCommonGetToken()` and `halCommonSetToken()` are general token APIs that can be used for both basic dynamic tokens and manufacturing tokens. See Accessing Manufacturing Tokens.

The following example explains the usage of these APIs. Assume an application needs to store configuration data at installation time. A basic token has been defined to use the token key `DEVICE_INSTALL_DATA` and the data structure looks like this:

```
typedef struct {
int8u install_date[11] /** YYYY-mm-dd + NULL */
int8u room_number; /** The room where this device is installed */
} InstallationData_t;
```

Then you can access it with a code snippet like this:

```
InstallationData_t data;
// Read the stored token data
halCommonGetToken(&data, TOKEN_DEVICE_INSTALL_DATA);
// Set the local copy of the data to new values
data.room_number = < user input data >
MEMCOPY(data.install_date, < user input data>, 0, sizeof(data.install_date));
// Update the stored token data with the new values
halCommonSetToken(TOKEN_DEVICE_INSTALL_DATA, &data);
```

Accessing Counter Tokens

There is a special API to increment the counter token (which is a special type of non-indexed tokens):

```
void halCommonIncrementCounterToken(token)
```

Note that although you can write the counter token with the common `halCommonSetToken()` call, doing so is inefficient and defeats the purpose of using a counter token.

The following example explains the usage of a counter token and the special API to increment the counter token. Counting the number of heating cycles a thermostat has initiated is a perfect use for a counter token. Assume it is named `LIFETIME_HEAT_CYCLES`, and it is an int32u data type.

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
void requestHeatCycle(void) {
/// < application logic to initiate heat cycle >
halCommonIncrementCounterToken(TOKEN_LIFETIME_HEAT_CYCLES);
}
int32u totalHeatCycles(void) {
int32u heatCycles;
halCommonGetToken(&heatCycles, TOKEN_LIFETIME_HEAT_CYCLES);
return heatCycles;
}
```

Note that to read a counter token, use `halCommonGetToken()` just as you would read a general basic token.

Accessing Indexed Tokens

The indexed token API functions include:

```
void halCommonGetIndexedToken(data, token, index)
void halCommonSetIndexedToken(token, index, data)
```

The following example explains the usage of these APIs for an indexed token. To store a set of similar values, such as an array of preferred temperature settings throughout the day, use the default data type int16s to store the desired temperatures and define an indexed token called `HOURLY_TEMPERATURES`.

A local copy of the entire data set would look like this:

```
int16s hourlyTemperatures[HOURS_IN_DAY]; /** 24 hours per day */
```

In the application code, you can access or update just one of the values in the day using the indexed token functions:

```
int16s getCurrentTargetTemperature(int8u hour) {
int16s temperatureThisHour = 0; /** Stores the temperature for return */
if (hour < HOURS_IN_DAY) {
halCommonGetIndexedToken(&temperatureThisHour,
TOKEN_HOURLY_TEMPERATURES, hour);
}
return temperatureThisHour;
}
void setTargetTemperature(int8u hour, int16s targetTemperature) {
if (hour < HOURS_IN_DAY) {
halCommonSetIndexedToken(TOKEN_HOURLY_TEMPERATURE, hour,
&temperatureThisHour);
}
}
```

Accessing Dynamic Tokens with Token Manager APIs

Refer to this information if you are developing with Connect in GSDK v3.0.

The networking stack provides a simple set of APIs for accessing token data. The APIs differ slightly depending on the type of the tokens. The rest of this section discusses how to access tokens depending on the types. You can find the full documentation in the stack API Reference.

Accessing Basic (Non-Indexed) Tokens

The non-indexed/basic token API functions include:

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
Ecode_t sl_token_get_data(uint32_t token,
uint32_t index,
void *data,
uint32_t length);
Ecode_t sl_token_set_data(uint32_t token,
uint32_t index,
void *data,
uint32_t length);
```

In this case, 'token' is the token key, ‘index’ is 1 for non-indexed/basic tokens, 'data' is the token data, and ‘length’ is the size in bytes of the data. Note that `sl_token_get_data()` and `sl_token_set_data()` are specific for basic dynamic tokens. For manufacturing tokens see Accessing Manufacturing Tokens.

The following example explains the usage of these APIs. Assume an application needs to store configuration data at installation time. A basic token has been defined to use the token key `DEVICE_INSTALL_DATA` and the data structure looks like this:

```
typedef struct {
int8u install_date[11] /** YYYY-mm-dd + NULL */
int8u room_number; /** The room where this device is installed */
} InstallationData_t;
```

Then you can access it with a code snippet like this:

```
InstallationData_t data;
// Read the stored token data
sl_token_get_data(TOKEN_DEVICE_INSTALL_DATA, 1, &data, sizeof(data));
// Set the local copy of the data to new values data.room_number = < user input data >
MEMCOPY(data.install_date, < user input data>, 0, sizeof(data.install_date));
// Update the stored token data with the new values
sl_token_set_data(TOKEN_DEVICE_INSTALL_DATA, 1, &data, sizeof(data));
```

Accessing Counter Tokens

There is a special API to increment the counter token (which is a special type of non-indexed tokens):

```
Ecode_t sl_token_increment_counter(uint32_t token);
```

Note that although you can write the counter token with the common `sl_token_set_data()` call, doing so is inefficient and defeats the purpose of using a counter token.

The following example explains the usage of a counter token and the special API to increment the counter token. Counting the number of heating cycles a thermostat has initiated is a perfect use for a counter token. Assume it is named `LIFETIME_HEAT_CYCLES`, and it is an int32u data type.

```
void requestHeatCycle(void) {
/// < application logic to initiate heat cycle >
sl_token_increment_counter(TOKEN_LIFETIME_HEAT_CYCLES);
}
int32u totalHeatCycles(void) {int32u heatCycles;
sl_token_get_data(TOKEN_LIFETIME_HEAT_CYCLES, 1, &heatCycles, sizeof(heatCycles));
return heatCycles;
}
```

Note that to read a counter token, use `sl_token_get_data()` just as you would read a general basic token.

Accessing Indexed Tokens

Copyright © 2025 Silicon Laboratories. All rights reserved.

The indexed token API functions are the same as the non-indexed/basic token API functions. The only difference is the 'index' parameter can be something other than 1.

The following example explains the usage of these APIs for an indexed token. To store a set of similar values, such as an array of preferred temperature settings throughout the day, use the default data type int16s to store the desired temperatures and define an indexed token called `HOURLY_TEMPERATURES`.

A local copy of the entire data set would look like this:

```
int16s hourlyTemperatures[HOURS_IN_DAY]; /** 24 hours per day */
```

In the application code, you can access or update just one of the values in the day using the indexed token functions:

```
int16s getCurrentTargetTemperature(int8u hour) {
int16s temperatureThisHour = 0; /** Stores the temperature for return */
if (hour < HOURS_IN_DAY) {
 sl_token_get_data(TOKEN_HOURLY_TEMPERATURES, hour, &temperatureThisHour, sizeof(temperatureThisHour));
}
return temperatureThisHour;
}
void setTargetTemperature(int8u hour, int16s targetTemperature) {
if (hour < HOURS_IN_DAY) {
 sl_token_set_data(TOKEN_HOURLY_TEMPERATURE, hour, &temperatureThisHour, sizeof(temperatureThisHour));
}
}
```

Manufacturing Tokens

Manufacturing tokens are defined in a similar way as basic (non-indexed) dynamic tokens, but use the DEFINE_MFG_TOKEN in the manufacturing token header, instead of the other `DEFINE_*_TOKEN` macros. Note that the `CREATOR_*` prefix – not `NVM3KEY_*` - should always be used with manufacturing tokens, even when the NVM plugin is being used for non-volatile storage. Refer to Creating Dynamic Tokens and to the API documentation on the DEFINE_MFG_TOKEN macro for more information.

There is a major difference, however, on where manufacturing tokens are stored and how they are accessed. Manufacturing tokens reside in the dedicated flash page for manufacturing tokens (with fixed absolute addresses). The rest of this section describes the necessary considerations for accessing manufacturing tokens.

Accessing Manufacturing Tokens

Manufacturing tokens, as the name suggests, are usually written once at manufacturing time into fixed locations in a dedicated flash page. Because their addresses are fixed, they can be easily read from external programming tools. Note, however, when the Read Protection feature has been enabled on the chip or on the dedicated flash page, manufacturing tokens can only be read from on-chip code.

Manufacturing tokens are not meant to be written often or from on-chip code, because they are at fixed locations. The same flash cell cannot be written repeatedly without erase operations in between. Writing a manufacturing token from on-chip code works only if the token is currently in an erased state. This means that manufacturing tokens can only be overwritten with external programming tools and not with on-chip code. Overwriting any manufacturing token that has been already written requires erasing the dedicated flash page for the manufacturing tokens. If Read Protection is enabled, it must be disabled first, which will erase the contents of the chip as a side effect. Manufacturing tokens are not wear-leveled so they should be overwritten sparingly.

Access manufacturing tokens with their own dedicated APIs, which take the same parameters as the basic token APIs:

- HAL: `halCommonGetMfgToken()` and `halCommonSetMfgToken()`
- Token Manager: `sl_token_get_manufacturing_data()` and `sl_token_set_manufacturing_data()`

The two primary purposes for using the dedicated manufacturing token access APIs are:

Copyright © 2025 Silicon Laboratories. All rights reserved.

1. For slightly faster access;
2. For access early in the boot process before `emberInit()` or `sl_token_init()` is called.

If you are using the HAL APIs, manufacturing tokens can also be accessed through the basic token APIs: `halCommonGetToken()` and `halCommonSetToken()`.

Accessing Default and Custom Tokens

Where to Find Default Token Definitions

To view the stack tokens, refer to the file:

```
`<install-dir>/stack/config/token-stack.h`
```

To view the Application Framework tokens, refer to the file `<project_name>_tokens.h` and the protocol-specific token file (such as `znet-token.h`) under your project directory after the project has been generated in AppBuilder in Connect v2.7.x and EmberZNet SDK v6.7.x/v6.8.x SDKs. `<project_name>_tokens.h` contains tokens for ZCL attributes that the application has selected to be stored in non-volatile memory. The protocol-specific token file includes plugin token headers and the custom application token header.

To view the manufacturing tokens for the EFR32 series of chips refer to the following files, respectively:

```
<install-dir>/hal/micro/cortexm3/efm32/token-manufacturing.h
```

Search for `CREATOR` to see the defined names. If the entire file seems overwhelming, focus only on the section describing the tokens. Some of the fixed manufacturing tokens may be set by the manufacturer when the board is created. For example, a custom EUI-64 address may be set by the vendor to override the internal EUI-64 address provided by Silicon Labs. Other tokens, such as the internal EUI-64, cannot be overwritten.

Add Custom Tokens

Refer to Section Creating Dynamic Tokens on the methods and APIs for creating custom tokens. You can also refer to the stack token definition file as mentioned near the top of Section Where to Find Default Token Definitions as a guide for creating custom application tokens.

After creating the tokens in a custom token header file, you need one more step: add the header file to the application by using the **Includes** tab in the .isc file in Simplicity Studio under the "Token Configuration" section. If you define both custom manufacturing tokens and custom application tokens, Silicon Labs recommends that you separate them into two different header files.

> **Note**: For custom token files, the inclusion guards (#ifndef) seen in `token-stack.h` should not be used.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Token Manager Component

Token Manager Component

To use the Token Manager, the token_manager component (token_manager.slcc) should be added to the project. The token_manager component will default to using the token_manager_nvm3 component, which brings in the NVM3 system. Using NVM3 for storage is highly recommended.

To use Simulated EEPROM v1 or v2 (SimEEv1, SimEEv2) for storage, the component token_manager_simee1 or token_manager_simee2 should be part of the project along with token_manager.

To upgrade SimEEv2 to NVM3, the sim_eeprom2_to_nvm3_upgrade component should be used along with token_manager. This upgrade component will pull in the necessary SimEE and NVM3 code.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Overview

Security

Silicon Labs offers a range of security features depending on the part you are using and your application and production needs. This section provides background on security and how to use the available security features.

- IoT Security Fundamentals: Introduces the security concepts that must be considered when implementing an Internet of Things (IoT) system. Using the ioXt Alliance's eight security principles as a structure, it clearly delineates the solutions Silicon Labs provides to support endpoint security and what you must do outside of the Silicon Labs framework.
- Series 2 Secure Debug: Describes how to lock and unlock the debug access of EFR32 Gecko Series 2 devices. Many aspects of the debug access, including the secure debug unlock are described. The Debug Challenge Interface (DCI) and Secure Engine (SE) Mailbox Interface for locking and unlocking debug access are also included.
- Production Programming of Series 2 Devices (PDF): Provides details on programming, provisioning, and configuring Series 2 devices in production environments. Covers Secure Engine Subsystem of Series 2 devices, which runs easily upgradeable Secure Engine (SE) or Virtual Secure Engine (VSE) firmware.
- Anti-Tamper Protection Configuration and Use: Anti-Tamper Protection Configuration and Use - Shows how to program, provision, and configure the anti-tamper module on EFR32 Series 2 devices with Secure Vault.
- Authenticating Silicon Labs Devices using Device Certificates: How to authenticate an EFR32 Series 2 device with Secure Vault, using secure device certificates and signatures.
- Secure Key Storage: Explains how to securely "wrap" keys in EFR32 Series 2 devices with Secure Vault, so they can be stored in non-volatile storage.
- Programming Series 2 Devices Using the Debug Challenge Interface (DCI) and Serial Wire Debug (SWD) (PDF): Describes how to provision and configure Series 2 devices through the DCI and SWD.
- Integrating Crypto Functionality Using PSA Crypto Compared to Mbed TLS (PDF): Describes how to integrate crypto functionality into applications using PSA Crypto compared to Mbed TLS.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Security Key Migration

Security Key Migration between SDKs

With the release of SiSDK 2024.12, Connect stack migrated to the new, PSA backed security library for its IEEE 802.15.4-like MAC security.

This has the benefit of improved speed with the PSA backed key storage that Connect is using since GSDK 4.1.

PSA is required to use the secure element-based storage on the parts that have the feature, and provides a much more secure key storage on parts that do not have secure element.

The main difference is that the legacy implementation requires a PSA key with algorithm set to `PSA_ALG_ECB_NO_PADDING`, while the new implementation needs `PSA_ALG_AEAD_WITH_SHORTENED_TAG(PSA_ALG_CCM, 4)`. Note that the **security algorithm itself did not change** and devices using the legacy implementation can communicate with devices using the new implementation. The legacy implementation ran part of the encryption on the CPU, hence the different algorithm required from PSA.

If you start a new project, the example project in Simplicity Studio are all set up with the new key storage, and the legacy methods are not recommended for new designs.

However, if you need to update an existing application, the key stored in the legacy method is not supported and migration is not straight forward. This document targets to help with that.

The target of the migration is always to get the key in "plain text", i.e., unencrypted, as a 16 Byte array and reimport it with the new method.

Migration from GSDK 4.1.0 and Newer

Using Legacy Key Storage

Connect stack in SiSDK 2024.12 still has the legacy encryption library available which supports the key storage. To enable this option, go to the project's component explorer enable the **Deprecated** Quality components, and install **AES Security (Library) | Legacy** from the Connect category. You can also remove **AES Security (Library)**, which is the new version, but you can have both installed to help migration.

Moving the Legacy Key to the New Storage

If the key was stored with the flag `PSA_KEY_USAGE_EXPORT`, it's recommended to enable both AES libraries as documented in the previous chapter, and export the key:

```
uint8_t exported_key[EMBER_ENCRYPTION_KEY_SIZE];
size_t exported_key_length;
psa_status_t psa_status;

psa_status = psa_export_key(security_key_id, exported_key EMBER_ENCRYPTION_KEY_SIZE, &exported_key_length);
```

If the key is successfully exported, you can use the code in the bottom of the article to reimport it with the new requirements.

Note that if you relied on our examples to import the key into PSA, your key is not exportable. In this case, getting the key from a device is not possible. However, in most cases, the key is available somewhere else, e.g., on the gateway in plain text, so a key redistribution is the recommended method.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Migration from GSDK 4.0.x and Older

In GSDK 4.0.x and older, the key was stored in the token `TOKEN_STACK_SECURITY_KEY` with `tokTypeStackKey` type, which is simply a 16 Byte long uint8_t array. The token can be read by `halCommonGetToken()` . Note though that we used SimEEv1 as the default backend for tokens in GSDK v2, and we switched to NVMv3 in GSDK v3, so be careful, as other migrations might be necessary from such an old version of Connect.

Once you have the key as a plain text, you can use the code in the bottom of the article to reimport it with the new requirements.

Importing the Key

The following code is the recommended way to configure and import the key with SiSDK 2024.12 and newer. This code was copied from the Connect Sensor example, and should be executed before `emberNetworkInit()`

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
  EmberKeyData security_key = { .contents = <your security key as 16B uint8 array> };

  security_key_id = PSA_AES_KEY_ID;
  psa_key_attributes_t key_attr = psa_key_attributes_init();
  psa_status_t psa_status = psa_get_key_attributes(security_key_id, &key_attr);
  if (psa_status == PSA_ERROR_INVALID_HANDLE) {
    app_log_info("No PSA AES key found, creating one.\n");
    psa_set_key_id(&key_attr, security_key_id);
    psa_set_key_algorithm(&key_attr, PSA_ALG_AEAD_WITH_SHORTENED_TAG(PSA_ALG_CCM, 4));
    //Add PSA_KEY_USAGE_EXPORT below if you want an exportable key
    psa_set_key_usage_flags(&key_attr, PSA_KEY_USAGE_ENCRYPT | PSA_KEY_USAGE_DECRYPT);
    psa_set_key_type(&key_attr, PSA_KEY_TYPE_AES);
    psa_set_key_bits(&key_attr, 128);

#ifdef PSA_KEY_LOCATION_SLI_SE_OPAQUE
    psa_set_key_lifetime(&key_attr,
                         PSA_KEY_LIFETIME_FROM_PERSISTENCE_AND_LOCATION(
                           PSA_KEY_LIFETIME_PERSISTENT,
                           PSA_KEY_LOCATION_SLI_SE_OPAQUE));
#else
#ifdef MBEDTLS_PSA_CRYPTO_STORAGE_C
    psa_set_key_lifetime(&key_attr,
                         PSA_KEY_LIFETIME_FROM_PERSISTENCE_AND_LOCATION(
                           PSA_KEY_LIFETIME_PERSISTENT,
                           PSA_KEY_LOCATION_LOCAL_STORAGE));
#else
    psa_set_key_lifetime(&key_attr,
                         PSA_KEY_LIFETIME_FROM_PERSISTENCE_AND_LOCATION(
                           PSA_KEY_LIFETIME_VOLATILE,
                           PSA_KEY_LOCATION_LOCAL_STORAGE));
#endif
#endif

    psa_status = psa_import_key(&key_attr,
                            security_key.contents,
                            (size_t)EMBER_ENCRYPTION_KEY_SIZE,
                            &security_key_id);

    if (psa_status == PSA_SUCCESS) {
      app_log_info("Security key import successful, key id: %lu\n", security_key_id);
    } else {
      app_log_info("Security Key import failed: %ld\n", psa_status);
    }
  } else {
    app_log_info("PSA AES key found, using the existing one.\n");
  }

  em_status = emberSetPsaSecurityKey(security_key_id);
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

IoT Endpoint Security Fundamentals

IoT Endpoint Security Fundamentals

NOTE: This section replaces *UG103.05: IoT Endpoint Security Fundamentals*. Further updates to this user guide will be provided here.

This guide introduces the security concepts that must be considered when implementing an Internet of Things (IoT) system. Using the ioXt Alliance's eight security principles as a structure, this guide clearly delineates the solutions Silicon Labs provides to support endpoint security and what you must do outside of the Silicon Labs framework. Where appropriate, Silicon Labs' approach to our own security is offered as an example. This guide is designed for product developers and managers.

Silicon Labs' *Fundamentals* series covers topics that project managers, application designers, and developers should understand before beginning to work on an embedded networking solution using Silicon Labs chips, networking stacks such as EmberZNet PRO or Silicon Labs Bluetooth, and associated development tools. These guides can be used as a starting place for anyone needing an introduction to developing wireless networking applications, or who is new to the Silicon Labs development environment.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Overview

Overview

Securing the IoT is challenging. It is also mission-critical. Threats are continuously evolving, and the demand on product developers to keep up can be burdensome – particularly in low-cost, resource-constrained IoT products. Protecting your product in a connected world is a necessity, as customer data and modern online business models are increasingly targets for costly hacks that jeopardize end-user privacy and corporate brand damage. Silicon Labs is committed to working with the security community, customers, and other experts to bring state-of-the-art technology to help protect your connected portfolio.

Silicon Labs is a member of the ioXt (Internet of Secure Things) Alliance. The ioXt Alliance was formed to bring together wireless carriers, leading consumer product manufacturers, standards groups, compliance labs and government organizations to align baseline security requirements, to set the stage for testing and compatibility certification, and to work together building global standards for the IoT world.

The ioXt alliance has produced the ioXt Security Pledge (https://www.ioxtalliance.org/s/ioXt-SecurityPledge-booklet-final.pdf) The pledge covers eight principles in the areas of Security, Upgradability, and Transparency. Silicon Labs has adopted these principles in our own operations as well as in the products we provide. Our approach to these principles is described in this document.

Copyright © 2025 Silicon Laboratories. All rights reserved.



The above image and all pledge language is reproduced from *The ioXt Security Pledge: 8 Principles for Consumer Product Design and Manufacturing to Ensure Security, Upgradability & Transparency (2019)*.

Copyright © 2025 Silicon Laboratories. All rights reserved.

No Universal Passwords

No Universal Passwords

The product shall not have a universal password; unique security credentials will be required for operation. Universal passwords allow an attacker to easily gain access to any device. Therefore, products shall either have a unique password or require the user to enter a new password immediately upon first use.

It is your responsibility to ensure that your product enforces the creation of a unique password before activation.

Silicon Labs' products are designed to be configured by the manufacturer before being delivered to customers, and therefore passwords are outside of our scope. However, Silicon Labs tools are designed to support the various levels of security provided by the protocol in question. Most protocols offer different security levels, with tradeoffs between security level and other features such as ease of network formation. You need to review and decide on the level required by your application. For example:

- The EmberZNet Pro SDK supports a highly secure centralized trust-center-controlled method that replaces a device's factory-programmed link key with a key that is unique to each device on the network.
- Z-Wave 700 products come with a factory-programmed unique S2 keypair on first power-up, and support SmartStart commissioning through a package QR code containing the public key.
- Bluetooth options range from an unsecured "Just Works" approach to a LE Secure Connections Pairing model. Application designers can implement additional device authentication methods, such as through the companion smartphone app, to help ensure secure pairing even for devices without a user interface.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Secured Interfaces

Secured Interfaces

All product interfaces shall be appropriately secured by the manufacturer.

The interfaces to be secured will vary by product configuration. For example, in an NCP topology the NCP interface must be secured. Debug interfaces should always be locked. Wireless interfaces should be secured by using strong pairing and commissioning methods and by enabling encrypted and authenticated transmissions.

While securing the interfaces is in the end your responsibility, Silicon Labs provides the tools to enable that security.

Both Series 1 and Series 2 devices are designed to support securing debug access. For Series 1 devices, that functionality is provided through writing a Debug Lock word to the device. Unlocking the device erases the main application and the key material stored in the Lockbits page. For Series 2 devices, securing debug access is done through the device's Secure Engine. Both allow the developer to lock the debug port itself. See Silicon Labs Gecko Bootloader User's Guide for Series 3 and Higher, Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher (series 1 and 2 devices), or *UG266: Silicon Labs Gecko Bootloader User's Guide for GSDK 3.2 and Lower* for an overview of securing debug access, and Series 2 Secure Debug for details on the Series 2 implementation. *UG104: Testing and Debugging Applications for the Silicon Labs EFR32MG Platforms* provides an overview of the various application testing stages and the debug access (hardware and software) required in each.

For more information on Wireless interface security in the different protocols, see the following:

- Zigbee Security
- Bluetooth LE Fundamentals and relevant KBAs
- *AN1037: Apple HomeKit Over Bluetooth®*
- *UG235.03: Architecture of the Silicon Labs Connect Stack v2.x*
- *UG435.03: Architecture of the Silicon Labs Connect Stack v3.x*

Copyright © 2025 Silicon Laboratories. All rights reserved.

Proven Cryptography

Proven Cryptography

Product security shall use strong, proven, updatable cryptography using open, peer-reviewed methods and algorithms.

An important aspect of any IoT device is how secure the device is when it communicates with other devices, gateways, or the cloud. This standard mandates using proven cryptographic methods rather than attempting to implement your own.

Developers commonly secure communications such as TCP/IP connections, Bluetooth, Zigbee, or Z-Wave using the standardized and proven cryptographic methods native to the protocol. However, if a microcontroller sends sensitive information over a simple interface such as a UART to another microcontroller, it is important to realize that data should also be secured to prevent someone from snooping the UART line.

Silicon Labs offers a hardware CRYPTO module that provides an efficient acceleration of common cryptographic operations and allows these to be used efficiently with low CPU overhead. The CRYPTO module includes hardware accelerators for the Advanced Encryption Standard (AES), Secure Hash Algorithm SHA-1 and SHA-2 (SHA-224 and SHA-256), and modular multiplication used in ECC (Elliptic Curve Cryptography) and GCM (Galois Counter Mode). The CRYPTO module can autonomously execute and iterate a sequence of instructions to aid software and speed up complex cryptographic functions like ECC, GCM, and CCM (Counter with CBC-MAC).

In addition to the CRYPTO module, Silicon Labs includes mbed TLS as part of the Gecko Platform SDK. mbed TLS is open source software licensed by ARM Limited. It provides an SSL library that makes it easy to use cryptography and SSL/TLS in applications. mbed TLS supports software implementations of all crypto algorithms that are supported by TLS 1.2 as well as a build API that allows hardware drivers to replace the software implementations when cipher accelerators are supported by the platform. Its modular framework allows for subcomponents like the crypto libraries to be incorporated into a design independently of the SSL/TLS components, saving valuable code space and runtime RAM. mbed TLS supports SSLv3 up to TLSv1.2 communication by providing the following:

- TCP/IP communication functions: listen, connect, accept, read/write.
- SSL/TLS communication functions: init, handshake, read/write.
- X.509 functions: CRT, CRL and key handling
- Random number generation
- Hashing
- Encryption/decryption

These functions are split up into logical interfaces. They can be used separately to provide any of the above functions or to mix-and-match into an SSL server/client solution that utilizes a X.509 PKI. Examples of such implementations are provided with the source code. Components or plugins and APIs provide configuration interfaces accessible through the various SDK installations.

For more information, see the latest MCU and Peripheral Software Documentation for the target part at https://docs.silabs.com.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Security by Default

Security by Default

Product security shall be appropriately enabled by default by the manufacturer.

The state in which a product is shipped is up to the manufacturer. This standard mandates that any security features provided with the product be enabled before shipping. Customers should not have to turn security on; rather they should actively have to disable it. For example, Silicon Labs Z-Wave end-nodes and gateway SDKs ship with S2 cryptography and SmartStart network formation enabled by default.

Silicon Labs believes that product security should be considered during product design, and not as an afterthought. Within development environments, all Silicon Labs application security features may be enabled or disabled as appropriate during application development. Security must also be considered during device design and testing. Bringing Up Custom Devices for the EFR32MG and EFR32FG Families describes the security tokens (keys, certificates, and so on) that can be programmed into a custom device to support various types of security, including that provided by the Gecko Bootloader (see Signed Software Updates).

Copyright © 2025 Silicon Laboratories. All rights reserved.

Signed Software Updates

Signed Software Updates

The product shall only support signed software updates. While it is critical that all products be updatable, it is just as critical that these update images be secured. A manufacturer must cryptographically sign update images to prevent tampering during deployment. The product must not use unsigned updates, as they could be fraudulent.

Silicon Labs development tools support building signed upgrade images and securely updating devices in the field, through the Silicon Labs Gecko Bootloader. The Gecko Bootloader can be configured to perform a variety of functions, from device initialization to firmware upgrades. Key features of the bootloader are:

- Useable across Silicon Labs Gecko microcontroller and wireless microcontroller families
- In-field upgradeable
- Configurable
- Enhanced security features, including:
- Secure Boot: When Secure Boot is enabled, the bootloader enforces cryptographic signature verification of the application image on every boot, using asymmetric cryptography. This ensures that the application was created and signed by a trusted party.
- Signed upgrade image file: The Gecko Bootloader supports enforcing cryptographic signature verification of the upgrade image file. This allows the bootloader and application to verify that the application or bootloader upgrade comes from a trusted source before starting the upgrade process, ensuring that the image file was created and signed by a trusted party.
- Encrypted upgrade image file: The image file can also be encrypted to prevent eavesdroppers from acquiring the plaintext firmware image.

On Series 1 devices, the Gecko Bootloader has a two-stage design, first stage and main stage, where a minimal first stage bootloader is used to upgrade the main bootloader. The first stage bootloader only contains functionality to read from and write to fixed addresses in internal flash. To perform a main bootloader upgrade, the running main bootloader verifies the integrity and authenticity of the bootloader upgrade image file. The running main bootloader then writes the upgrade image to a fixed location in internal flash and issues a reboot into the first stage bootloader. The first stage bootloader verifies the integrity of the main bootloader firmware upgrade image, by computing a CRC32 checksum before copying the upgrade image to the main bootloader location.

On Series 2 devices, the Gecko Bootloader consists only of the main stage bootloader. The main bootloader is upgradable through the hardware peripheral Secure Engine. The Secure Engine provides functionality to install an image to address 0x0 in internal flash, by copying from a configurable location in internal flash. To perform a main bootloader upgrade, the running main bootloader verifies the integrity and authenticity of the bootloader upgrade image file. The running main bootloader then writes the upgrade image to the upgrade location in flash and requests that the Secure Engine install it. The Secure Engine is also capable of verifying the authenticity of the main bootloader update image against a root of trust. The Secure Engine itself is upgradable using the same mechanism.

In summary, Series 2 devices support a hardware root of trust and a Secure Boot process that verifies the authenticity and integrity of Gecko Bootloader, whereas in Series 1 devices, the authenticity and integrity of Gecko Bootloader are assumed trusted and are not explicitly checked.

The Gecko Bootloader can enforce application image security on two levels:

- Secure Boot refers to the verification of the authenticity of the application image in main flash on every boot of the device. When Secure Boot is enabled, the cryptographic signature of the application image in flash is verified on every boot, before the application is allowed to run. Secure Boot is not enabled by default in the example configurations provided by Silicon Labs, but enabling it is highly recommended to ensure the validity and integrity of firmware images.
- Secure Firmware Upgrade refers to the verification of the authenticity of an upgrade image before performing a bootload, and optionally enforcing that upgrade images are encrypted. The Secure Firmware Upgrade process uses symmetric

Copyright © 2025 Silicon Laboratories. All rights reserved.

encryption to encrypt the upgrade image, and asymmetric cryptography to sign the upgrade image in order to ensure its integrity and authenticity.

For more information on Silicon Labs' support for software update security, refer to the following:

Bootloaders in general: Bootloader Fundamentals

The Gecko Bootloader in general: Silicon Labs Gecko Bootloader User's Guide for Series 3 and Higher, Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher (series 1 and 2 devices), or *UG266: Silicon Labs Gecko Bootloader User's Guide for GSDK 3.2 and Lower*.

Using the Gecko Bootloader with specific protocols:

- Using the Gecko Bootloader with EmberZNet
- Using the Gecko Bootloader with Silicon Labs Connect
- Using the Gecko Bootloader with Silicon Labs Bluetooth Applications

Secure Boot on Series 2 devices: Series 2 Secure Boot with RTSL

Copyright © 2025 Silicon Laboratories. All rights reserved.

Automatically Applied Updates

Automatically Applied Updates

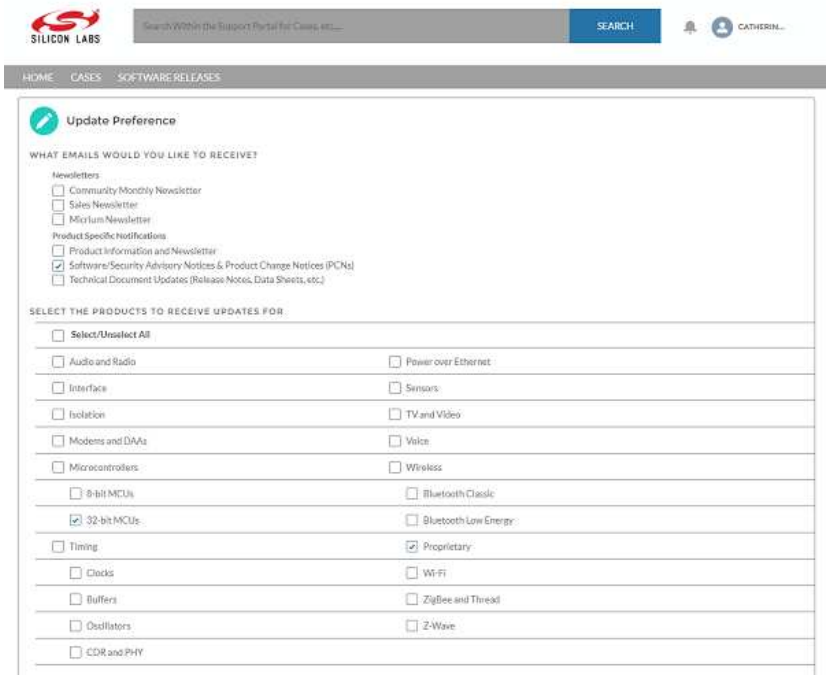
The manufacturer will act quickly to apply timely security updates. Whenever a security vulnerability is detected, the manufacturer will automatically apply a patch to the product. No user intervention will be required.

It is the manufacturer's responsibility to develop and implement automatic security updates. The design and methodology of such systems, for example through a Cloud-connected infrastructure or by direct intervention by a service representative, is up to you.

Silicon Labs will notify you of any security-related updates, as described in Vulnerability Reporting Program. Your responsibility is to evaluate the level of risk that vulnerability poses for your particular product and to integrate the update into your platform as appropriate so that your end users are protected. Updated components might include the protocol libraries, Secure Engine firmware inside the Series 2 family, or an SDK module such as the Gecko Bootloader that enforces secure OTA updates and secure boot functionality.

Silicon Labs recommends the following:

- Subscribe to security updates through our Salesforce portal. To review or change your subscriptions, log in to the portal, click **HOME** to go to the portal home page and then click the **Manage Notifications** tile. Make sure that **Software/Security Advisory Notices & Product Change Notices (PCNs)** is checked, and that you are subscribed at minimum for your platform and protocol. Click **Save** to save any changes.



- Do not turn off Simplicity Studio's update notification. Within Simplicity Studio, you can download updates and easily access product release notes.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Vulnerability Reporting Program

Vulnerability Reporting Program

The manufacturer shall implement a vulnerability reporting program, which will be addressed in a timely manner. All companies that offer Internet-connected devices and services shall provide a public point of contact as part of a vulnerability disclosure policy in order that security researchers and others are able to report issues. Disclosed vulnerabilities should be acted on in a timely manner.

Manufacturers are responsible for implementing their own program. For any individual vulnerability, you will need to weigh the value of transparency with your customers against the risk of malicious use of the information to exploit a vulnerability before it can be addressed. Silicon Labs makes similar decisions about how broadly to report security vulnerabilities discovered in our products.

Silicon Labs customers and security researchers can report security vulnerabilities in Silicon Labs hardware and software products on the Silicon Labs website: https://www.silabs.com/security/product-security.

Silicon Labs' Security Vulnerability Disclosure Policy may be found here:

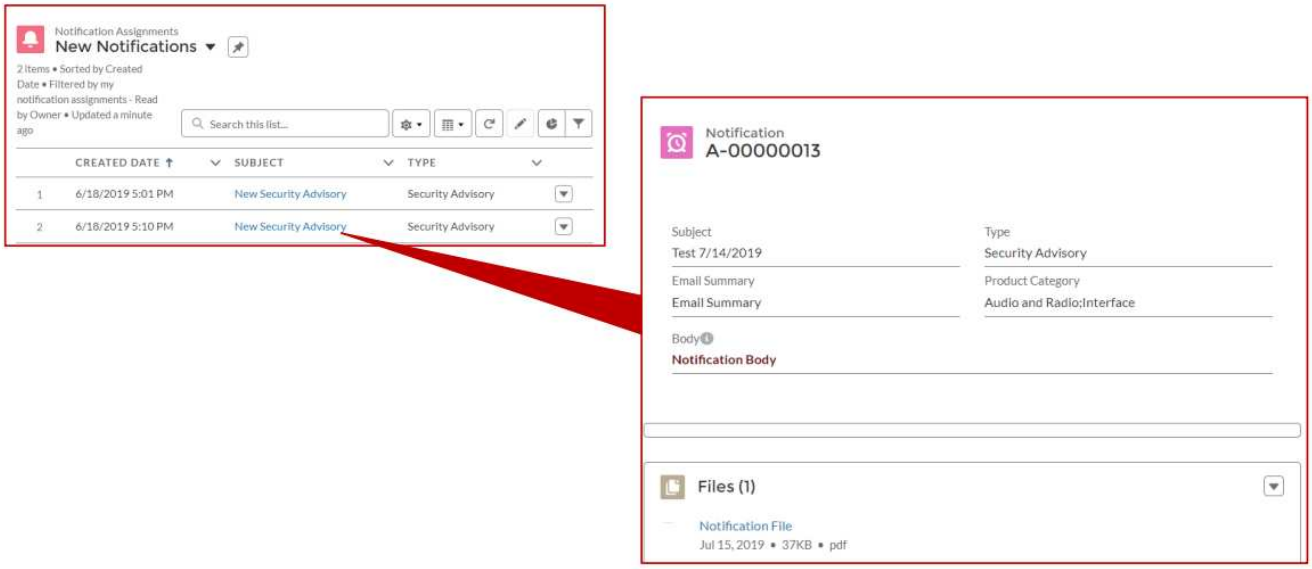
https://www.silabs.com/documents/public/miscellaneous/PS1012-Security_Vulnerability_Disclosure_Policy.pdf

Silicon Labs has a Product Security Incident Response Team (PSIRT) that is dedicated to the case management of reported security vulnerabilities. The PSIRT works with other Silicon Labs groups including Applications, Developers, Sales, and Marketing to assess reported vulnerabilities, perform technical analysis and determine an appropriate response. The key processes for addressing vulnerabilities include:

- **Triage**: Determines what is needed to reproduce the vulnerability.
- **Technical Analysis and Disposition**: Confirms the validity of the security vulnerability, its scope, and its impact, and provides a resolution or disposition decision. Silicon Labs scores incidents according to CVSS 3.1 (Common Vulnerability Scoring System): low, medium, high, critical.
- **Output**: Communicates with our customers. The level and method of disclosure beyond the reporting entity depends on the severity and scope of the vulnerability.

Silicon Labs' provides broad vulnerability reporting to customers subscribed through our Salesforce portal (see Automatically Applied Updates for information on how to subscribe). A subscribed customer will see Security Advisory notifications something like the following:

Copyright © 2025 Silicon Laboratories. All rights reserved.



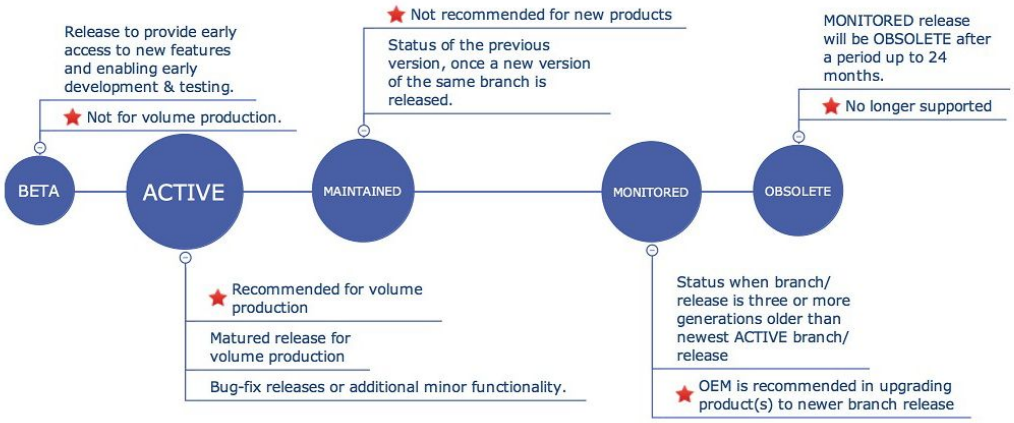
Copyright © 2025 Silicon Laboratories. All rights reserved.

Security Expiration Date

Security Expiration Date

The manufacturer shall be transparent about the period of time that security updates will be provided. Like a manufacturer's product warranty, there shall be transparency around the support period of security updates.

Manufacturers should provide details about product support at various stages and publish security expiration dates. Z-Wave's Protocol Lifecycle provides an example.



The Lifecycle details in what phases updates will be applied, and to what product branch. For details on the various phases and how the lifecycle is implemented for specific Z-Wave products, see:

https://www.silabs.com/products/development-tools/software/z-wave/embedded-sdk/life-cycle

Copyright © 2025 Silicon Laboratories. All rights reserved.

Next Steps

Next Steps

The Silicon Labs Security web page (https://www.silabs.com/security) contains links to a variety of general security-related resources. You may wish to bookmark the page, as it will be continually updated with new content, new tools, and new flows.

If you are already in development, we strongly recommend that you implement the standards described here as you develop, test, and release your product to customers.

If you are in the early stages of your product design and have not already selected a device or development environment, we recommend that you include security considerations in your decision. Silicon Labs provides information about the security features of our devices and development environments. Section EFR32 Series 2 Device Security Features highlights the features and their documentation references. In addition, protocol-specific security information is available in the following documents.

- Zigbee Security
- *AN1302: Bluetooth® Low Energy Application Security Design Considerations in SDK v3.x and Higher*
- *AN1329: Using Silicon Labs Secure Vault Features with OpenThread*
- Bluetooth LE Fundamentals and relevant Knowledge Base Articles (KBAs)
- *UG235.03: Architecture of the Silicon Labs Connect Stack v2.x*
- *UG435.03: Architecture of the Silicon Labs Connect Stack v3.x*

Copyright © 2025 Silicon Laboratories. All rights reserved.

Series 2 Device Security Features

Series 2 Device Security Features

Protecting IoT devices against security threats is central to a quality product. Silicon Labs offers several security options to help developers build secure devices, secure application software, and secure paths of communication to manage those devices. Silicon Labs' security offerings were significantly enhanced by the introduction of the Series 2 products that included a Secure Engine. The Secure Engine is a tamper-resistant component used to securely store sensitive data and keys, and to execute cryptographic functions and secure services.

On Series 1 devices, the security features are implemented by the TRNG (if available) and CRYPTO peripherals.

On Series 2 devices, the security features are implemented by the Secure Engine and CRYPTOACC (if available). The Secure Engine may be hardware-based, or virtual (software-based). Throughout this document, the following abbreviations are used:

- HSE: Hardware Secure Engine
- VSE: Virtual Secure Engine
- SE: Secure Engine (either HSE or VSE)

Additional security features are provided by Secure Vault. Three levels of Secure Vault feature support are available, depending on the part and SE implementation, as reflected in the following table:

Security Level (1)	SE Support	MCU	Wireless SoC (2)
Secure Vault Base (SVB)	N/A	EFM32JG1, EFM32PG1, EFM32JG12, EFM32PG12, EFM32GG11, EFM32GG12, EFM32TG11	EFR32xG1, EFR32xG12, EFR32xG13, EFR32xG14
Secure Vault Mid (SVM)	VSE (VSE-SVM)	EFM32PG22C	EFR32xG22C, EFR32xG27C (3)
Secure Vault Mid (SVM)	HSE (HSE-SVM)	-	EFR32xG21A, EFR32xG21A (Rev C), EFR32MR21A (Rev C), EFR32xG23A, EFR32xG24A, EFR32xG25A, EFR32xG28A
Secure Vault High (SVH)	HSE only (HSE-SVH)	EFM32PG23B, EFM32PG28B	EFR32xG21B, EFR32xG12B (Rev C) EFR32xG23B, EFR32xG24B, EFR32xG25B, EFR32xG28B

Notes:

1. The features of different Secure Vault levels can be found in https://www.silabs.com/security.
2. The x is a letter B, F, M, or Z.
3. Unlike other VSE-SVM parts, the EFR32xG27C device has a built-in PUF.

Secure Vault Mid consists of two core security functions:

- **Secure Boot**: Process where the initial boot phase is executed from an immutable memory (such as ROM) and where code is authenticated before being authorized for execution.
- **Secure Debug access control**: The ability to lock access to the debug ports for operational security, and to securely unlock them when access is required by an authorized entity.

Secure Vault High offers additional security options:

- **Secure Key Storage**: Protects cryptographic keys by "wrapping" or encrypting the keys using a root key known only to the HSE-SVH.
- **Anti-Tamper protection**: A configurable module to protect the device against tamper attacks.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Device authentication: Functionality that uses a secure device identity certificate along with digital signatures to verify the source or target of device communications.

A Secure Engine Manager and other tools allow users to configure and control their devices both in-house during testing and manufacturing, and after the device is in the field.

Silicon Labs strongly recommends installing the latest SE FW image on Series 2 devices and updating to the latest GSDK to mitigate security vulnerabilities. The latest SE FW image can be found in this Windows folder for GSDK v4.x:

C:\Users\<PC USER NAME>\SimplicityStudio\SDKs\gecko_sdk\util\se_release\public

Refer to *AN1222: Production Programming of Series 2 Device*s for guidance on the SE firmware upgrade procedure. The latest SE firmware shipped with Series 2 devices and modules (if available) at the time of this writing are listed in the following table:

:::custom-table{30%,}

Series 2 MCU and Wireless SoC VSE - SVM	Shipped SE Firmware Version
EFM32PG22C	1.2.12
EFR32xG22C	1.2.12
EFR32xG22C (Rev D)	1.2.14
EFR32xG27C	2.2.1
:::	

:::custom-table{30%,}

Series 2 Wireless SoC HSE - SVM	Shipped SE Firmware Version
EFR32xG21A	1.2.13
EFR32MR21A (Rev C)	1.2.16
EFR32xG21A (Rev C)	1.2.16
EFR32xG23A	2.1.7
EFR32xG24A	2.1.7
EFR32xG25A	2.2.1
EFR32xG28A	2.2.2
:::	

:::custom-table{30%,}

Series 2 MCU and Wireless SoC HSE - SVH	Shipped SE Firmware Version
EFR32xG21B	1.2.13
EFR32xG21B (Rev C)	1.2.16
EFM32PG23B	2.1.7
EFR32xG23B	2.1.7
EFR32xG24B	2.1.7
EFR32xG25B	2.2.1
EFM32PG28B	2.2.2
EFR32xG28B	2.2.2
:::	

In support of these products Silicon Labs offers whitepapers, webinars, and documentation. The following table summarizes the key security documents:

:::custom-table{30%,50%,20%}

Copyright © 2025 Silicon Laboratories. All rights reserved.

Document	Summary	Applicability
Document	Summary	Applicability
Series 2 Secure Debug	How to lock and unlock Series 2 debug access, including background information about the Secure Engine	Series 2
Series 2 Secure Boot with RTSL	Describes the secure boot process on Series 2 devices using Secure Engine.	Series 2
Anti-Tamper Protection Configuration and Use	How to program, provision, and configure the anti-tamper module	Series 2 with SVH
Authenticating Silicon Labs Devices using Device Certificates	How to authenticate a device using secure device certificates and signatures, at any time during the life of the product	Series 2 with SVH
Secure Key Storage	How to securely “wrap” keys so they can be stored in non-volatile storage	Series 2 with SVH
AN1222: Production Programming of Series 2 Devices	How to program, provision, and configure security information using Secure Engine during device production	Series 2
AN1303: Programming Series 2 Devices Using the Debug Challenge Interface (DCI) and Serial Wire Debug (SWD)	How to provision and configure Series 2 devices through the DCI and how to program their internal flash memory through the SWD	Series 2
AN1311: Integrating Crypto Functionality Using PSA Crypto Compared to Mbed TLS	How to integrate crypto functionality into applications using Silicon Labs implementation of PSA Crypto compared to Mbed TLS	Series 1 and Series 2
Series 2 TrustZone	Describes the basics of TrustZone, secure and privileged programming model, and shows how to upgrade existing application to TrustZone.	Series 2

Copyright © 2025 Silicon Laboratories. All rights reserved.

Series 2 Secure Debug

Series 2 Secure Debug

NOTE: This section replaces *AN1190: Series 2 Secure Debug*. Further updates to this application note will be provided here.

This application note describes how to lock and unlock the debug access of Series 2 devices. Many aspects of the debug access, including the secure debug unlock, are discussed. The Debug Challenge Interface (DCI) and Mailbox Interface for locking and unlocking debug access are also included.

The debug locks and unlocks for the Cortex-M33 debug interface are implemented through the Secure Engine on Series 2 devices.

Key Points

- Basic overview of the Secure Engine.
- Debug port access by Debug Challenge Interface (DCI) or Mailbox Interface.
- New locking and unlocking features for Series 2 devices.
- Examples for provisioning and Secure Debug Unlock.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Series 2 Device Security Features

Series 2 Device Security Features

Protecting IoT devices against security threats is central to a quality product. Silicon Labs offers several security options to help developers build secure devices, secure application software, and secure paths of communication to manage those devices. Silicon Labs' security offerings were significantly enhanced by the introduction of the Series 2 products that included a Secure Engine. The Secure Engine is a tamper-resistant component used to securely store sensitive data and keys and to execute cryptographic functions and secure services.

On Series 1 devices, the security features are implemented by the TRNG (if available) and CRYPTO peripherals.

On Series 2 devices, the security features are implemented by the Secure Engine and CRYPTOACC (if available). The Secure Engine may be hardware-based, or virtual (software-based). Throughout this document, the following abbreviations are used:

- HSE - Hardware Secure Engine
- VSE - Virtual Secure Engine
- SE - Secure Engine (either HSE or VSE)

Additional security features are provided by Secure Vault. Three levels of Secure Vault feature support are available, depending on the part and SE implementation, as reflected in the following table:

Level (1)	SE Support	Part (2)
Secure Vault High (SVH)	HSE only (HSE-SVH)	Refer to IoT Endpoint Security Fundamentals for details on supporting devices.
Secure Vault Mid (SVM)	HSE (HSE-SVM)	"
"	VSE (VSE-SVM)	"
Secure Vault Base (SVB)	N/A	"

Notes:

1. The features of different Secure Vault levels can be found in https://www.silabs.com/security.
2. IoT Endpoint Security Fundamentals.

Secure Vault Mid consists of two core security functions:

- Secure Boot: Process where the initial boot phase is executed from an immutable memory (such as ROM) and where code is authenticated before being authorized for execution.
- Secure Debug access control: The ability to lock access to the debug ports for operational security, and to securely unlock them when access is required by an authorized entity.

Secure Vault High offers additional security options:

- Secure Key Storage: Protects cryptographic keys by "wrapping" or encrypting the keys using a root key known only to the HSE-SVH.
- Anti-Tamper protection: A configurable module to protect the device against tamper attacks.
- Device authentication: Functionality that uses a secure device identity certificate along with digital signatures to verify the source or target of device communications.

A Secure Engine Manager and other tools allow users to configure and control their devices both in-house during testing and manufacturing, and after the device is in the field.

User Assistance

Copyright © 2025 Silicon Laboratories. All rights reserved.

In support of these products, Silicon Labs offers whitepapers, webinars, and documentation. The following table summarizes the key security documents:

Document	Summary	Applicability
Series 2 Secure Debug (this application note)	How to lock and unlock Series 2 debug access, including background information about the SE	Secure Vault Mid and High
Series 2 Secure Boot with RTSL	Describes the secure boot process on Series 2 devices using SE	Secure Vault Mid and High
Anti-Tamper Protection Configuration and Use	How to program, provision, and configure the anti-tamper module	Secure Vault High
Authenticating Silicon Labs Devices using Device Certificates	How to authenticate a device using secure device certificates and signatures, at any time during the life of the product	Secure Vault High
Secure Key Storage	How to securely 'wrap' keys so they can be stored in non-volatile storage.	Secure Vault High
AN1222: Production Programming of Series 2 Devices	How to program, provision, and configure security information using SE during device production	Secure Vault Mid and High

Key Reference

Public/Private keypairs along with other keys are used throughout Silicon Labs security implementations. Because terminology can sometimes be confusing, the following table lists the key names, their applicability, and the documentation where they are used.

Key Name	Customer Programmed	Purpose
Public Sign key (Sign Key Public)	Yes	Secure Boot binary authentication and/or OTA upgrade payload authentication
Public Command key (Command Key Public)	Yes	Secure Debug Unlock or Disable Tamper command authentication
OTA Decryption key (GBL Decryption key) aka AES-128 Key	Yes	Decrypting GBL payloads used for firmware upgrades
Attestation key aka Private Device Key	No	Device authentication for secure identity

SE Firmware

Silicon Labs strongly recommends installing the latest SE firmware on Series 2 devices to support the required security features. Refer to AN1222 for the procedure to upgrade the SE firmware and IoT Endpoint Security Fundamentals for the latest SE Firmware shipped with Series 2 devices and modules.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Introduction To Secure Debug

Introduction to Secure Debug

Debug Lock

All devices require the capability to lock out debug access to the device. This prevents attackers from using the debug interface to perform the following illegal operations:

- Reprogramming the device
- Interrogating the device
- Interfering with the operation of the device

A fairly standard practice during the board-level test in production is to program, test, and lock the parts.

Three different locks can be enabled on the Series 2 debug interface:

- Standard-debug-lock
- Permanent-debug-lock
- Secure-debug-lock

Silicon Labs provides Custom Part Manufacturing Service (CPMS) to securely configure the debug port of the chip to one of the three possible locks before the devices leave the factory.

Debug Unlock

Users need to unlock parts under a number of circumstances:

- Code development
- Field failure diagnosis
- Product field service
- Existing inventory reprogramming

Two different unlocks can run on the Series 2 debug interface:

- Standard-debug-unlock
- Secure-debug-unlock

Copyright © 2025 Silicon Laboratories. All rights reserved.

Secure Engine Subsystem

Secure Engine Subsystem

Overview

The HSE refers to a separate security co-processor that provides hardware isolation between security functions and the host processor.

The VSE refers to a collection of security functions available to the host processor in Root mode if a separate security co-processor is not provided.

The SE is used to perform a series of cryptographic operations and other secure system operations as described in the following table.

Operation	VSE-SVM	HSE-SVM	HSE-SVH	Description
Unique ID	Y	Y	Y	Software can identify every device.
Secure Boot with RTSL	Y	Y	Y	Only boot authenticated firmware.
Secure Debug	Y	Y	Y	Allow enhanced failure analysis.
Crypto Engine (1)	-	Y	Y	Up to 256-bit ciphers and elliptic curves.
TRNG (1)	-	Y	Y	Generate keys for cryptography.
DPA Countermeasures	-	Y	Y	Resist side channel attacks.
Secure Key Storage	-	-	Y	Protected by PUF technology.
Secure Key Management	-	-	Y	Isolate encrypted keys from application code.
Secure Attestation	-	-	Y	Ensure integrity and authenticity.
Anti-Tamper	-	-	Y	Detect tamper and protect keys/data.
Advanced Crypto	-	-	Y	Up to 512-bit ciphers and 521-bit elliptic curves.

Note:

1. On VSE-SVM devices, the crypto engine and TRNG (True Random Number Generator) are implemented by the CRYPTOACC (Cryptographic Accelerator) peripheral.

To start using the secure debug unlock functionality, the device needs to be provisioned. These steps include writing one-time-programmable (OTP) settings to the SE to determine which functionality is enabled, and uploading the Public Command Key to validate a secure debug attempt.

This application note describes how the different device debug locks and unlocks are implemented through the SE on Series 2 devices.

The secure debug feature is implemented by Root code executed by the HSE Core or by the Cortex-M33 operating in VSE (Root mode).

Silicon Labs strongly recommends installing the latest SE firmware on Series 2 devices to support the required security features. The latest SE firmware image (`.seu` and `.hex`) and release notes can be found in the Windows folder below.

For GSDK v3.2 and lower:

`C:\SiliconLabs\SimplicityStudio\v5\developer\sdks\gecko_sdk_suite\<GSDK VERSION>\util\se_release\public`

Copyright © 2025 Silicon Laboratories. All rights reserved.

For GSDK v4.0 and higher:

`C:\Users\<PC USER NAME>\SimplicityStudio\SDKs\gecko_sdk\util\se_release\public`

Command Interface

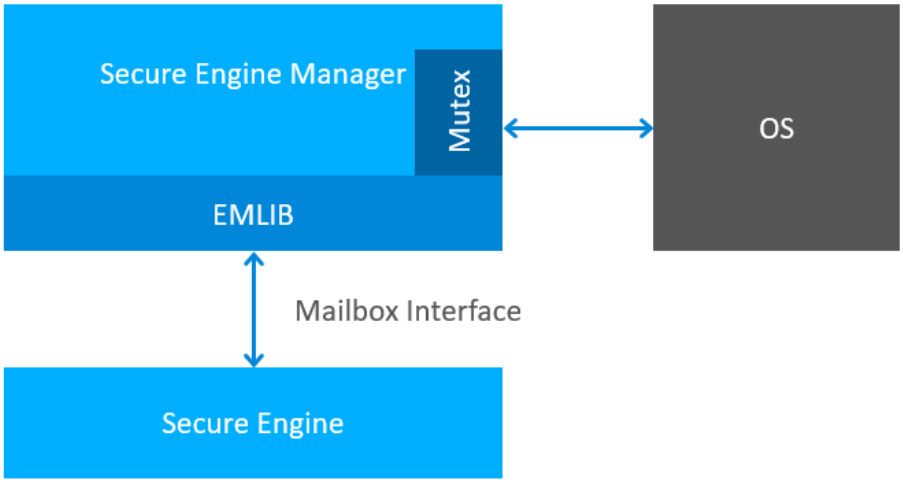
Interaction with the SE is performed over a command interface. The command interface is available through a dedicated Debug Challenge Interface (DCI) as well as through a mailbox interface from the Cortex-M33.

Some commands may not be available at all times and may not be accessible over both interfaces. The DCI typically only contains operations for setting up a new device and for locking it down (meant for production processes), while the mailbox interface also contains commands to support cryptographic operations in HSE.

Mailbox

Mailbox operations should not be performed directly, but rather should be executed through the appropriate functions in `em_se.c` of `emlib`. The `em_se.c` provides an abstraction of the mailbox interface, allowing message construction and DMA transfer setup.

On top of `emlib`, the Secure Engine Manager (SE Manager) provides an abstraction of the Secure Engine's command set. The SE Manager also provides APIs for cryptographic operations and thread synchronization. The SE Manager is available in **GSDK v3.0** or later.



> **Note**: Some functions in `em_se.c` of `emlib` are deprecated in **GSDK v3.0** and will be removed in a future version of `emlib`. All high-level functionality has been moved to the SE Manager.

Debug Challenge Interface (DCI)

The Debug Challenge Interface (DCI) is made available through commands in Simplicity Studio and Simplicity Commander. This is the easiest way to access and set up the different security options.

For more information about DCI, see AN1303: Programming Series 2 Devices using the Debug Challenge Interface (DCI) and Serial Wire Debug (SWD).

Copyright © 2025 Silicon Laboratories. All rights reserved.

Debug Lock

Debug Lock

Overview

The debug access port connected to the Series 2 device's Cortex-M33 processor can be closed by issuing commands to the SE, either from a debugger over DCI or through the mailbox interface. These three debug lock properties govern the behavior of the debug lock.

Property	Description If Set	Default Value
Debug Lock	The debug port is kept locked on boot.	False (Disabled)
Device Erase	The Erase Device command is available.	True (Enabled)
Secure Debug	Secure debug unlock is available.	False (Disabled)

The following sections describe how to interact with these properties and how to enable debug locks using the SE command interface either over DCI or the mailbox interface. The status of the debug lock can be inspected using the Read Lock Status command.

Standard Debug Unlock

The device is in standard debug unlock state if the debug lock properties are in default values.

Secure Debug	Device Erase	Debug Lock	Description
Disabled	Enabled	Disabled (Unlock)	All debug operations are allowed.

Standard Debug Lock

With the default properties in the table above, the device can be locked using the Apply Lock command. The typical flow for this configuration is simply to issue the `Apply Lock` command after the device has been programmed, either using a DCI command from the programming debugger or through the mailbox interface.

Secure Debug	Device Erase	Debug Lock	Description
Disabled	Enabled	Enabled (Standard)	The Erase Device command will wipe the main flash and RAM, and then a reset will yield an unlocked device.

The standard debug lock behaves similarly to Series 1 devices. The access port can be closed, but issuing a device erase wipes the device and opens the debug port again.

Permanent Debug Lock

The Erase Device command can be disabled, which permanently enables the debug lock. This can be done at any time by issuing the Disable Device Erase command, even after the debug lock has been enabled.

Secure Debug	Device Erase	Debug Lock	Description
Disabled	Disabled	Enabled (Permanent)	The part cannot be unlocked. Devices with Permanent Debug Lock engaged cannot be returned for failure analysis.

Secure Debug Lock

Copyright © 2025 Silicon Laboratories. All rights reserved.

For secure debug lock, the debug interface can be temporarily enabled by answering a challenge if the Secure debug property is enabled before locking.

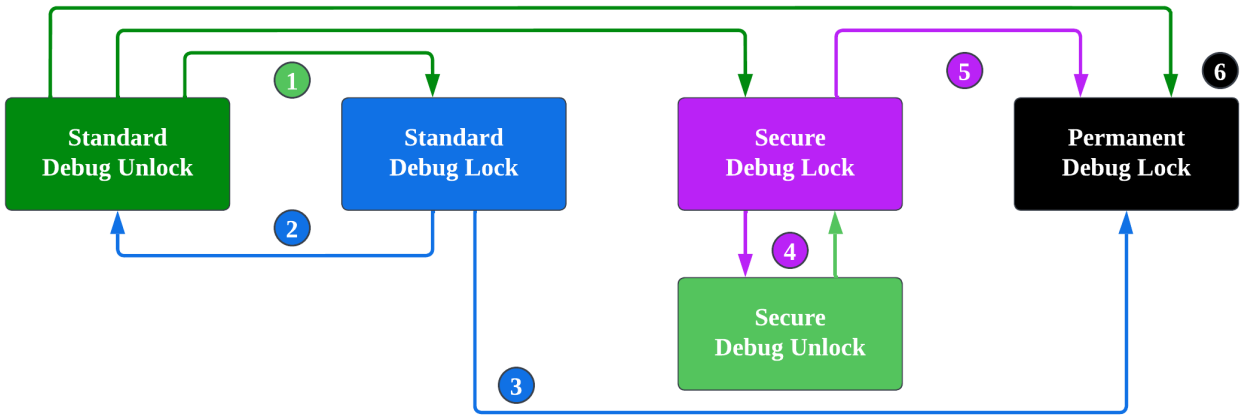
Secure Debug	Device Erase	Debug Lock	Description
Enabled (1)	Disabled (2)	Enabled (Secure)	Secure debug unlock is enabled, which makes it possible to securely open the debug lock temporarily to reprogram or debug a locked device.

Note:

1. Secure debug is enabled in two steps before the debug lock is enabled:
   a. Install the Public Command Key using Simplicity Studio or Simplicity Commander or directly through the SE Manager API.
   b. Enable secure debug using Simplicity Studio or Simplicity Commander or directly through the SE Manager API.
2. Disable the device erase using Simplicity Studio or Simplicity Commander or directly through the SE Manager API. This is an **IRREVERSIBLE** action and should be disabled **AFTER** the secure debug is enabled.

Debug Lock State Transition

The following figure describes the transitions between different debug lock states.



1. Standard debug unlock can transit to any debug lock state.
2. Standard debug lock can revert to standard debug unlock via an Erase Device command (erase the main flash and RAM). After the device is reset, debug port remains unlocked until it is explicitly locked again.
3. Standard debug lock can transit to permanent debug lock by disabling the Device Erase property but cannot transit to secure debug lock.
4. Secure debug lock can use Debug Unlock Token to temporary transit to secure debug unlock, which does not erase the main flash and RAM but enables debug operations. The device reverts to the secure debug lock through a power-on or pin reset.
5. Secure debug lock can transit to permanent debug lock by disabling the Secure Debug property but cannot transit to standard debug lock.
6. Permanent debug lock is a terminal state and cannot transit to any debug lock or unlock state.

Debug Lock Command Reference

The commands for debug lock are described in the following table.

DCI Command (1)	Mailbox (SE Manager) API (2)	Description	Availability
Apply Lock	sl_se_apply_debug_lock	Enables the debug lock for the part.	While debug is unlocked.

Copyright © 2025 Silicon Laboratories. All rights reserved.

DCI Command (1)	Mailbox (SE Manager) API (2)	Description	Availability
Read Lock Status	sl_se_get_debug_lock_status	Returns the current debug lock status and configuration.	Always.
Disable Device Erase	sl_se_disable_device_erase	Disables the Erase Device command. This command does not lock the debug interface to the part, but it is an IRREVERSIBLE action for the part.	Always.
Disable Secure Debug	sl_se_disable_secure_debug	Disables the secure debug functionality that can be used to open a locked debug port.	While secure debug is enabled.
Enable Secure Debug	sl_se_enable_secure_debug	Enables the secure debug functionality that can be used to open a locked debug port.	While debug is unlocked and Public Command Key is uploaded.
Set debug options	sl_se_set_debug_options	Configures the TrustZone access permissions of the debug interface. (3)	While debug is unlocked.
Init Pub Key	sl_se_init_otp_key	Used to provision a single public key during device initialization. The public key cannot be changed once written, and the command will be unavailable for that key.	Available once for each key.
Read Pub Key	sl_se_read_pubkey	Reads the stored public key.	Always.
Get Challenge	sl_se_roll_challenge	Used to roll the current challenge value (16 bytes) to revoke secure debug access. (4)	While Public Command Key is uploaded.

Notes:

1. Performing these commands over DCI is implemented in Simplicity Studio and Simplicity Commander.
2. The `sl_se_apply_debug_lock`, `sl_se_get_debug_lock_status`, `sl_se_init_otp_key`, and `sl_se_read_pubkey` are available on all Series 2 devices. Other APIs are only available on HSE devices. The SE Manager API document can be found at https://docs.silabs.com/gecko-platform/latest/service/api/group-sl-se-manager.
3. For more information about debug options, see TrustZone Debug Authentication.
4. A new challenge will only be generated if the current one has been successfully used at least once.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Debug Unlock

Debug Unlock

Overview

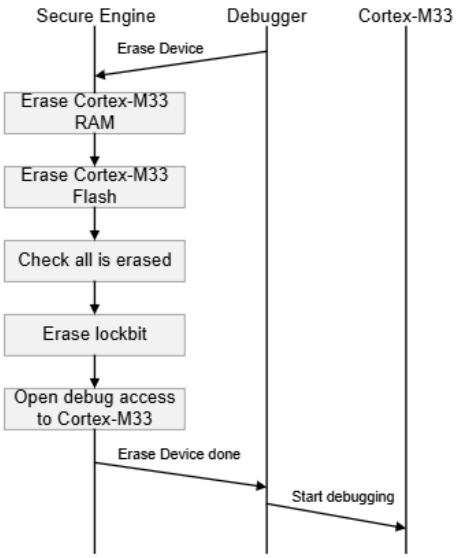
The debug access port connected to the Series 2 device's Cortex-M33 processor can be opened by issuing commands to the SE, either from a debugger over DCI or through the mailbox interface.

New on the Series 2 devices is the addition of secure debug unlock functionality. When enabled, it is possible to request a challenge from the device and, by answering the challenge, disable the debug lock until the next power-on or pin reset.

The status of the debug lock can be inspected using the Read Lock Status command.

Standard Debug Unlock

With the properties of the standard debug lock or secure debug lock with Device Erase enabled, the device can be returned to the standard debug unlock state using the Erase Device command. This command will wipe the main flash and RAM and verify they are empty before opening the debug lock. It will not wipe user data and provisioned SE settings.



Secure Debug Unlock

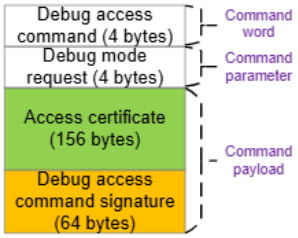
In a secure debug unlock scenario, the customer, who has control over the Private Command Key for a SE, has programmed a Public Command Key into the device. The Public Command Key is used to verify the signature on a certificate, telling the SE what authorization has been given by the owner of the key (customer) to the one issuing the command (customer or delegate). Authorization can be granted, for example, to unlock only the debug port on the Cortex-M33, or to restore only specific tamper signals on HSE-SVH devices.

Copyright © 2025 Silicon Laboratories. All rights reserved.

This mode is particularly useful in failure analysis scenarios because it allows devices to be unlocked without losing flash and RAM contents.

Debug Unlock Token

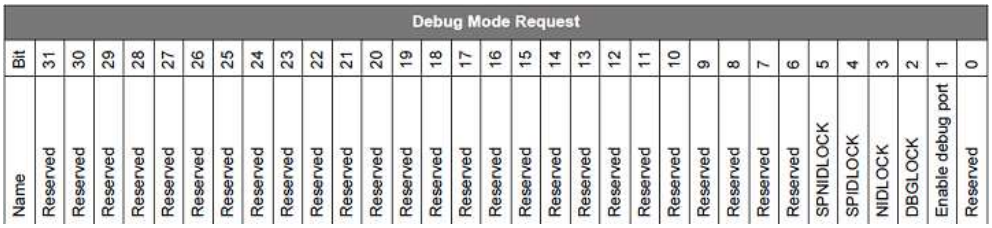
The elements of the Debug Unlock Token are described in the following figures and table.



Element	Value	Description
Debug access command	0xfd010001	The command word of the Debug Unlock Token.
Debug mode request	Device-dependent	The command parameter of the debug access command.
Access certificate (1)	Device-dependent	See section Access Certificate.
Debug access command signature (1)	Device-dependent	See section Challenge Response.

Note:

1. The debug access command payload consists of an access certificate and a debug access command signature.



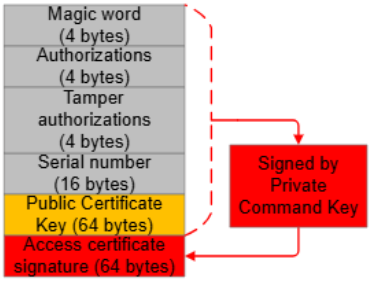
Notes:

- Enable debug port - Debug port enabled if set.
- DBGLOCK (Non-secure, Invasive debug lock) - The Invasive debug features for the Non-secure state are unlocked if set.
- NIDLOCK (Non-secure, Non-invasive debug lock) - The Non-invasive debug features for the Non-secure state are unlocked if set.
- SPIDLOCK (Secure, Invasive debug lock) - The Invasive debug features for the Secure state are unlocked if set.
- SPNIDLOCK (Secure, Non-Invasive debug lock) - The Non-invasive debug features for the Secure state are unlocked if set.
- All reserved bits should be 0, and bit 1 must be 1 to access the debug port.
- For the TrustZone-unaware debugging, bits 2 to 5 are irrelevant, so bits 1 to 5 are usually set (`0x0000003e`) to match with the Authorizations in the access certificate.
- For the TrustZone-aware debugging, bits 2 to 5 are relevant. Refer to Trust Zone Debug Authentication for details about these debug options.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Access Certificate

The elements of the access certificate are described in the following figures and table.



Element	Value	Description
Magic word	0xe5ecce01	A constant value used to identify the access certificate.
Authorizations	0x0000003e (1)	A value used to authorize which bit in the debug mode request can be enabled for secure debug.
Tamper Authorizations	0x00000000 or 0xffffffb6 (2)	A value used to authorize which bit in the tamper disable mask can be set to disable the tamper response.
Serial number	Device-dependent	A number used to compare against the on-chip serial number for secure debug or tamper disable.
Public Certificate Key (3)	Device-dependent	The public key corresponding to the Private Certificate Key (3) used to generate the signature (ECDSA-P256-SHA256) in a challenge response.
Access certificate signature	Device-dependent	All the content above is signed (ECDSA-P256-SHA256) by the Private Command Key corresponding to the Public Command Key in the SE OTP.

Notes:

1. This value allows all debug options to be reset for secure debug.
2. Value that sets available bits in the tamper disable mask for tamper disable (HSE-SVH device only).
3. The Private/Public Certificate Key is a randomly generated key pair. It can be ephemeral or retainable.

The Private Certificate Key can be used repeatedly to generate the signature in a challenge response on one device until the Private/Public Certificate Key pair is discarded. This can reduce the frequency of access to the Private Command Key, allowing more restrictive access control on that key.

For more information about tamper disable, see Anti-Tamper Protection Configuration and Use.

Copyright © 2025 Silicon Laboratories. All rights reserved.

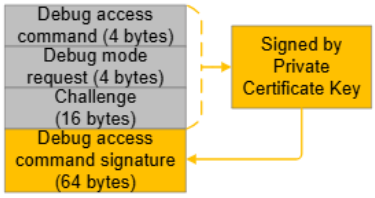
Authorizations																															
Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	SPNIDLOCK request mask	SPIDLOCK request mask	NIDLOCK request mask	DBGLOCK request mask	Enable debug port request mask	Reserved

Notes:

- Set the bit to enable the corresponding bit in the debug mode request.
- The Debug Unlock Token will reset the corresponding debug option if the same bit is set in Debug mode request and Authorizations.

Challenge Response

The elements of the challenge response are described in the following figure and table.



Element	Value	Description
Debug access command	0xfd010001	The command word of the Debug Unlock Token.
Debug mode request	Device-dependent	The command parameter of the debug access command.
Challenge	Device-dependent (1)	A random value generated by the SE.
Debug access command signature	Device-dependent (2)	All the content above is signed (ECDSA-P256-SHA256) by the Private Certificate Key corresponding to the Public Certificate Key in the access certificate.

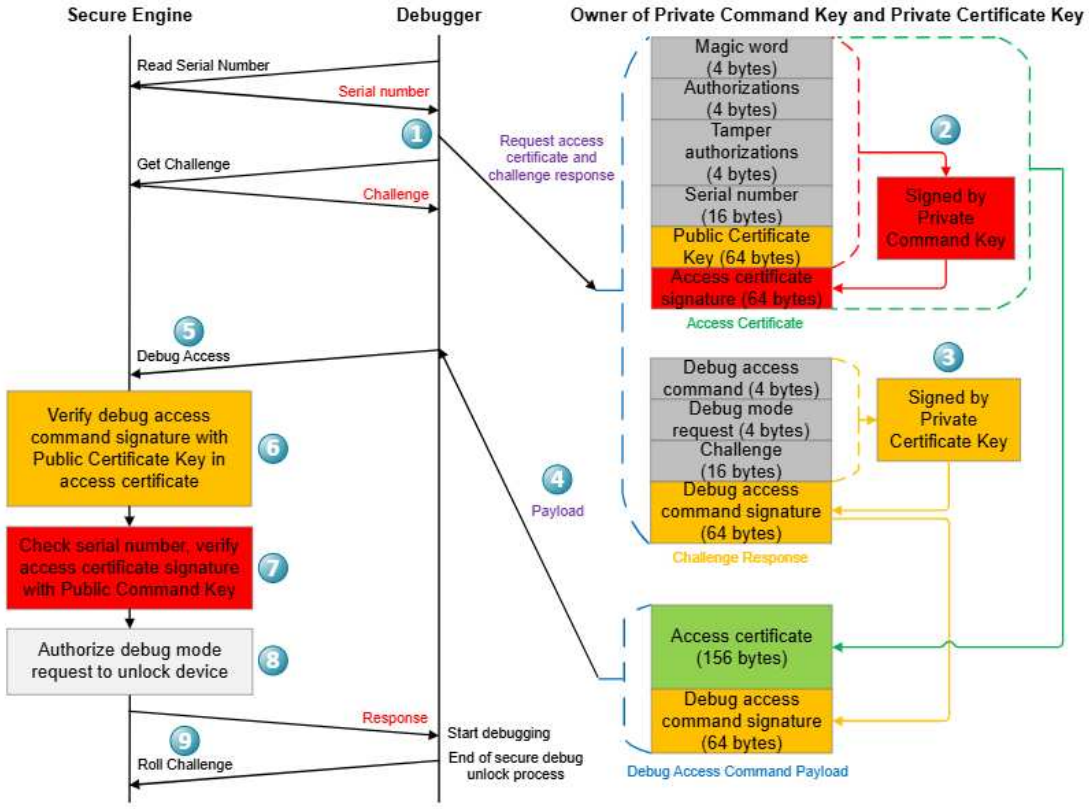
Notes:

1. The challenge remains unchanged until it is updated to a new random value by rolling the challenge. The Private Certificate Key can be reused for signing when the device challenge is refreshed.
2. This signature is the final argument of the Debug Unlock Token.

Debug Access Flow

Copyright © 2025 Silicon Laboratories. All rights reserved.

The debug access flow is described in the following figure.



1. Get the serial number and challenge from the SE.
2. Generate the access certificate with the device serial number.
3. Generate the challenge response with device challenge.
4. Generate the debug access command payload with access certificate and debug access command signature.
5. Send the Debug Unlock Token to the SE.
6. Verify the debug access command signature using the Public Certificate Key in the access certificate.
7. Verify the serial number and the access certificate signature using the on-chip serial number and Public Command Key in the SE OTP.
8. Authorize the debug mode request to reset the debug options until the next power-on or pin reset.
9. Roll the challenge to invalidate the current Debug Unlock Token.

TrustZone Debug Authentication

The debug and trace support in the Cortex-M33 devices are based on the CoreSight architecture, which can be classified into Invasive and Non-invasive debugging features as described in the following table.

Classification	Debug and Trace Features	Description
Invasive	Core debug (e.g., single stepping), Breakpoints, Data watchpoints, Halt mode debugging	These features halt the Cortex-M33 core and change the program execution flow.
Non-invasive	Embedded Trace Macrocell (ETM), Micro Trace Buffer (MTB), Data trace, Instrumentation Trace Macrocell (ITM), Profiling	These features have a minor or no impact on the program execution flow.

The separation of Invasive and Non-invasive debug and trace operations in CoreSight architecture can apply to TrustZone debug authentication, which defines the permission levels of the debug and trace features on Secure and Non-secure worlds.

The table below describes four debug options in SE to support TrustZone debug authentication. It is possible to restrict the TrustZone access permissions of the debug interface by setting one or more of the following options.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Debug Option	Description
DBGLOCK	Non-secure, Invasive debug lock. If this bit is set, the Invasive debug features for the Non-secure state are locked.
NIDLOCK	Non-secure, Non-invasive debug lock. If this bit is set, the Non-invasive debug features for the Non-secure state are locked.
SPIDLOCK	Secure, Invasive debug lock. If this bit is set, the Invasive debug features for the Secure state are locked.
SPNIDLOCK	Secure, Non-invasive debug lock. If this bit is set, the Non-invasive debug features for the Secure state are locked.

Notes:

- Use Simplicity Commander or the SE Manager API to set the debug options.
- The state of the debug options is stored permanently in SE and can only be reset to the default value (`0000`) through the Erase Device command (if enabled).
- A secure debug lock device (Device Erase was disabled) can only use the Debug Unlock Token to temporarily unlock (reset) the debug options to debug the Secure and Non-secure applications.

The following conditions are recommended (1, 2, and 3) or mandatory (4) when setting up the debug options for secure debug unlock.

1. If SPIDLOCK is unlocked, then DBGLOCK should also be unlocked.
2. If SPNIDLOCK is unlocked, then NIDLOCK should also be unlocked.
3. If DBGLOCK is unlocked, the NIDLOCK should also be unlocked.
4. If SPIDLOCK is unlocked, then SPNIDLOCK is automatically unlocked.

The following table lists the recommended combinations of debug options.

SPNIDLOCK	SPIDLOCK	NIDLOCK	DBGLOCK	Description
0	0	0	0	Allows all debug and trace features for both the Secure and the Non-secure world (default setting).
0	1	0	0	Only allows a Non-invasive debug in the Secure world. Allows both Invasive and Non-invasive debugs in the Non-secure world.
0	1	0	1	Only allows a Non-invasive debug in the Secure and the Non-secure world.
1	1	0	0	Only allows debug and trace features in the Non-secure world.
1	1	0	1	Only allows a Non-invasive debug in the Non-secure world.
1	1	1	1	All debug and trace features are disabled.

Notes:

- Trace Point Interface Unit (TPIU) registers' access fault will occur and lock the processor in a security assertion if both NIDLOCK and DBGLOCK in debug option are set (`xx11`). The device will be unrecoverable if it is in the permanent debug lock state.
- The workaround is to avoid using the `xx11` debug option or avoid accessing the TPIU registers and upgrade to SE firmware **≥ v1.2.14** (xG21 and xG22) or **≥ v2.2.1** (other Series 2 devices) so that the debug options cannot be modified after the device is locked.

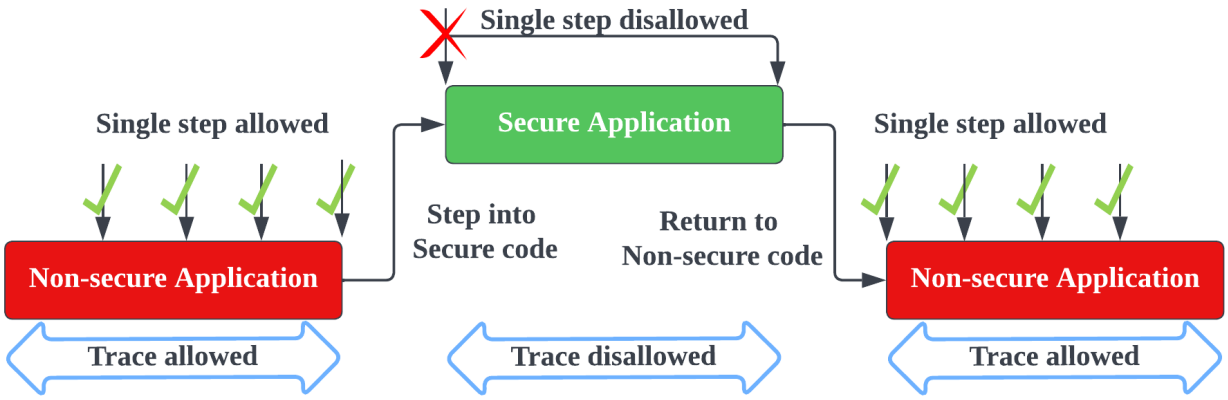
The highly recommended setting of debug options is to allow debugging in the Non-secure world while, at the same time, disabling debugging for the Secure world (`1100`).

- Secure memories (flash and RAM) are not accessible by the debugger.
- All debug access is blocked from accessing Secure addresses.
-

Copyright © 2025 Silicon Laboratories. All rights reserved.

The debugger will ignore the vector-catch events generated by the Secure exceptions.
- Trace sources (e.g., ETM) will stop generating instruction/data trace packets when the Cortex-M33 is in a Secure state.
- The debugger can neither halt a Secure application (e.g., breakpoint) nor single step into it.
- The Cortex-M33 will not stop when stepping into the Secure application until it returns to the Non-secure state.

The figure below describes the debug scenario of debug options with `1100` configuration.



The following examples describe the relationship between **debug options** and **debug mode request** when performing a secure debug unlock on Series 2 devices.

Example 1: All debug and trace features for both the Secure and the Non-secure world are allowed (`0000`)

Debug Options	Authorizations	Debug Mode Request	Debug options after Secure Debug Unlock	Description
0000	00|1111|10	00|xxxx|10	0000	No action

Example 2: Only debug and trace features in the Non-secure world are allowed (`1100`)

Debug Options	Authorizations	Debug Mode Request	Debug options after Secure Debug Unlock	Description
1100	00|1111|10	00|00xx|10	1100	No action
1100	00|1111|10	00|10xx|10	0100	Unlock SPNIDLOCK
1100	00|1111|10	00|01xx|10 or 00|11xx|10	0000(reset SPIDLOCK will automatically unlock SPNIDLOCK)	Unlock SPNIDLOCK and SPIDLOCK

Notes:

- The bit order of debug options are SPNIDLOCK (MSB), SPIDLOCK, NIDLOCK, and DBGLOCK (LSB).
- Debug options: 0 = Unlocked, 1 = Locked
- Authorizations in the access certificate: 0 = Disable, 1 = Enable
- The authorizations in the access certificate are usually set to `00|1111|10` (`0x3e`), so the corresponding debug options (bits 2 to 5) can be reset (unlocked) by debug mode request during secure debug unlock.
- Debug mode request (bits 2 to 5) in the Debug Unlock Token:
  -0 = No action on the corresponding debug option if it was locked (i.e., 1)
  -1 = Reset (unlock) the corresponding debug option from 1 to 0 if it was locked (i.e., 1)
  -x = No action (either 0 or 1) on the corresponding debug option if it was unlocked (i.e., 0)
- Debug options return to the original state after power-on or pin reset.

Debug Unlock Command Reference

The commands for debug unlock are described in the following table.

Copyright © 2025 Silicon Laboratories. All rights reserved.

DCI Command (1)	Mailbox (SE Manager) API (2)	Description	Availability
Erase Device	sl_se_erase_device	Performs a device mass erase and resets the debug configuration to its initial unlocked state.	While Device Erase is enabled.
Read Serial Number	sl_se_get_serialnumber	Reads out the serial number (16 bytes) of the Series 2 device.	Always.
Get Challenge	sl_se_get_challenge	Reads out the current challenge value (16 bytes) for Secure debug unlock.	While Public Command Key is uploaded.
Debug Access	sl_se_open_debug	Opens the secure debug access of the Cortex-M33.	Only when Secure Debug is enabled.

Notes:

1. Performing these commands over DCI is implemented in Simplicity Studio and Simplicity Commander.
2. These APIs are only available on HSE devices. The SE Manager API document can be found at https://docs.silabs.com/gecko-platform/latest/service/api/group-sl-se-manager.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Examples

Examples

Overview

The examples for Series 2 debug lock and debug unlock are described in the following table.

Example	Device (Radio Board)	SE Firmware	Tool
Standard debug lock	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.14	SE Manager
Standard debug lock and unlock	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.14	Simplicity Commander
"	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Studio 5
Provision Public Command Key	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.14	SE Manager
Secure debug lock	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.14	SE Manager
Provision Public Command Key and secure debug lock	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.14	Simplicity Commander
"	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Studio 5
Secure debug unlock and roll challenge	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.14	SE Manager
"	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.14	Simplicity Commander
"	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Studio 5
Secure debug unlock	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	IAR v8.50.9
Permanent debug lock	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.14	Simplicity Commander
"	EFR32MG21A010F1024IM32 (BRD4181A)	Version 1.2.9	Simplicity Studio 5

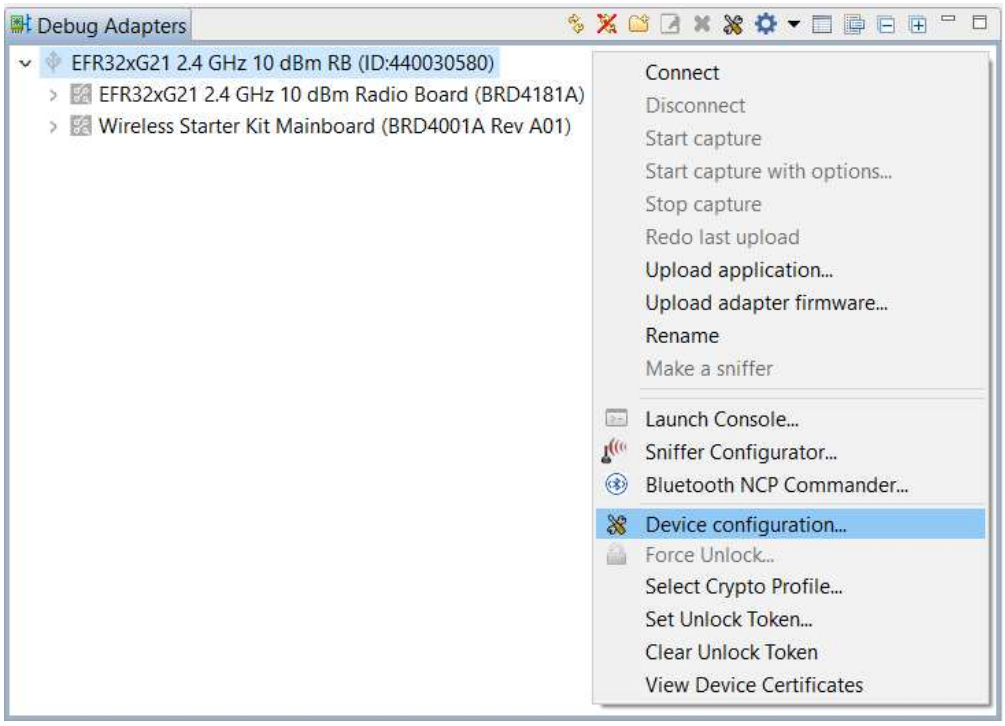
> **Note**: Unless specified in the example, these examples can be applied to other Series 2 devices.

Using Simplicity Studio

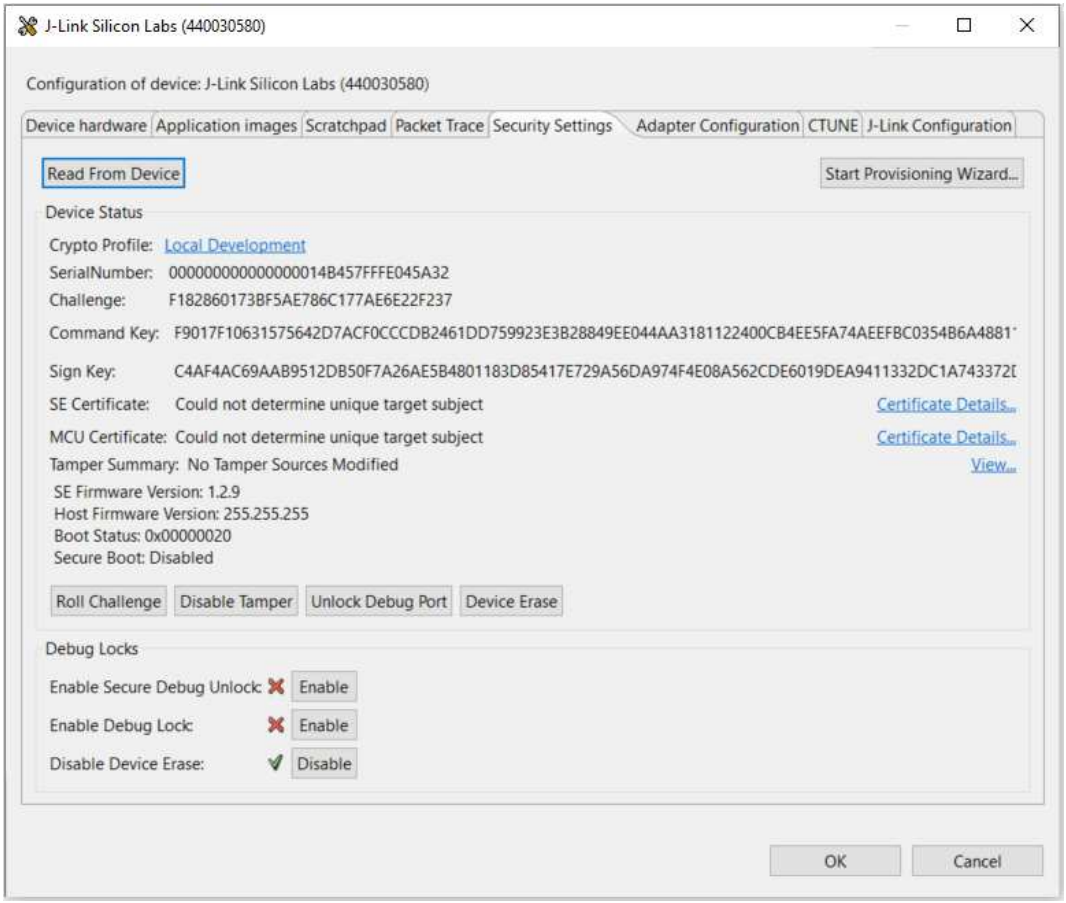
The security operations are performed in the Security Settings of Simplicity Studio. This application note uses Simplicity Studio v5.2.1.1. The procedures and pictures may be different for the other versions of Simplicity Studio 5.

1. Right-click the selected debug adapter **RB (ID:J-Link serial number)** to display the context menu.

Copyright © 2025 Silicon Laboratories. All rights reserved.



2. Click **Device configuration...** to open the **Configuration of device: J-Link Silicon Labs (serial number)** dialog box. Click the **Security Settings** tab to get the selected device configuration.



Copyright © 2025 Silicon Laboratories. All rights reserved.

Using Simplicity Commander

1. This application note uses Simplicity Commander v1.14.2. The procedures and console output may be different for the other versions of Simplicity Commander. The latest version of Simplicity Commander can be downloaded from https://www.silabs.com/developers/mcu-programming-options.

```
commander --version
```

```
Simplicity Commander 1v14p2b1232
JLink DLL version: 7.70d
Qt 5.12.10 Copyright (C) 2017 The Qt Company Ltd.
EMDLL Version: 0v18p7b669
mbed TLS version: 2.16.6
Emulator found with SN=440048205 USBAddr=0
DONE
```

2. The Simplicity Commander's Command Line Interface (CLI) is invoked by `commander.exe` in the Simplicity Commander folder. The location for Simplicity Studio 5 in Windows is `C:\SiliconLabs\SimplicityStudio\v5\developer\adapter_packs\commander`. For ease of use, it is highly recommended to add the path of `commander.exe` to the system PATH in Windows.
3. If more than one Wireless Starter Kit (WSTK) is connected via USB, the target WSTK must be specified using the `--serialno \<J-Link serial number>` option.
4. If the WSTK is in debug mode OUT, the target device must be specified using the `--device \<device name>` option.

For more information about Simplicity Commander, see UG162: Simplicity Commander Reference Guide.

Using External Tools

1. The secure debug unlock example uses the **OpenSSL** to sign the access certificate and challenge response. The Windows version of OpenSSL can be downloaded from https://slproweb.com/products/Win32OpenSSL.html. This application note uses OpenSSL Version 1.1.1h (Win64).

```
openssl version
```

```
OpenSSL 1.1.1h  22 Sep 2020
```

   The OpenSSL's Command Line Interface (CLI) is invoked by `openssl.exe` in the OpenSSL folder. The location in Windows (Win64) is `C:\Program Files\OpenSSL-Win64\bin`. For ease of use, it is highly recommended to add the path of `openssl.exe` to the system `PATH` in Windows.
2. The secure debug unlock example uses the free **Hex Editor Neo** to edit the binary files generated by Simplicity Commander. The Windows version of Hex Editor Neo can be downloaded from https://www.hhdsoftware.com/free-hex-editor.

Using Platform Examples

Simplicity Studio 5 includes the SE Manager platform examples for debug lock, key provisioning, and secure debug unlock. This application note uses platform examples of GSDK v4.2.1. The console output may be different on other versions of GSDK.

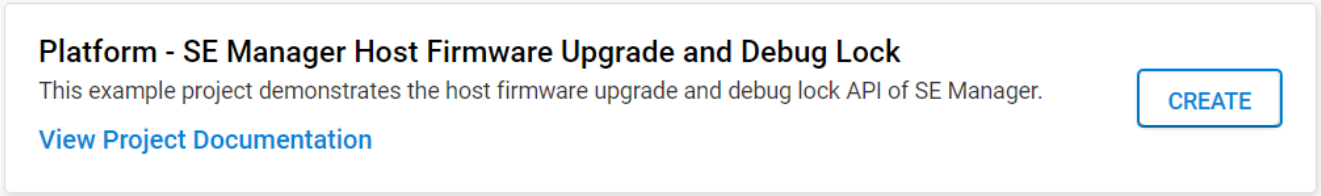
Refer to the corresponding `readme` file for details about each SE Manager platform example. This file also includes the procedures to create the project and run the example.

Standard Debug Lock and Unlock

SE Manager - Debug Lock Platform Example

Click `the View Project Documentation` link to open the `readme` file.

Copyright © 2025 Silicon Laboratories. All rights reserved.



1. Press `SPACE` then `ENTER` to select the debug lock operation.

```
SE Manager Host Firmware Upgrade and Debug Lock Example - Core running at 38000 kHz.
 . SE manager initialization... SL_STATUS_OK (cycles: 10 time: 0 us)
 . Current selection is HOST FIRMWARE UPGRADE.
 + Press SPACE to select HOST FIRMWARE UPGRADE or DEBUG LOCK, press ENTER to run.
 + Current selection is DEBUG LOCK.
 . Get debug lock status... SL_STATUS_OK (cycles: 8788 time: 231 us)
 + Debug lock: Disabled
 + Press ENTER to apply debug lock or press SPACE to exit.
```

2. Press `ENTER` again to lock the device.

```
 . Apply the debug lock...  SL_STATUS_OK (cycles: 52585 time: 1383 us)
 + Get debug lock status... SL_STATUS_OK (cycles: 8769 time: 230 us)
 + Debug lock: Enabled
 . Current selection is DEBUG LOCK.
 + Press SPACE to select HOST FIRMWARE UPGRADE or DEBUG LOCK, press ENTER to run.
```

Simplicity Commander

1. Run the `security status` command to get the selected device configuration.

```
commander security status --device EFR32MG21A010F1024 --serialno 440048205
```

```
SE Firmware version : 1.2.14
Serial number       : 00000000000000014b457fffe045a93
Debug lock          : Disabled
Device erase        : Enabled
Secure debug unlock : Disabled
Tamper status       : OK
Secure boot         : Disabled
Boot status         : 0x20 - OK
DONE
```

2. Run the `security lock` command to lock the selected device.

```
commander security lock --device EFR32MG21A010F1024 --serialno 440048205
```

```
WARNING: Secure debug unlock is disabled. Only way to regain debug access is to run a device erase.
Device is now locked.
DONE
```

3. Run the `security status` command again to check the device configuration.

```
commander security status --device EFR32MG21A010F1024 --serialno 440048205
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
SE Firmware version : 1.2.14
Serial number       : 00000000000000014b457fffe045a93
Debug lock          : Enabled
Device erase        : Enabled
Secure debug unlock : Disabled
Tamper status       : OK
Secure boot         : Disabled
Boot status         : 0x20 - OK
DONE
```

4. Run the `security erasedevice` command to unlock the selected device.

```
commander security erasedevice --device EFR32MG21A010F1024 --serialno 440048205
```

```
Successfully erased device
DONE
```

> **Note**: Issue a power-on or pin reset to complete the unlock process.

5. Run the `security status` command again to check the device configuration.

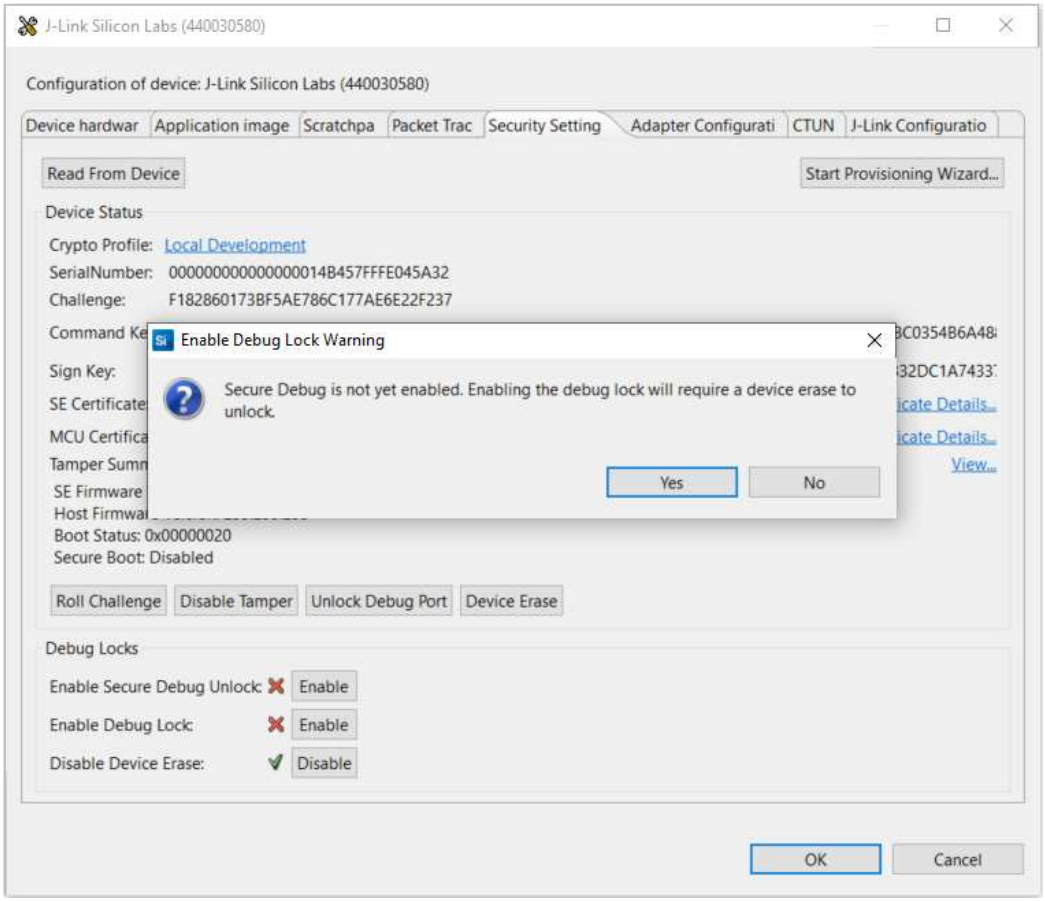
```
commander security status --device EFR32MG21A010F1024 --serialno 440048205
```

```
SE Firmware version : 1.2.14
Serial number       : 00000000000000014b457fffe045a93
Debug lock          : Disabled
Device erase        : Enabled
Secure debug unlock : Disabled
Tamper status       : OK
Secure boot         : Disabled
Boot status         : 0x20 - OK
DONE
```

Simplicity Studio

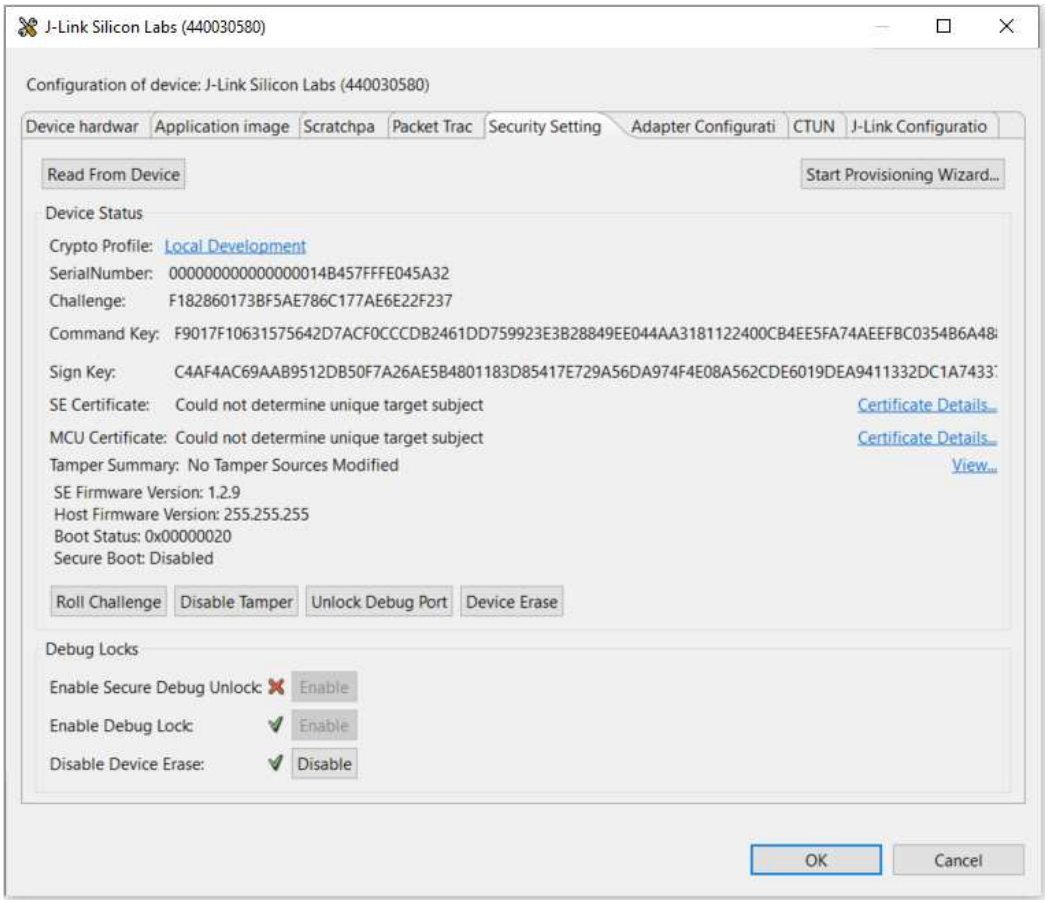
1. Open the **Security Settings** of the selected device as described in Using Simplicity Studio.
2. Click **[Enable]** next to **Enable Debug Lock**: to lock the device. The following **Enable Debug Lock Warning** is displayed. Click **[Yes]** to confirm. This configures standard debug lock.

Copyright © 2025 Silicon Laboratories. All rights reserved.



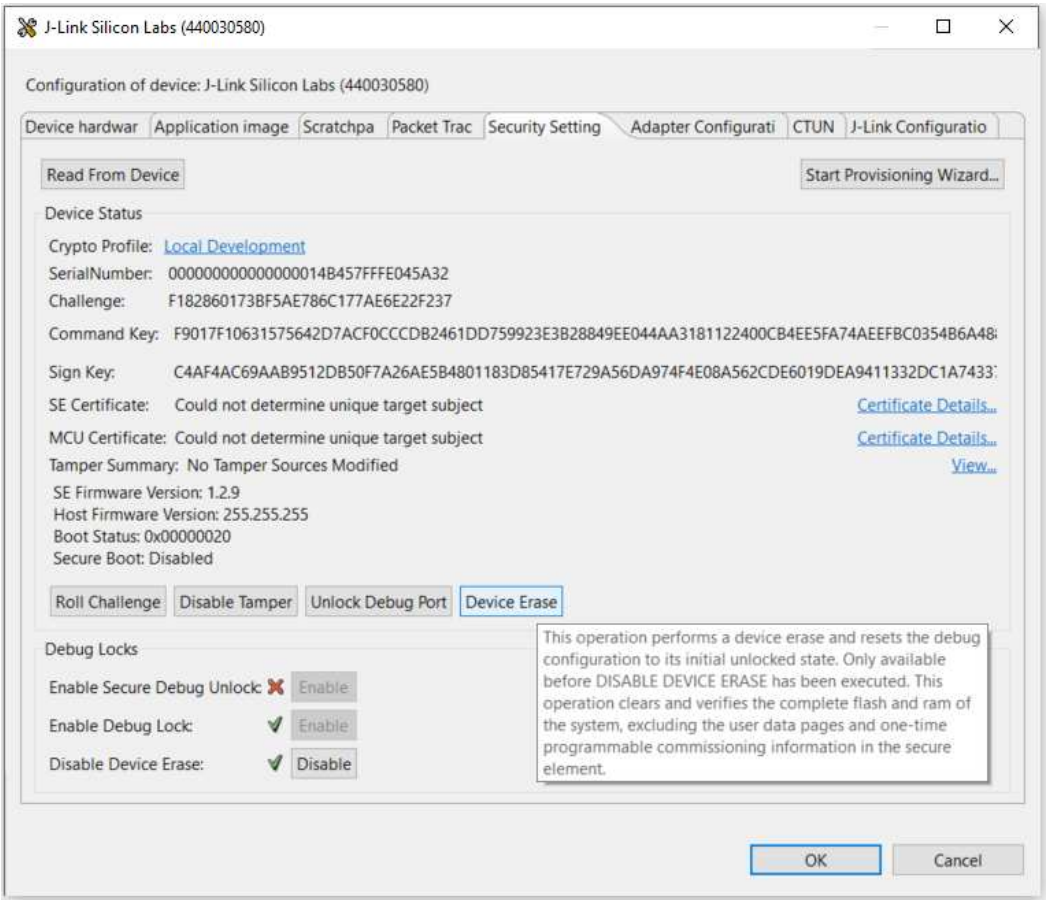
The **[Enable]** controls next to **Enable Secure Debug Unlock**: and **Enable Debug Lock**: are grayed out after standard debug lock is enabled.

Copyright © 2025 Silicon Laboratories. All rights reserved.



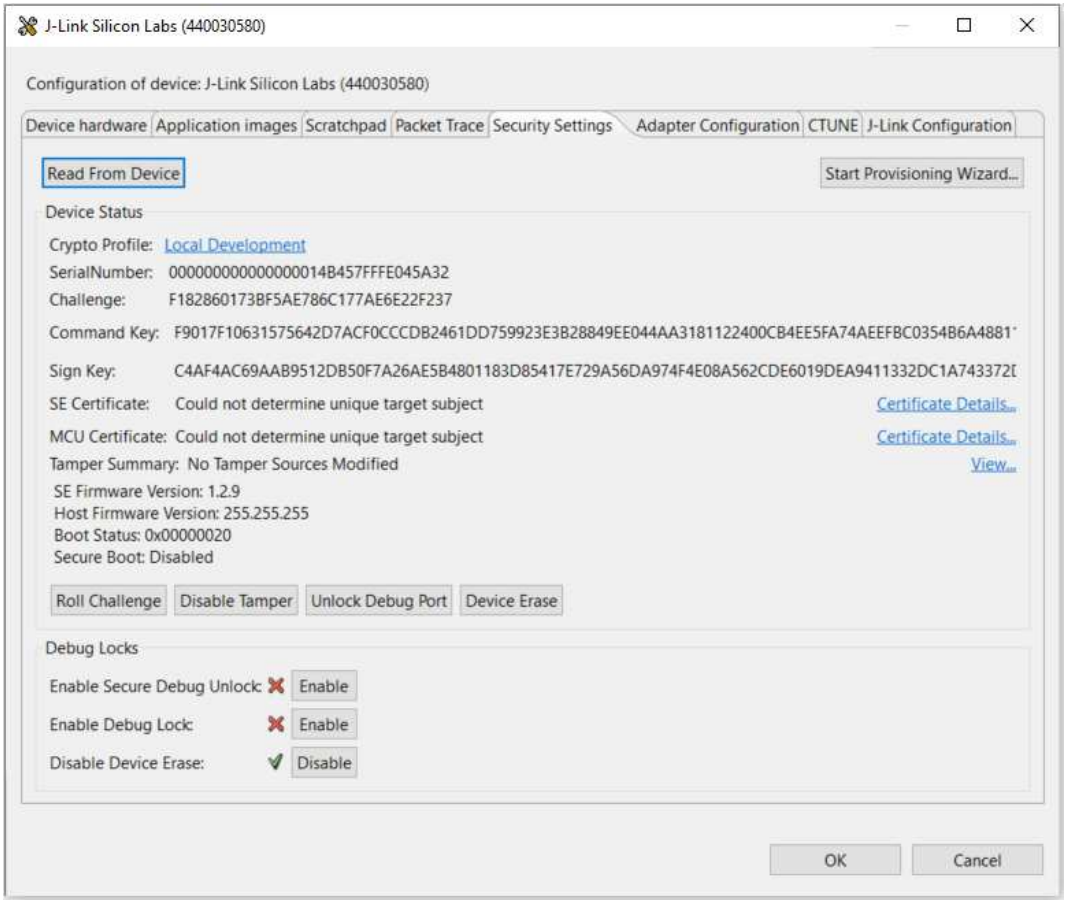
3. Click **[Device Erase]** to unlock the device.

Copyright © 2025 Silicon Laboratories. All rights reserved.



4. The device will return to the unlock state. Click **[OK]** to exit.

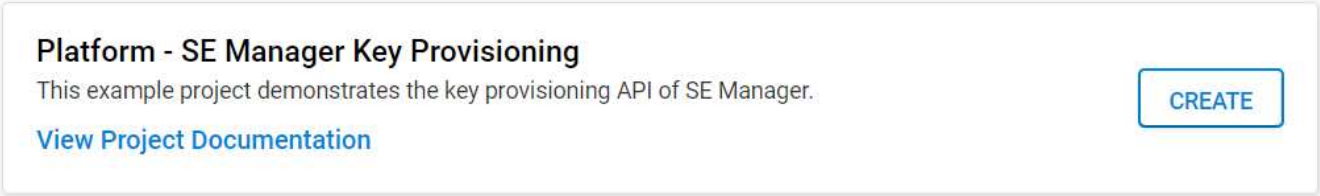
Copyright © 2025 Silicon Laboratories. All rights reserved.



Provision Public Command Key and Secure Debug Lock

SE Manager - Key Provisioning Platform Example

Click the `View Project Documentation` link to open the `readme` file.



1. Press `SPACE` to skip the programming of AES-128 key.

```
SE Manager Key Provisioning Example - Core running at 38000 kHz.
 . SE manager initialization... SL_STATUS_OK (cycles: 9 time: 0 us)
 . Get current SE firmware version... SL_STATUS_OK (cycles: 3578 time: 94 us)
 + Current SE firmware version (MSB..LSB): 00010209
 . Read SE OTP configuration... SL_STATUS_COMMAND_IS_INVALID (cycles: 3908 time: 102 us)
 . Press ENTER to program 128-bit AES key in SE OTP or press SPACE to skip.
 . Encrypt 16 bytes plaintext with 128-bit AES OTP key... SL_STATUS_FAIL (cycles: 4627 time: 121 us)
 . Press ENTER to program public sign key in SE OTP or press SPACE to skip.
```

2. Press `SPACE` to skip the programming of Public Sign Key.

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
. Get public sign key... SL_STATUS_FAIL (cycles: 4144 time: 109 us)
. Press ENTER to program public command key in SE OTP or press SPACE to skip.
```

3. Press `ENTER` to program the default Public Command Key in flash to the SE OTP.

```
+ Warning: The public command key in SE OTP cannot be changed once written!
+ Press ENTER to confirm or press SPACE to skip if you are not sure.
```

4. Press `ENTER` to confirm the operation.

```
. Initialize public command key... SL_STATUS_OK (cycles: 56052 time: 1475 us)
. Get public command key... SL_STATUS_OK (cycles: 7135 time: 187 us)
+ The public command key (64 bytes):
  B1 BC 6F 6F A5 66 40 ED 52 2B 2E E0 F5 B3 CF 7E
  5D 48 F6 0B E8 14 8F 0D C0 84 40 F0 A4 E1 DC A4
  7C 04 11 9E D6 A1 BE 31 B7 70 7E 5F 9D 00 1A 65
  9A 05 10 03 E9 5E 1B 93 6F 05 C3 7E A7 93 AD 63
. Press ENTER to initialize SE OTP for secure boot configuration or press SPACE to skip.
```

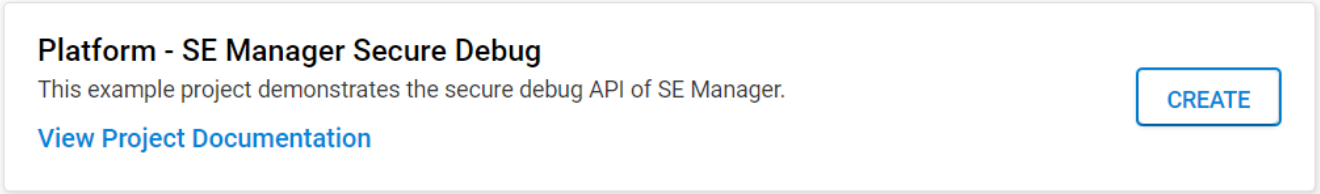
5. Press `SPACE` to skip the secure boot configuration.

```
. SE manager deinitialization... SL_STATUS_OK (cycles: 7 time: 0 us)
```

SE Manager - Secure Debug Platform Example

Click the `View Project Documentation` link to open the `readme` file.

> **Note**: The secure debug platform example can only run on the HSE device.



1. Use a standard debug unlock device with matched Public Command Key.

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
SE Manager Secure Debug Example - Core running at 38000 kHz.
 . SE manager initialization... SL_STATUS_OK (cycles: 9 time: 0 us)
 . Get SE status... SL_STATUS_OK (cycles: 8496 time: 223 us)
 + The SE firmware version (MSB..LSB): 0001020E
 + Debug lock: Disabled
 + Device Erase: Enabled
 + Secure debug: Disabled
 + Secure boot: Disabled
 + Debug lock state: Unlocked
 + Non-secure, Invasive debug lock (DBGLOCK) configuration: Unlocked
 + Non-secure, Non-invasive debug lock (NIDLOCK) configuration: Unlocked
 + Secure, Invasive debug lock (SPIDLOCK) configuration: Unlocked
 + Secure, Non-invasive debug lock (SPNIDLOCK) configuration: Unlocked
 + Non-secure, Invasive debug lock (DBGLOCK) current state: Unlocked
 + Non-secure, Non-invasive debug lock (NIDLOCK) current state: Unlocked
 + Secure, Invasive debug lock (SPIDLOCK) current state: Unlocked
 + Secure, Non-invasive debug lock (SPNIDLOCK) current state: Unlocked. The device is in normal state and secure debug is disabled.
 + Exporting a public command key from a hard-coded private command key... SL_STATUS_OK (cycles: 202467 time: 5328 us)
 + Reading the public command key from SE OTP... SL_STATUS_OK (cycles: 7589 time: 199 us)
 + Comparing exported public command key with SE OTP public command key... OK
 + Press ENTER to enable secure debug or press SPACE to exit.
```

2. Press `ENTER` to enable the secure debug.

```
 + Enable the secure debug... SL_STATUS_OK (cycles: 48313 time: 1271 us)
 + Press ENTER to lock the device or press SPACE to disable the secure debug and exit.
```

3. Press `ENTER` to lock the device with debug options `0x0c`.

```
 + Setting the debug options (0xc)... SL_STATUS_OK (cycles: 51091 time: 1344 us)
 + Locking the device... SL_STATUS_OK (cycles: 89683 time: 2360 us)
 + Device erase is enabled, press ENTER to disable device erase (optional if just for testing) or press SPACE to skip.
```

4. Press `ENTER` to disable device erase.

```
 + Warning: This is a ONE-TIME command which PERMANETLY disables device erase!
 + Press ENTER to confirm or press SPACE to skip if you are not sure.
```

5. Press `ENTER` to confirm the operation.

```
 . Get SE status... SL_STATUS_OK (cycles: 8496 time: 223 us)
 + The SE firmware version (MSB..LSB): 0001020E
 + Debug lock: Enabled
 + Device Erase: Disabled
 + Secure debug: Enabled
 + Secure boot: Disabled
 + Debug lock state: Locked
 + Non-secure, Invasive debug lock (DBGLOCK) configuration: Unlocked
 + Non-secure, Non-invasive debug lock (NIDLOCK) configuration: Unlocked
 + Secure, Invasive debug lock (SPIDLOCK) configuration: Locked
 + Secure, Non-invasive debug lock (SPNIDLOCK) configuration: Locked
 + Non-secure, Invasive debug lock (DBGLOCK) current state: Unlocked
 + Non-secure, Non-invasive debug lock (NIDLOCK) current state: Unlocked
 + Secure, Invasive debug lock (SPIDLOCK) current state: Locked
 + Secure, Non-invasive debug lock (SPNIDLOCK) current state: Locked. The device is in secure debug lock state.
 + Press ENTER to issue a secure debug unlock or press SPACE to exit.
```

6. Press `SPACE` to exit.

```
 . SE manager deinitialization... SL_STATUS_OK (cycles: 9 time: 0 us)
```

Simplicity Commander

1. Run the `security status` command to get the selected device configuration.

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
commander security status --device EFR32MG21A010F1024 --serialno 440048205
```

```
SE Firmware version : 1.2.14
Serial number       : 00000000000000014b457fffe045a93
Debug lock          : Disabled
Device erase        : Enabled
Secure debug unlock : Disabled
Tamper status       : OK
Secure boot         : Disabled
Boot status         : 0x20 - OK
DONE
```

2. Run the `security writekey` command to provision the Public Command Key (e.g., `command_pubkey.pem`).

```
commander security writekey --command command_pubkey.pem --device EFR32MG21A010F1024 --serialno 440048205
```

```
Device has serial number 00000000000000014b457fffe045a93
================================================================================
Please look through any warnings before proceeding.
THIS IS A ONE-TIME command which permanently ties debug and tamper access to certificates signed by this key.
Type 'continue' and hit enter to proceed or Ctrl-C to abort:
================================================================================
continue
DONE
```

> **Note**: The Public Command Key cannot be changed once written.

3. Run the `security readkey` command to read the Public Command Key from the SE OTP.

```
commander security readkey --command --device EFR32MG21A010F1024 --serialno 440048205
```

```
B1BC6F6FA56640ED522B2EE0F5B3CF7E5D48F60BE8148F0DC08440F0A4E1DCA4
7C04119ED6A1BE31B7707E5F9D001A659A051003E95E1B936F05C37EA793AD63
DONE
```

4. Run the `security lockconfig` command to enable the secure debug.

```
commander security lockconfig --secure-debug-unlock enable --device EFR32MG21A010F1024 --serialno 440048205
```

```
Secure debug unlock was enabled
DONE
```

5. a. For the **TrustZone-unaware** application, run the `security lockcommand` to lock the selected device.

```
commander security lock --device EFR32MG21A010F1024 --serialno 440048205
```

```
Device is now locked.
DONE
```

b. For the **TrustZone-aware** application, run the security lock --trustzone #### command to set the debug options (e.g., `1100`) and lock the selected device. The bit order of #### is SPNIDLOCK (MSB), SPIDLOCK, NIDLOCK, and DBGLOCK (LSB).

```
commander security lock --trustzone 1100 --device EFR32MG21A010F1024 --serialno 440048205
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
Writing debug restriction bits:
DBGLOCK:   0
NIDLOCK:   0
SPIDLOCK:  1
SPNIDLOCK: 1
Device is now locked.
DONE
```

Notes:

- The `--trustzone` option for the `security lock` command requires Simplicity Commander **≥ v1.13.3**.
- It is strongly recommended to upgrade to SE firmware **≥ v1.2.14** (xG21 and xG22) or **≥ v2.2.1** (other Series 2 devices) so that the debug options cannot be modified after the device is locked.
- Use `commander security lock` without the `--trustzone` #### option if the default setting of debug options (`0000`) is good enough for a TrustZone-aware application.

6. Run the `security disabledeviceerase` command to disable device erase. This is an **IRREVERSIBLE** action, and should be the last step in production.

```
commander security disabledeviceerase --device EFR32MG21A010F1024 --serialno 440048205
```

```
================================================================================
THIS IS A ONE-TIME command which Permanently disables device erase.
If secure debug lock has not been set, there is no way to regain debug access to this device.
Type 'continue' and hit enter to proceed or Ctrl-C to abort:
================================================================================
continue
Disabled device erase successfully
DONE
```

> **Note**: The debug options cannot be reset to the default value `0000` (unlock) if the device erase option is disabled.

7. a. For Simplicity Commander **< v1.13.3**, run the `security status` command to check the debug lock status of the device.

```
commander security status --device EFR32MG21A010F1024 --serialno 440048205
```

```
SE Firmware version : 1.2.14
Serial number       : 00000000000000014b457fffe045a93
Debug lock          : Enabled
Device erase        : Disabled
Secure debug unlock : Enabled
Tamper status       : OK
Secure boot         : Disabled
Boot status         : 0x20 - OK
DONE
```

b. For Simplicity Commander **≥ v1.13.3**, run the `security status --trustzone` command to check the full debug lock status of the device.

```
commander security status --trustzone --device EFR32MG21A010F1024 --serialno 440048205
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
SE Firmware version : 1.2.14
Serial number      : 000000000000000014b457fffe045a93
Debug lock         : Enabled
Device erase       : Disabled
Secure debug unlock : Enabled
Debug lock state: Locked
Non-secure, invasive debug lock     (DBGLOCK)  : Unlocked
Non-secure, non-invasive debug lock (NIDLOCK)  : Unlocked
Secure, invasive debug lock         (SPIDLOCK) : Locked
Secure, non-invasive debug lock     (SPNIDLOCK): Locked
Non-secure, invasive debug lock state     (DBGLOCK)  : Unlocked
Non-secure, non-invasive debug lock state (NIDLOCK)  : Unlocked
Secure, invasive debug lock state         (SPIDLOCK) : Locked
Secure, non-invasive debug lock state     (SPNIDLOCK): Locked
Tamper status      : OK
Secure boot        : Disabled
Boot status        : 0x20 - OK
DONE
```

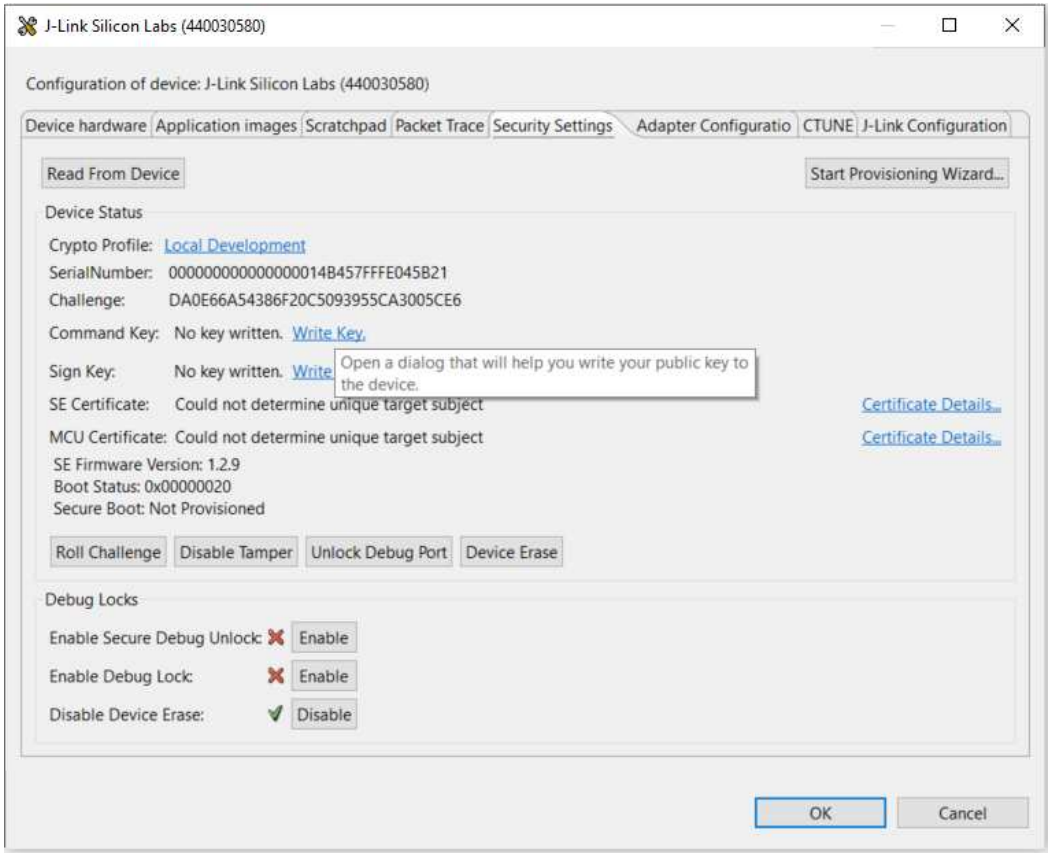
Simplicity Studio

1. Run `the util keytotoken` command to convert the Public Command Key file (PEM format) into a text file (`command_pubkey.txt`).

```
commander `util keytotoken` command_pubkey.pem --outfile command_pubkey.txt
```

```
Writing EC tokens to command_pubkey.txt...
DONE
```

2. Open **Security Settings** of the selected device as described in Using Simplicity Studio.
3. Click the **WriteKey** link next to **Command Key**: to open a dialog box.



4. The **Write Command Key** dialog box is displayed.

Copyright © 2025 Silicon Laboratories. All rights reserved.



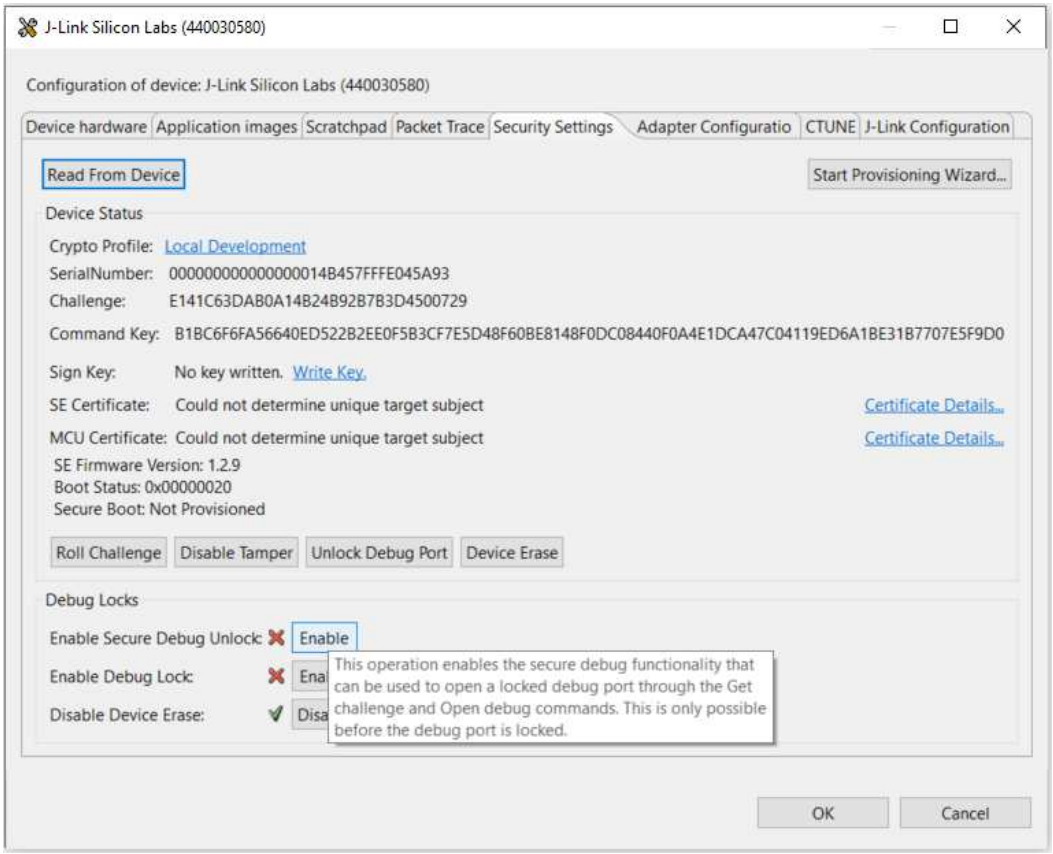
5. Open the `command_pubkey.txt` file generated in step 1.

```
MFG_SIGNED_BOOTLOADER_KEY_X : B1BC6F6FA56640ED522B2EE0F5B3CF7E5D48F60BE8148F0DC08440F0A4E1DCA4
MFG_SIGNED_BOOTLOADER_KEY_Y : 7C04119ED6A1BE31B7707E5F9D001A659A051003E95E1B936F05C37EA793AD63
```

6. Copy Public Command Key (X-point `B1BC...` first, then Y-point `7C04...`) to **Command Key**: box.

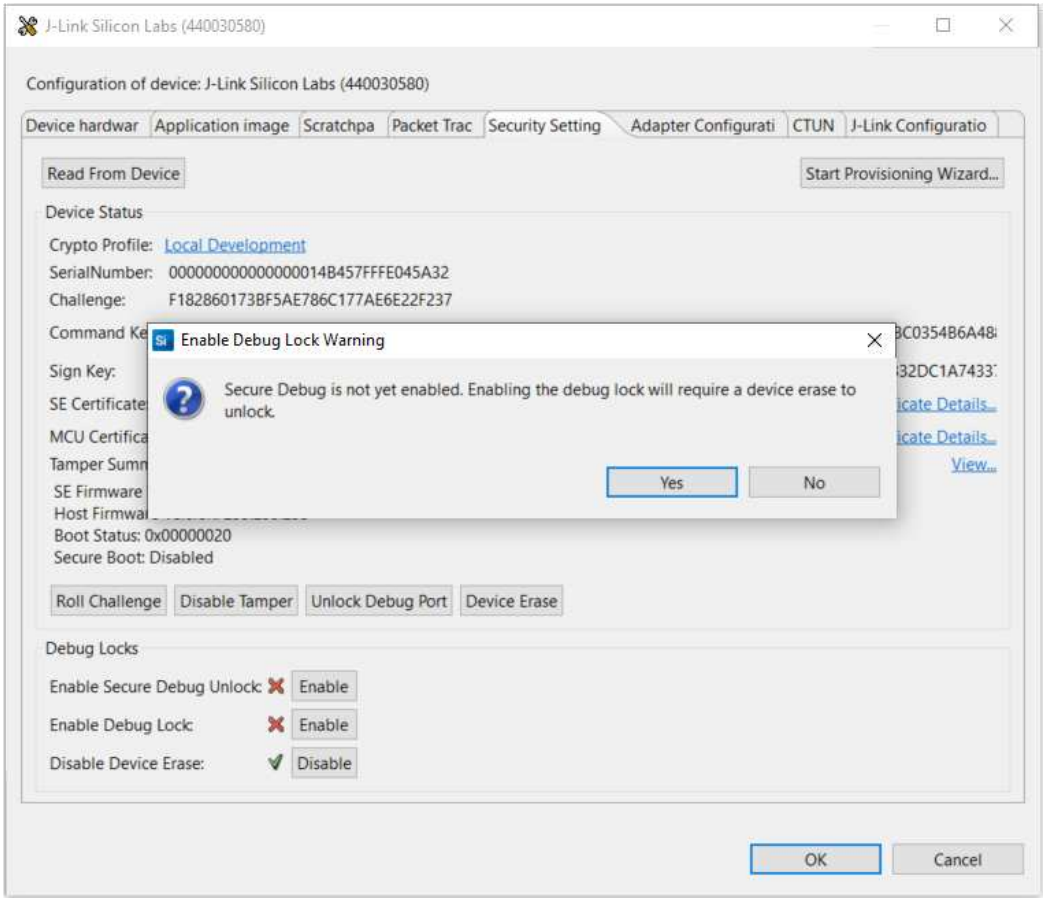


7. Click **[Write]** to provision the Public Command Key.
8. Click **[Enable]** next to **Enable Secure Debug Unlock**: to enable the secure debug functionality.



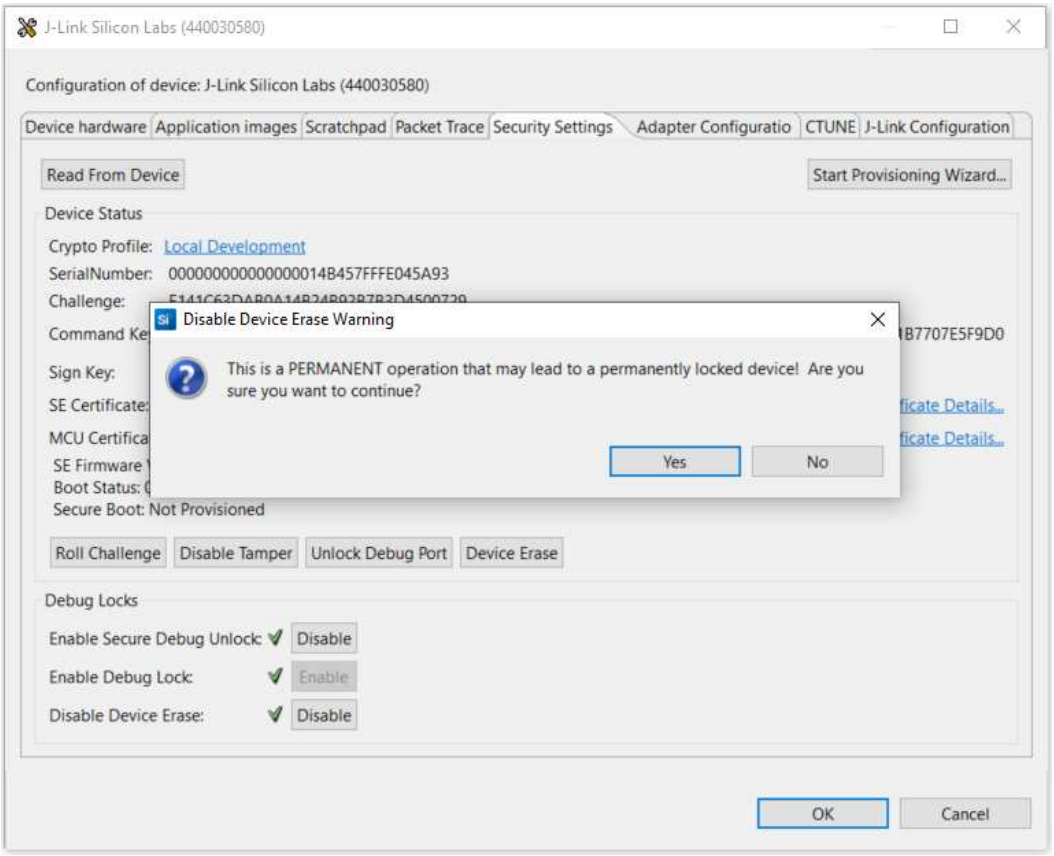
Copyright © 2025 Silicon Laboratories. All rights reserved.

Click **[Enable]** next to **Enable Debug Lock**: to lock the device. This configures secure debug lock.



10. Click **[Disable]** next to **Disable Device Erase**: to disable the device erase. The following **Disable Device Erase Warning** is displayed. Click **[Yes]** to confirm.

Copyright © 2025 Silicon Laboratories. All rights reserved.



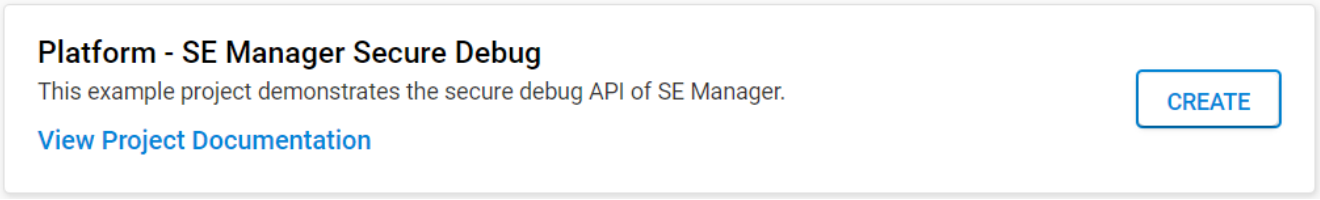
> **Note**: This is an **IRREVERSIBLE** action, and should be the last step in production.

Secure Debug Unlock and Roll Challenge

SE Manager - Secure Debug Platform Example

Click the `View Project Documentation` link to open the `readme` file.

> **Note**: The secure debug platform example can only run on the HSE device.



1. Use a secure debug lock device with matched Public Command Key.

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
. Get SE status... SL_STATUS_OK (cycles: 8496 time: 223 us)
+ The SE firmware version (MSB..LSB): 0001020E
+ Debug lock: Enabled
+ Device Erase: Disabled
+ Secure debug: Enabled
+ Secure boot: Disabled
+ Debug lock state: Locked
+ Non-secure, Invasive debug lock (DBGLOCK) configuration: Unlocked
+ Non-secure, Non-invasive debug lock (NIDLOCK) configuration: Unlocked
+ Secure, Invasive debug lock (SPIDLOCK) configuration: Locked
+ Secure, Non-invasive debug lock (SPNIDLOCK) configuration: Locked
+ Non-secure, Invasive debug lock (DBGLOCK) current state: Unlocked
+ Non-secure, Non-invasive debug lock (NIDLOCK) current state: Unlocked
+ Secure, Invasive debug lock (SPIDLOCK) current state: Locked
+ Secure, Non-invasive debug lock (SPNIDLOCK) current state: Locked. The device is in secure debug lock state.
+ Press ENTER to issue a secure debug unlock or press SPACE to exit.
```

2. Press `ENTER` to unlock the device with debug mode request `0x3e`.

```
+ Creating a private certificate key in a buffer... SL_STATUS_OK (cycles: 202354 time: 5325 us)
+ Exporting a public certificate key from a private certificate key... SL_STATUS_OK (cycles: 199394 time: 5247 us)
+ Read the serial number of the SE and save it to access certificate... SL_STATUS_OK (cycles: 7084 time: 186 us)
+ Signing the access certificate with private command key... SL_STATUS_OK (cycles: 221849 time: 5838 us)
+ Request challenge from the SE and save it to challenge response... SL_STATUS_OK (cycles: 4418 time: 116 us)
+ Signing the challenge response with private certificate key... SL_STATUS_OK (cycles: 220833 time: 5811 us)
+ Creating an unlock token (DEBUG_MODE_REQUEST = 0x3e) to unlock the device... SL_STATUS_OK (cycles: 935778 time: 24625 us)
+ Get debug status to verify the device is unlocked... SL_STATUS_OK (cycles: 9017 time: 237 us)
+ Success to unlock the device!
. Get SE status... SL_STATUS_OK (cycles: 8683 time: 228 us)
+ The SE firmware version (MSB..LSB): 0001020D
+ Debug lock: Enabled
+ Device Erase: Enabled
+ Secure debug: Enabled
+ Secure boot: Disabled
+ Debug lock state: Unlocked
+ Non-secure, Invasive debug lock (DBGLOCK) configuration: Unlocked
+ Non-secure, Non-invasive debug lock (NIDLOCK) configuration: Unlocked
+ Secure, Invasive debug lock (SPIDLOCK) configuration: Locked
+ Secure, Non-invasive debug lock (SPNIDLOCK) configuration: Locked
+ Non-secure, Invasive debug lock (DBGLOCK) current state: Unlocked
+ Non-secure, Non-invasive debug lock (NIDLOCK) current state: Unlocked
+ Secure, Invasive debug lock (SPIDLOCK) current state: Unlocked
+ Secure, Non-invasive debug lock (SPNIDLOCK) current state: Unlocked. The device is in secure debug unlock state.
+ Issue a power-on or pin reset to re-enable the secure debug lock.
+ Press ENTER to roll the challenge to invalidate the current unlock token or press SPACE to exit.
```

3. Press `ENTER` to roll the challenge.

```
. Check and roll the challenge.
+ Request current challenge from the SE... SL_STATUS_OK (cycles: 4450 time: 117 us)
+ The current challenge (16 bytes):
  FA A7 AA 5E EF E6 18 23 E5 21 89 84 DB 7E 52 7D
+ Rolling the challenge... SL_STATUS_OK (cycles: 19757 time: 519 us)
+ Request rolled challenge from the SE... SL_STATUS_OK (cycles: 4628 time: 121 us)
+ The rolled challenge (16 bytes):
  5A 7A 81 CC 6E 46 C1 EF B4 A4 CA 7A DD A9 85 EB
+ Issue a power-on or pin reset to activate the rolled challenge.
. SE manager deinitialization... SL_STATUS_OK (cycles: 9 time: 0 us)
```

Simplicity Commander

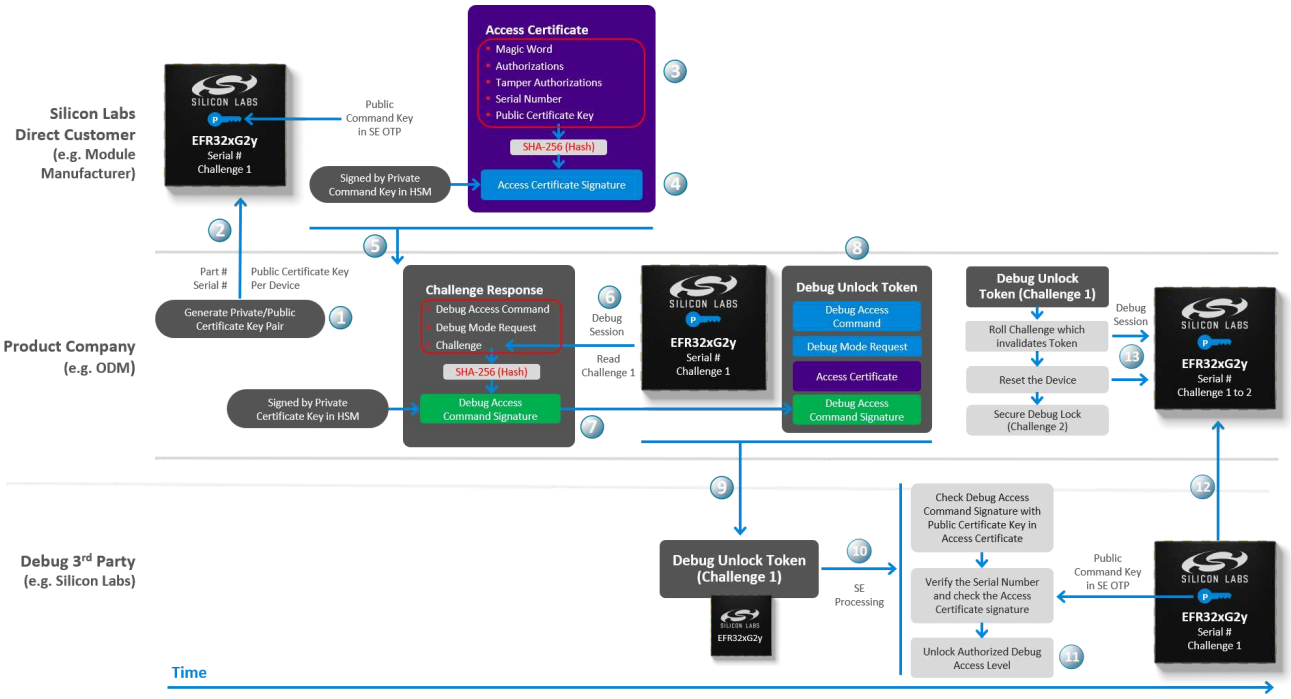
The secure debug was designed with three organizations in mind:

Copyright © 2025 Silicon Laboratories. All rights reserved.

- Direct Customer to whom Silicon Labs sells the chip. This chip has the Public Command Key installed in the SE OTP.
- That Direct Customer may be creating a white-labeled product for another company or a sub-component that goes into another company's product. The Product Company is the customer of the direct customer.
- The Debug 3rd Party could be anyone, internal or external, that the Product Company decides is qualified to debug the device.

Because the Public Command Key is installed into the SE OTP of a large number of devices and cannot be changed, the corresponding Private Command Key must be guarded by a very stringent process. If this Private Command Key is ever leaked, all the devices programmed with the corresponding Public Command Key will be compromised.

A secure debug unlock user case is described in the following figure.



The secure debug unlock flow moving across the time axis from left to right is explained below:

1. The Product Company creates a Private/Public Certificate Key pair for each device. Because the key pair is assigned only to a single device, the company may not need to protect the Private Certificate Key as securely as the Private Command Key by the Direct Customer.
   In this example, the Private/Public Certificate Key pair (`cert_key.pem` and `cert_pubkey.pem`) is generated by running the `util genkey` command.

   ```
   commander util genkey --type ecc-p256 --privkey cert_key.pem --pubkey cert_pubkey.pem
   ```

   ```
   Generating ECC P256 key pair...
   Writing private key file in PEM format to cert_key.pem
   Writing public key file in PEM format to cert_pubkey.pem
   DONE
   ```

2. The Public Certificate Key (`cert_pubkey.pem`) for each device is passed to the Silicon Labs Direct Customer. The part number and serial number are also required if Direct Customer cannot access the device.
   If necessary, run the `security status` command to get the device serial number.

   ```
   commander security status --device EFR32MG21A010F1024 --serialno 440048205
   ```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
SE Firmware version : 1.2.14
Serial number      : 00000000000000014b457fffe045a32Debug lock        : Enabled
Device erase       : Disabled
Secure debug unlock : Enabled
Tamper status      : OK
Secure boot        : Disabled
Boot status        : 0x20 - OK
DONE
```

3. The Direct Customer then places that Public Certificate Key in the access certificate. The access certificate is per device because it contains the unique device serial number. This certificate is generated once upon creation of the device, and thereafter, is generally only modified when the Private/Public Certificate Key pair is changed by the Product Company.
   Run the `security gencert` command with the following parameters from the Product Company to generate an unsigned access certificate (`access_certificate.extsign`) in Security Store:
   - Device part number
   - Device serial number
   - Public Certificate Key

```
commander security gencert --device EFR32MG21A010F1024 --deviceserialno 00000000000000014b457fffe045a32
--cert-pubkey cert_pubkey.pem --extsign
```

```
Authorization file written to Security Store:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000014b457fffe045a32/certificate_authori

Cert key written to Security Store:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000014b457fffe045a32/cert_pubkey.pem

Created an unsigned certificate in Security Store:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000014b457fffe045a32/access_certificate

DONE
```

   Note:
   - The `--extsign` option to create an unsigned access certificate is only available in Simplicity Commander Version 1.11.2 or above.
   - The unsigned access certificate is generated with the default certificate authorization file (`certificate_authorization.json`) which uses `0x0000003e` for Authorizations and `0x00000000` (HSE-SVM device) for Tamper Authorizations (Table Elements of the Access Certificate on page 12).
4. The signing of the access certificate can be done by passing an unsigned access certificate to a Hardware Security Module (HSM) containing the Private Command Key.
   In this example, the OpenSSL is used to sign the access certificate (`access_certificate.extsign`) in Security Store with the Private Command Key (`command_key.pem`). The access certificate signature is in the cert_signature.binfile.

```
openssl dgst -sha256 -binary -sign command_key.pem -out cert_signature.binaccess_certificate.extsign
```

   Run the `util signcert` command with the following parameters to verify the signature and generate the signed access certificate (`access_certificate.bin`):
   - Unsigned access certificate
   - Access certificate signature
   - Public Command Key

```
commander util signcert access_certificate.extsign --cert-type access --signature cert_signature.bin
--verify command_pubkey.pem --outfile access_certificate.bin
```

```
R = D97E43FEA278207080D6D0808B46810C1167F123AF1CA9FAF2DE0F4322B97ACE
S = FEDFEA11A3C83AFFCD5293283B13A50580862B9F651AAE08012C2BFB6BA8E697
Successfully verified signature
Successfully signed certificate
DONE
```

   Notes:

Copyright © 2025 Silicon Laboratories. All rights reserved.

- The access certificate signature can be in a Raw or Distinguished Encoding Rules (DER) format.

5. The access certificate is passed to the Product Company. The purpose of the access certificate is to grant overall debug access capabilities to the Product Company and authorize them to allow third parties to debug the device. The Product Company can now use the access certificate to generate the Debug Unlock Token. The same access certificate can be used to generate as many Debug Unlock Tokens as necessary without having to ever go back to the Direct Customer.
6. To create the Debug Unlock Token, a debug session must be started with the device and the challenge value (which is a random number `Challenge 1` in this example) should be read out to generate the challenge response.
   Run the `security gencommand` command to generate the challenge response without debug access command signature and store it in a file (`command_unsign.bin`).

```
commander security gencommand --action debug-unlock--unlock-param 1111 -o command_unsign.bin --nostore
--device EFR32MG21A010F1024 --serialno 440048205
```

```
Unsigned command file written to:
command_unsign.bin
DONE
```

   Notes:
   - The data in the `--unlock-param` option are the bits 2 to 5 of debug mode request in the challenge response.
   - The default value 1111(reset all debug options) is in place if the `security gencommand` command does not include the `--unlock-param` option.
7. The challenge response is then cryptographically hashed (SHA-256) to create a digest. The digest is then signed by the Private Certificate Key to generate the debug access command signature.
   The signing of the challenge response can be done by passing an unsigned challenge response to a Hardware Security Module (HSM) containing the Private Certificate Key.
   In this example, the OpenSSL is used to sign the challenge response (`command_unsign.bin`) with the Private Certificate Key (`cert_key.pem`). The debug access command signature is in the command_signature.binfile.

```
openssl dgst -sha256 -binary -sign cert_key.pem -out command_signature.bincommand_unsign.bin
```

8. Run the `security unlock` command with the access certificate (`access_certificate.bin`) from Direct Customer and debug access command signature (`command_signature.bin`) in step 7 to generate the Debug Unlock Token.

```
commander security unlock --cert access_certificate.bin --command-signature command_signature.bin--unlock-param 1111 --device
EFR32MG21A010F1024 --serialno 440048205
```

```
Certificate written to Security Store:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_000000000000000014b457fffe045a32/access_certificate

R = 67B51151F1E5F1BB9A49EC8D5885B221BD3D331D53741EEF54F81F0F3CB40455
S = 066C6AB5EDE3AE784DB1F75F44C5CA931736116D5A2104DBF44BC77ED8F49282
Command signature is valid
Secure debug successfully unlocked
Command unlock payload was stored in Security Store
DONE
```

   Notes:
   - Put the required files in the same folder to run the command.
   - The debug access command signature can be in a Raw or Distinguished Encoding Rules (DER) format.
   - It requires Simplicity Commander Version 1.11.2 or above to support signature in DER format.
   - The data in the `--unlock-param` option are the bits 2 to 5 of debug mode request in the Debug Unlock Token. This value **MUST** be equal to the value of `--unlock-param` option in step 6.
   - The default value `1111` (reset all debug options) is in place if the `security unlock` command does not include the `--unlock-param` option.
9. **(Alternative)** The key protection is not required if the Private Certificate Key is ephemeral. Steps 6 to 8 can be implemented by running the `security unlock` command with the access certificate (`access_certificate.bin`)from the Direct Customer and Private Certificate Key (`cert_key.pem`) to generate the Debug Unlock Token.

```
commander security unlock --cert access_certificate.bin --cert-privkey cert_key.pem--unlock-param 1111
--device EFR32MG21A010F1024 --serialno 440048205
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
Certificate written to Security Store:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000014b457fffe045a32/access_certificate

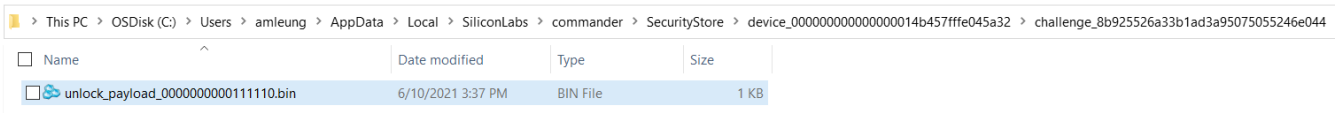
Cert key written to Security Store:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000014b457fffe045a32/cert_pubkey.pem

Created unsigned unlock command
Signed unlock command using
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000014b457fffe045a32/cert_key.pem
Secure debug successfully unlocked
Command unlock payload was stored in Security Store
DONE
```

Notes:

- The data in the `--unlock-param` option are the bits 2 to 5 of debug mode request in the Debug Unlock Token.
- The default value `1111` (reset all debug options) is in place if the `security unlock` command does not include the `--unlock-param` option.

10. The Debug Unlock Token (aka `Command unlock payload`) file (`unlock_payload_0000000000111110.bin`, where `0000000000111110` is the value of debug mode request) is stored in the Security Store. The location in Windows is `C:\Users\<PC user name>\AppData\Local\SiliconLabs\commander\SecurityStore\device_\<Serial number>\challenge_\<Challenge value>`.



Users can also use the security getpathcommand to get the path of the Security Store or a specified device.

```
commander security getpath
```

```
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore
DONE
```

```
commander security getpath --deviceserialno 00000000000000014b457fffe045a32
```

```
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000014b457fffe045a32
DONE
```

11. The Debug Unlock Token and the device are now delivered to the Debug 3rd Party.
Run the `security gencommand` command to create the Security Store to place the Debug Unlock Token file.

```
commander security gencommand --action debug-unlock --device EFR32MG21A010F1024 --serialno 440048205
```

```
Unsigned command file written to Security Store:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000014b457fffe045a32/challenge_8b9255

DONE
```

Copy the Debug Unlock Token file (`unlock_payload_0000000000111110.bin`) from Product Company to the Windows Security Store `challenge_\<Challenge value>` folder located in `C:\Users\<PC user name>\AppData\Local\SiliconLabs\commander\SecurityStore\device_\<Serial number>\challenge_\<Challenge value>`.

12. The device compares the Debug Unlock Token contents with its internal serial number, challenge value, and Public Command Key to determine the token's authenticity. If authentic, it will execute the debug access command to unlock the device; otherwise, it will ignore the command.
Run the `security unlock` command to unlock the device.

```
commander security unlock --unlock-param 1111 --device EFR32MG21A010F1024 --serialno 440048205
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
Unlocking with unlock payload:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000014b457fffe045a32/challenge_8b9255
 debug successfully unlocked
DONE
```

Notes:

- If the security store has multiple tokens for the selected device, use --unlock-param option to specify which unlock token is chosen to unlock the device.
- Simplicity Commander will only use the token with value 1111 (error if not available) from the security store to unlock the device if the security unlock command does not include the --unlock-param option.

13. For Simplicity Commander **≥ v1.13.3**, run the `security status --trustzone` command to check the full debug lock status of the device.

```
commander security status --trustzone --device EFR32MG21A010F1024 --serialno 440048205
```

```
SE Firmware version : 1.2.14
Serial number       : 00000000000000014b457fffe045a32
Debug lock          : Enabled
Device erase        : Disabled
Secure debug unlock : Enabled
Debug lock state: Unlocked
Non-secure, invasive debug lock     (DBGLOCK)  : Unlocked
Non-secure, non-invasive debug lock (NIDLOCK)  : Unlocked
Secure, invasive debug lock         (SPIDLOCK) : Locked
Secure, non-invasive debug lock     (SPNIDLOCK): Locked
Non-secure, invasive debug lock state     (DBGLOCK)  : Unlocked
Non-secure, non-invasive debug lock state (NIDLOCK)  : Unlocked
Secure, invasive debug lock state         (SPIDLOCK) : Unlocked
Secure, non-invasive debug lock state     (SPNIDLOCK): Unlocked
Tamper status       : OK
Secure boot         : Disabled
Boot status         : 0×20 - OK
DONE
```

14. The Debug 3rd Party can now use this same Debug Unlock Token to unlock the device (step 12), over and over again after each power-on or pin reset, until they have finished debugging the device.
15. Once the Debug 3rd Party has finished debugging, they will send the device back to the Product Company.
16. Once the Product Company receives the device, they will immediately start a debug session, roll the challenge (from `Challenge 1` to `Challenge 2` in this example), and put the device back into the secure debug lock state. Rolling the challenge will effectively invalidate any Debug Unlock Token that has been previously given to any third party.
    Run the `security rollchallenge` command and reset the device to invalidate the current Debug Unlock Token. The challenge cannot be rolled before it has been used at least once — that is, by running the `security unlock` or `security disabletamper` command.

```
commander security rollchallenge --device EFR32MG21A010F1024 --serialno 440048205
```

```
Challenge was rolled successfully.
DONE
```

The unlock token is invalidated after rolling the challenge because any previously issued Debug Unlock Token now contains a different challenge value (`Challenge 1`) than the challenge value currently in the device (`Challenge 2`).
The validation process of any previously issued Debug Unlock Token will always fail until a new Debug Unlock Token is issued with a current matching challenge value (`Challenge 2`).

> **Note**: Direct Customer can directly use the Private Command Key on the connected chip to generate the Debug Unlock Token in Security Store. But it has a high risk (cannot use HSM) to leak the Private Command Key to a 3rd party when using this approach.

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
commander security unlock --command-key command_key.pem--unlock-param 1111 --device EFR32MG21A010F1024
--serialno 440048205
```

```
Authorization file written to Security Store:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000014b457fffe045a32/certificate_authori

Generating ECC P256 key pair...
Cert public key stored at:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000014b457fffe045a32/cert_pubkey.pem

Cert private key stored at:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000014b457fffe045a32/cert_key.pem
Command public key stored at:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000014b457fffe045a32/command_pubkey

Command private key stored at:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000014b457fffe045a32/command_key.per

Certificate was signed with key:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000014b457fffe045a32/command_key.per

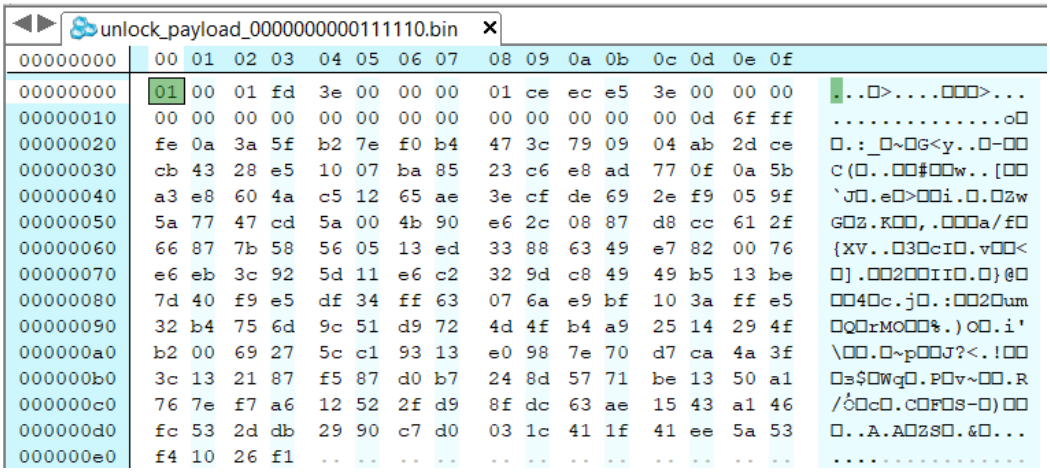
Certificate written to Security Store:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000014b457fffe045a32/access_certificate

Created unsigned unlock command
Signed unlock command using
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000014b457fffe045a32/cert_key.pem
Secure debug successfully unlocked
Command unlock payload was stored in Security Store
DONE
```

Simplicity Studio

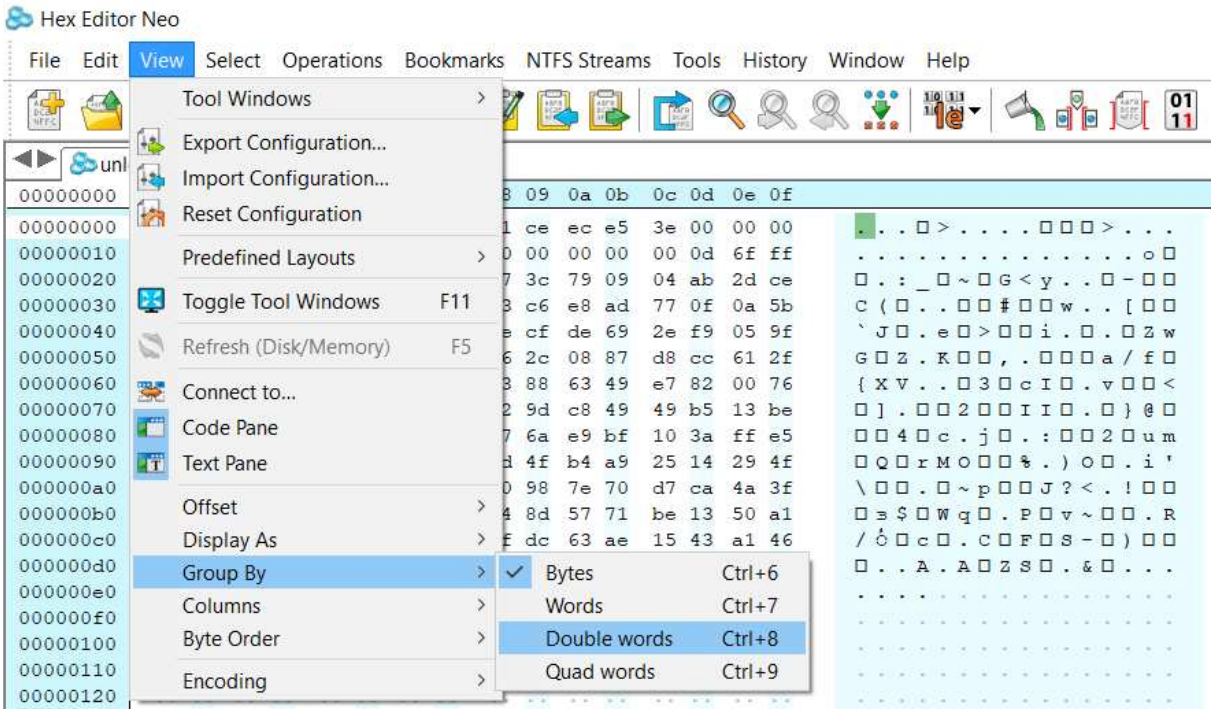
Use the Debug Unlock Token file (`unlock_payload_0000000000111110.bin`) generated in Using Simplicity Commander steps 8 or 9 to unlock the device with Simplicity Studio.

1. Open the `unlock_payload_0000000000111110.bin` file with the Hex Editor Neo.

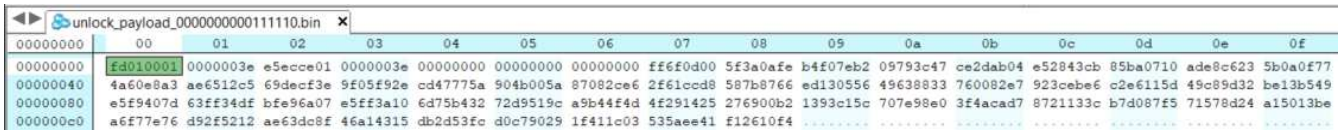


2. Click **View** to open the context menu, and then select **Group By** → **Double words** to convert the token into a little-endian format.

Copyright © 2025 Silicon Laboratories. All rights reserved.



3. Select all (Ctrl+A) and copy (Ctrl+C) the Debug Unlock Token to a text editor.

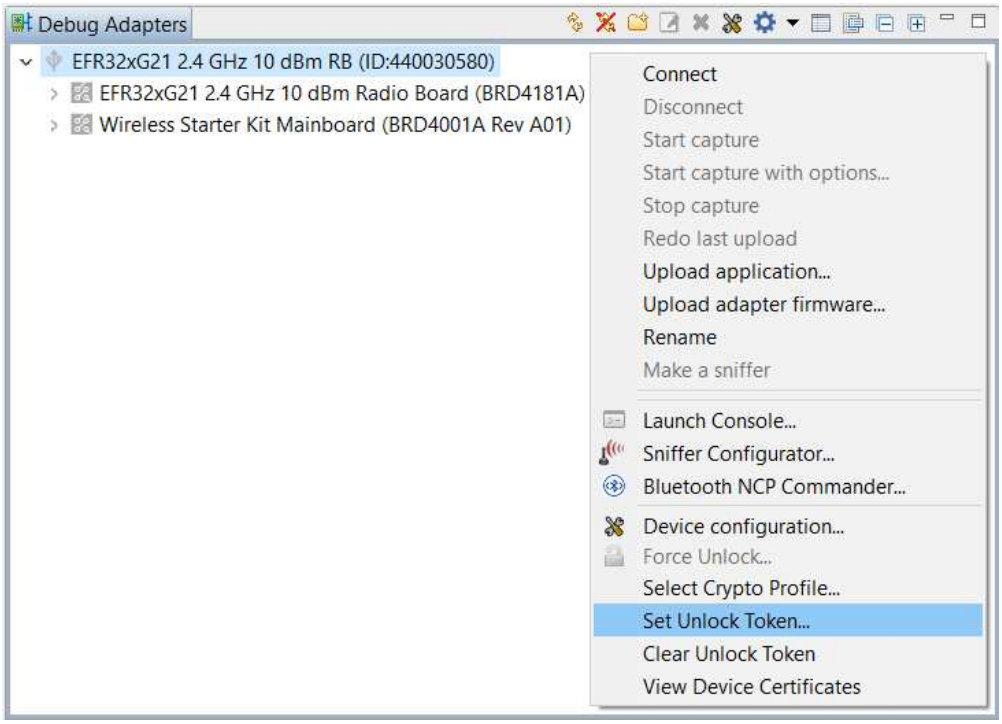


4. Use the text editor to remove all the spaces from the token.

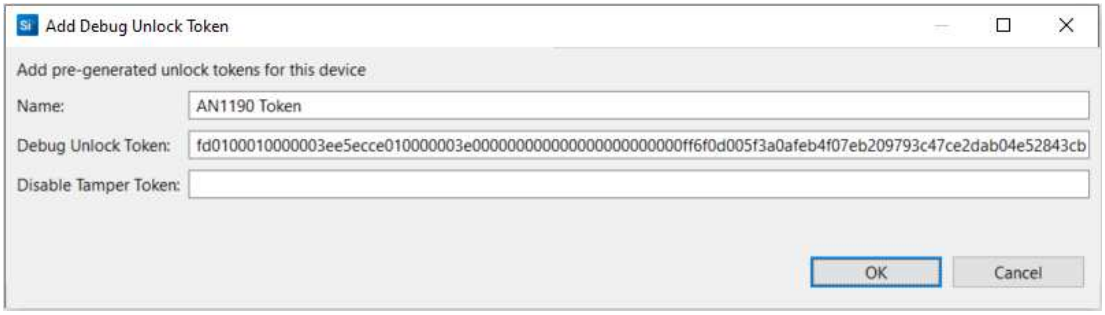
```
fd0100010000003ee5ecce010000003e00000000000000000000000000ff6f0d00...
```

5. Right-click the selected debug adapter **RB (ID:J-Link serial number)** to display the context menu.

Copyright © 2025 Silicon Laboratories. All rights reserved.



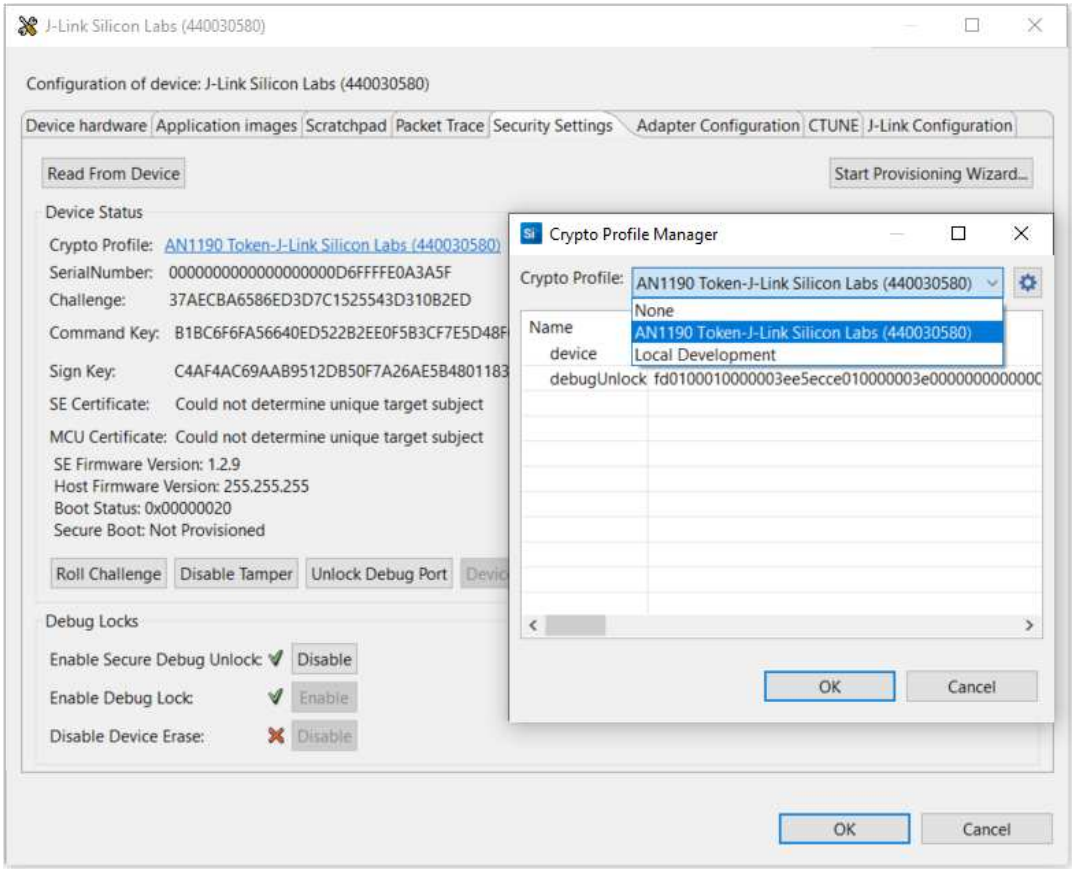
6. Click **Set Unlock Token** to open the **Add Debug Unlock Token** dialog box. Enter the name (e.g., `AN1190 Token`) for this Debug Unlock Token, and copy the content in step 4 to the **Debug Unlock Token**: box. Click **[OK]** to confirm and exit.



> **Note**: The Simplicity Studio can only keep one Debug Unlock Token on each WSTK.

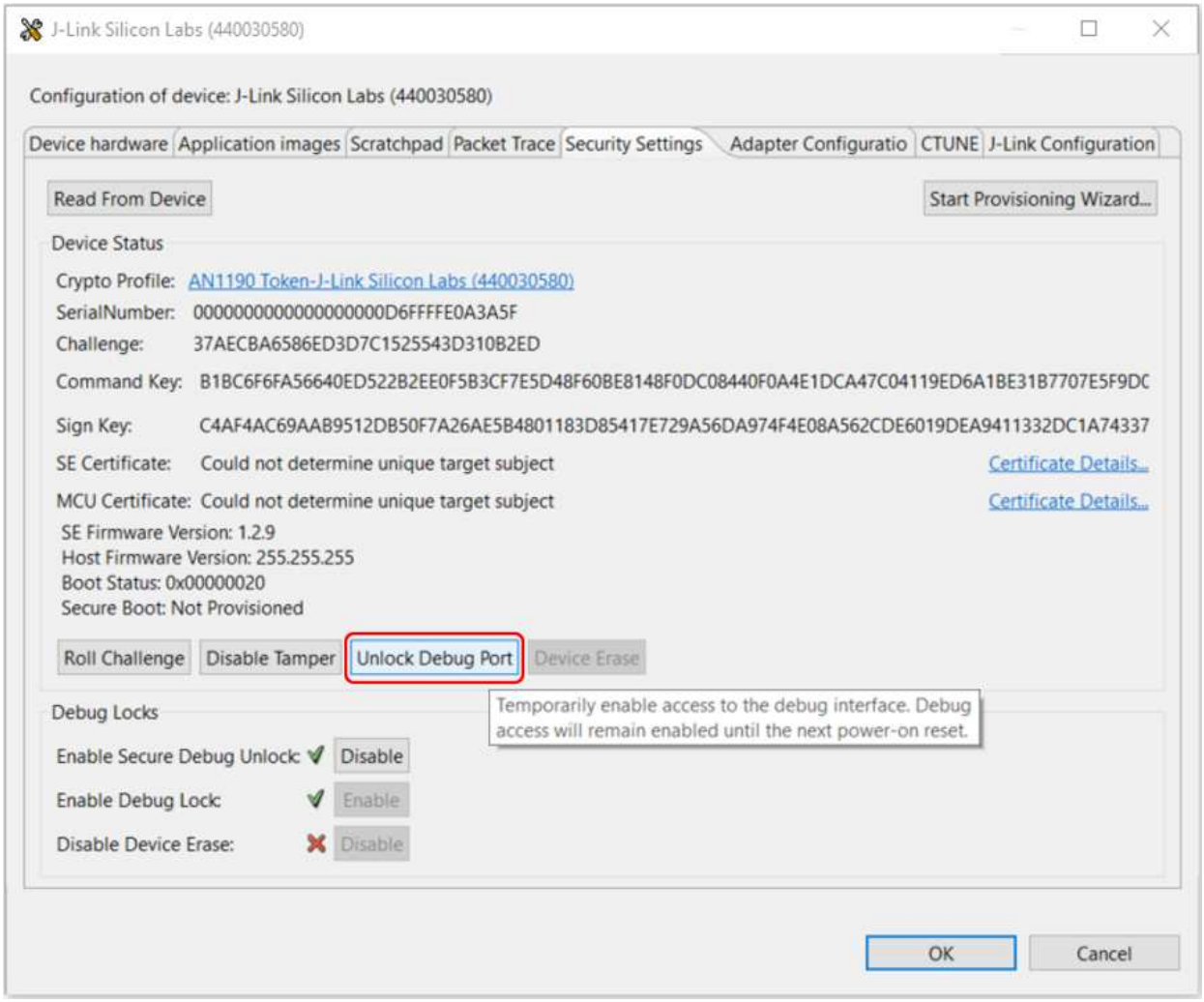
7. Open **Security Settings** of the selected device as described in Using Simplicity Studio.
8. The token added in step 6 should display on the **Crypto Profile**: field. If not, click the link next to **Crypto Profile**: to select the token from the **Crypto Profile Manager** drop-down list. The Simplicity Studio will automatically add the WSTK J-Link serial number (`-J-Link Silicon Labs (serial number)`) to the token's name.

Copyright © 2025 Silicon Laboratories. All rights reserved.



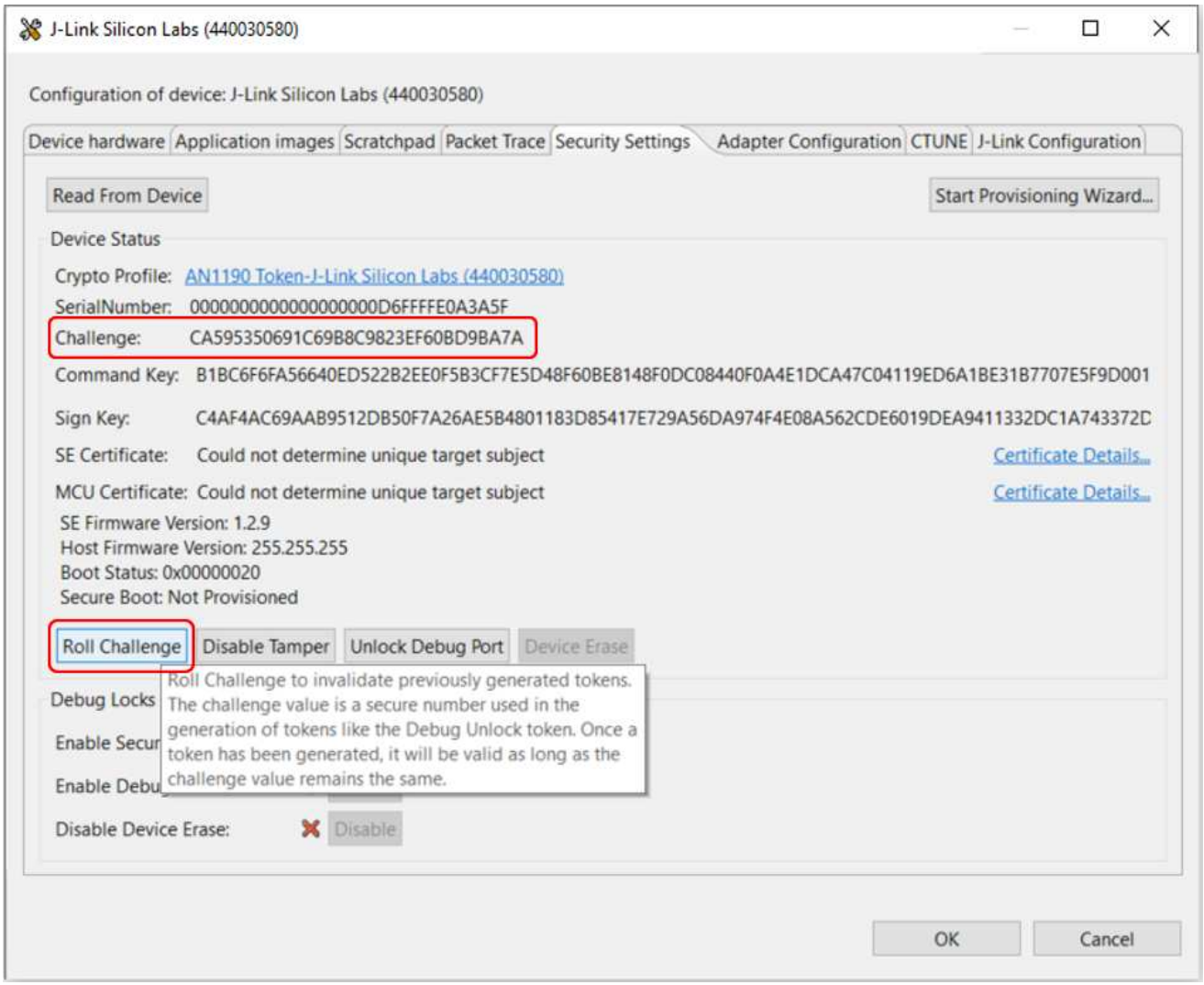
9. Click **[Unlock Debug Port]** to use the token in **Crypto Profile**: to unlock the device (invalid token will display an error message). The device stays in the unlock state until the next power-on or pin reset. Click **[OK]** to exit.

Copyright © 2025 Silicon Laboratories. All rights reserved.



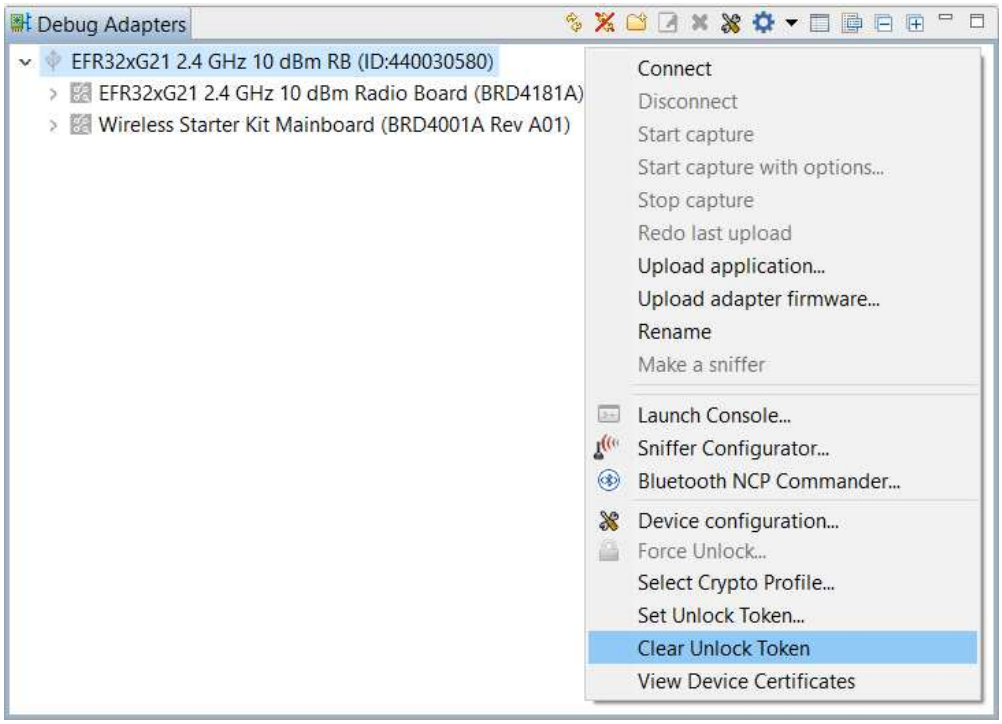
10. The Simplicity IDE will automatically use the selected Debug Unlock Token in **Crypto Profile** for debugging and flashing. For other IDE, the device should unlock again (step 9) after power-on or pin reset. After finished debugging, open the **Security Settings** of the selected device as described in Using Simplicity Studio.
11. Click **[Roll Challenge]** to generate a new challenge value to invalidate the Debug Unlock Token added in step 6. Click **[OK]** to exit.

Copyright © 2025 Silicon Laboratories. All rights reserved.



12. Right-click the selected debug adapter **RB Board (ID:J-Link serial number)** to display the context menu.

Copyright © 2025 Silicon Laboratories. All rights reserved.

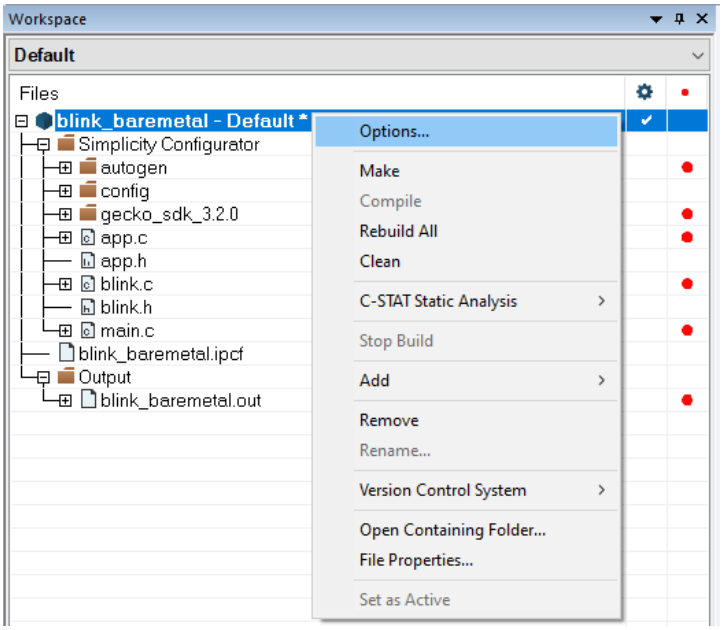


13. Click **[Clear Unlock Token]** to delete the WSTK Debug Unlock Token from Simplicity Studio.

IAR

Use the Debug Unlock Token file (`unlock_payload_0000000000111110.bin`) in the Security Store (Simplicity Commander step 11) to unlock the device with IAR (Windows).

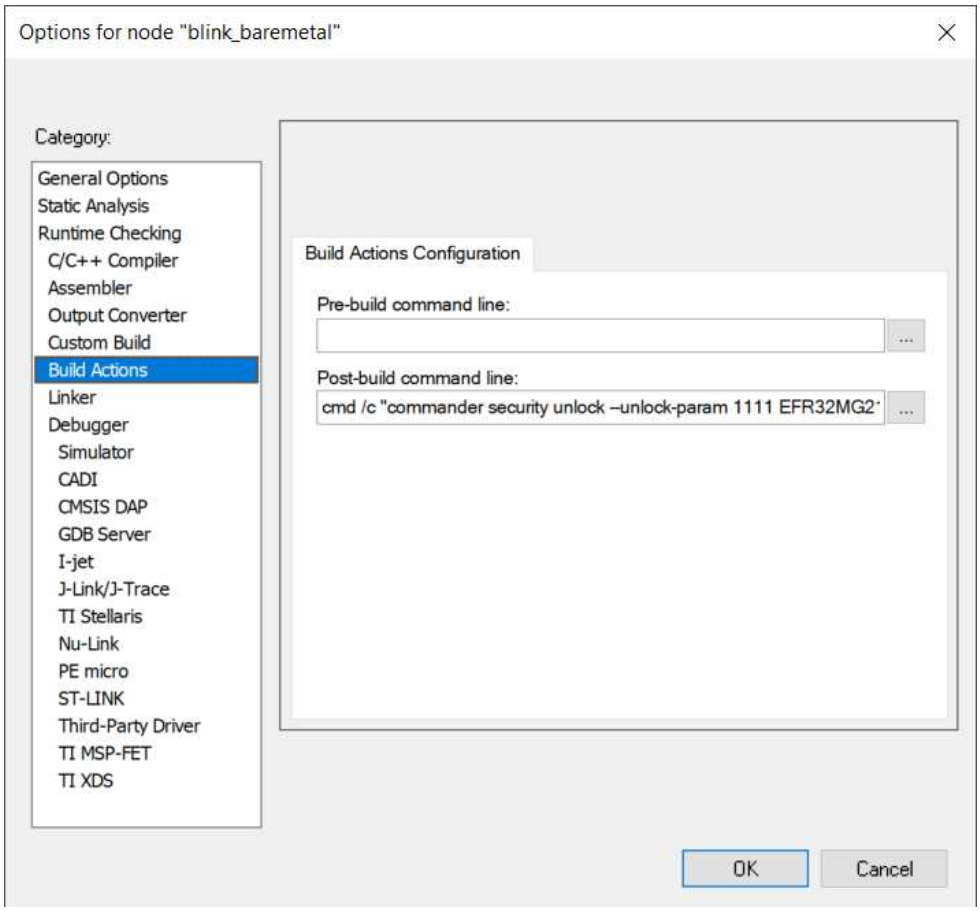
1. The Windows environment variable `PATH` should include the folder (`C:\SiliconLabs\SimplicityStudio\v5\developer\adapter_packs\commander`) that locates the `commander.exe` of Simplicity Commander.
2. Right-click the project in the workspace, and then click **Options...**.



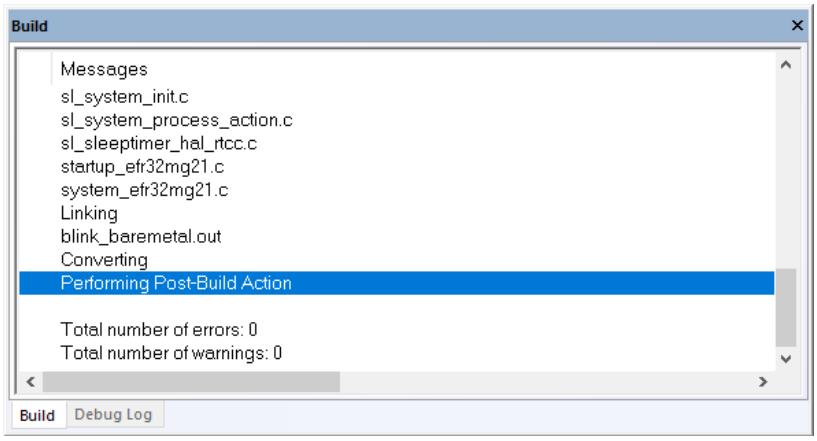
Copyright © 2025 Silicon Laboratories. All rights reserved.

3. Click **Build Actions** to open the **Build Actions Configuration** dialog box. Enter the phrase below to the **Post-build command line**: box. Click **[OK]** to exit.

```
cmd /c "commander security unlock--unlock-param 1111 EFR32MG21A010F1024 --serialno 440048205 > $PROJ_DIR$\log.txt 2>&1"
```



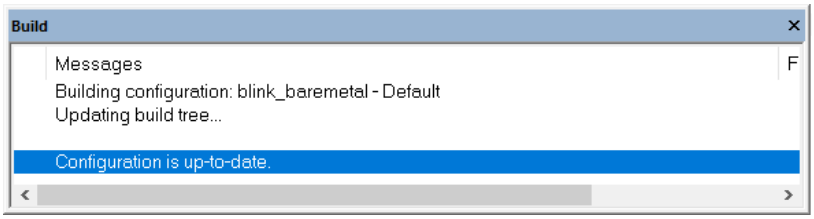
4. After building the project, the `security unlock` in the **Post-build command** unlocks the device using the Debug Unlock Token in Security Store. The device stays in the unlock state until the next power-on or pin reset.



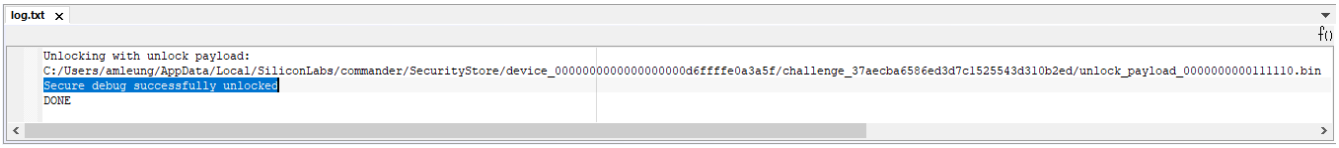
> **Note**: If the project is already up-to-date, it will not invoke the **Post-build command** to unlock the device. Use a dummy edit (add space or newline) on one of the source files in the project to trigger the build

Copyright © 2025 Silicon Laboratories. All rights reserved.

> action.



5. The `> $PROJ_DIR$\log.txt 2>&1` redirects the `security unlock` command output to the `log.txt` file in the IAR project folder.



Permanent Debug Lock

Simplicity Commander

1. Run the `security status` command to get the selected device configuration.

```
commander security status --device EFR32MG21A010F1024 --serialno 440048205
```

```
SE Firmware version : 1.2.14
Serial number       : 00000000000000000014b457fffe045a32
Debug lock          : Disabled
Device erase        : Enabled
Secure debug unlock : Disabled
Tamper status       : OK
Secure boot         : Disabled
Boot status         : 0x20 - OK
DONE
```

2. Run the `security lock` command to lock the selected device.

```
commander security lock --device EFR32MG21A010F1024 --serialno 440048205
```

```
WARNING: Secure debug unlock is disabled. Only way to regain debug access is to run a device erase.
Device is now locked.
DONE
```

3. Run the `security disabledeviceerase` command to disable device erase.

```
commander security disabledeviceerase --device EFR32MG21A010F1024 --serialno 440048205
```

```
================================================================================
THIS IS A ONE-TIME command which Permanently disables device erase.
If secure debug lock has not been set, there is no way to regain debug access to this device.
Type 'continue' and hit enter to proceed or Ctrl-C to abort:
================================================================================
continue
Disabled device erase successfully
DONE
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

> **Note**: This is an **IRREVERSIBLE** action, and should be the last step in production.

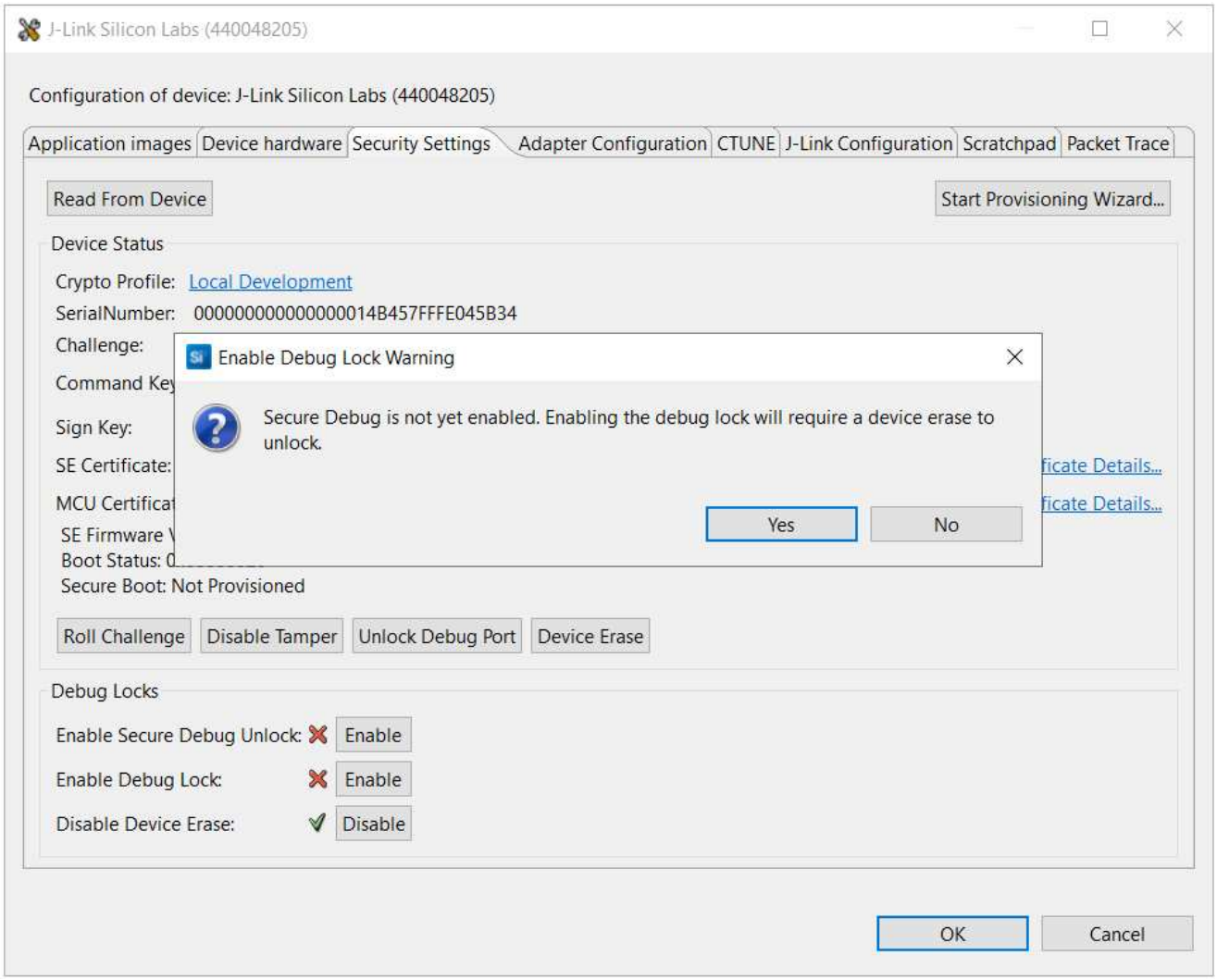
4. Run the `security status` command again to check the device configuration.

```
commander security status --device EFR32MG21A010F1024 --serialno 440048205
```

```
SE Firmware version : 1.2.14
Serial number       : 0000000000000000014b457fffe045a32
Debug lock          : Enabled
Device erase        : Disabled
Secure debug unlock : Disabled
Tamper status       : OK
Secure boot         : Disabled
Boot status         : 0×20 - OK
DONE
```

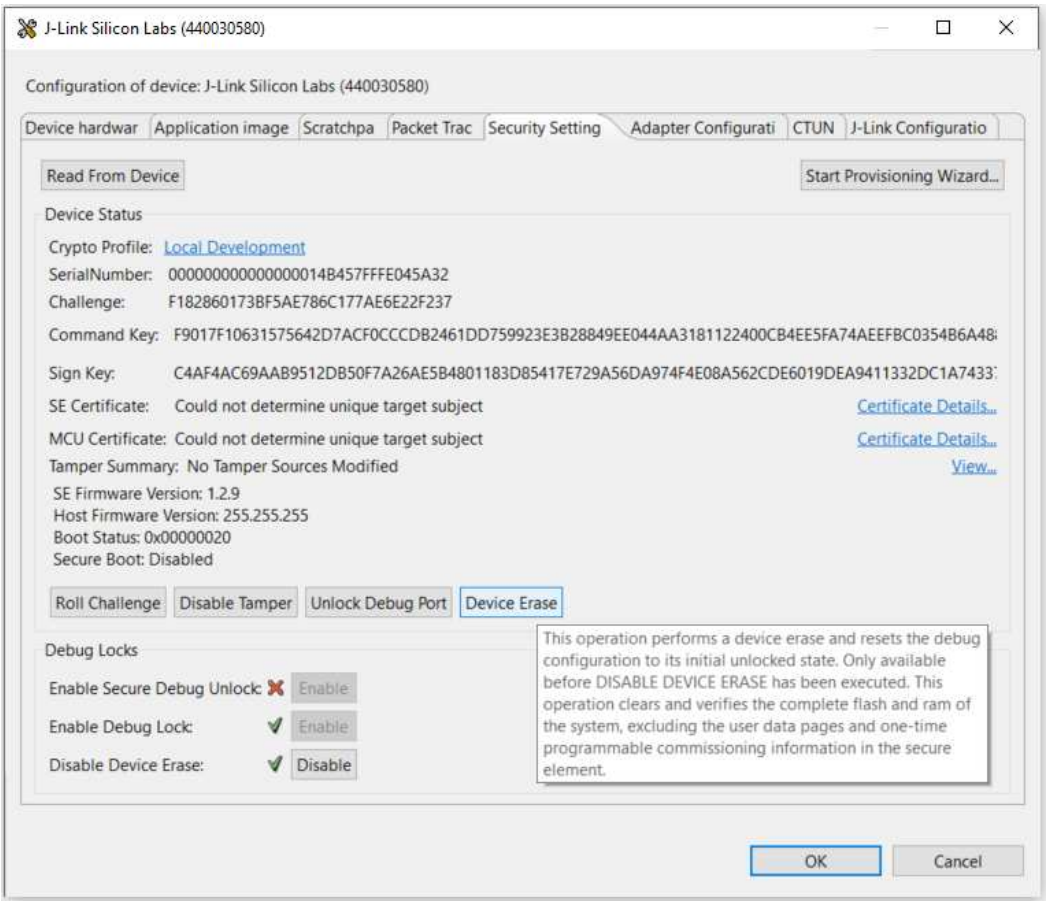
Simplicity Studio

1. Open **Security Settings** of the selected device as described in Using Simplicity Studio.
2. Click **[Enable]** next to **Enable Debug Lock**: to lock the device. The following **Enable Debug Lock Warning** is displayed. Click **[Yes]** to confirm. This configures standard debug lock.



Copyright © 2025 Silicon Laboratories. All rights reserved.

3. Click **[Disable]** next to **Disable Device Erase**: to disable the device erase. The following **Disable Device Erase Warning** is displayed. Click **[Yes]** to confirm. This configures a permanent debug lock.



> **Note**: This is an **IRREVERSIBLE** action, and should be the last step in production.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Precautions

Precautions

Device Erase for Secure Debug

Disable the Device Erase is mandatory for secure debug as described in the following table.

Secure Debug	Device Erase	Debug Lock	State	Description
Enabled	Enabled	Enabled	Insecure debug lock (1)	The device will return to the default debug lock properties after applying the standard debug unlock. (2)
Enabled	Disabled (3)	Enabled	Secure debug lock	The device cannot be unlocked using the Erase Device command. The device will change to the permanent debug lock state if disabling the Secure Debug property. (4)

Notes:

1. This state is only for secure debug testing.
2. See Standard Debug Unlock.
3. This is an **IRREVERSIBLE** action and should be disabled **AFTER** the secure debug is enabled.
4. See Permanent Debug Lock.

```
commander security lockconfig --secure-debug-unlock disable --device EFR32MG21A010F1024
--serialno 440048205
```

```
================================================================================
WARNING: Device erase is disabled and secure debug access is locked.
If disabling secured debug access, there is no way to regain debug access to this device if continuing with this command.
Type 'continue' and hit enter to proceed or Ctrl-C to abort:
================================================================================
continue
Secure debug unlock was disabled
DONE
```

Secure Boot and Debug Lock

The following table describes the different debug lock scenarios on the secure boot-enabled device.

Secure Debug	Device Erase	Debug Lock	State	Recover from Secure Boot Failure
Disabled	Enabled	Disabled	Standard debug unlock	Flash a correctly signed image.
Disabled	Enabled	Enabled	Standard debug lock	Flash a correctly signed image after standard debug unlocking the device.
Disabled	Disabled	Enabled	Permanent debug lock	There is no way to recover the device. Make sure the programmed image is correctly signed before locking the device.
Enabled	Disabled	Enabled	Secure debug lock	Flash a correctly signed image after secure debug unlocking the device.

Copyright © 2025 Silicon Laboratories. All rights reserved.

> **Note**: See section *Recover Devices when Secure Boot Fails* in Series 2 Secure Boot with RTSL to flash a correctly signed image on different debug lock scenarios.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Failure Analysis

Failure Analysis

The following table describes the different scenarios when returning a Series 2 device to Silicon Labs for failure analysis.

State	Secure Boot Disabled	Secure Boot Enabled (2)
Standard debug unlock	Device erase is not necessary for failure analysis.	Device erase is not necessary, but a correctly signed image is required to perform failure analysis.
Standard debug lock	Device erase is required to perform failure analysis.	Require device erase and correctly signed image to perform failure analysis.
Permanent debug lock	Cannot perform failure analysis.	Cannot perform failure analysis.
Secure debug lock (1)	Require debug unlock token to perform failure analysis.	Require debug unlock token and correctly signed image to perform failure analysis.

Notes:

1. Follow the procedures in Simplicity Commander to generate a valid debug unlock token for each device returned to Silicon Labs for failure analysis.
2. Secure boot enabled devices, especially with secure boot failure, may limit Silicon Labs' ability to determine the root cause of failure.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Anti-Tamper Protection Configuration and Use

Anti-Tamper Protection Configuration and Use

> Note: This section replaces *AN1247: Anti-Tamper Protection Configuration and Use* . Further updates to this application note will be provided here.

This application note describes how to program, provision, and configure the anti-tamper module. Many aspects of the anti-tamper module, including disabling the anti-tamper response when needed, are discussed.

The anti-tamper module is only available on High devices. The external tamper detect module is available on some Secure Vault Mid devices (e.g. xG27) and Secure Vault High devices (e.g. xG25B).

Key Points

- Tamper responses
- Tamper sources
- Tamper configuration
- Tamper disable
- Examples of provisioning and disabling the anti-tamper module

Copyright © 2025 Silicon Laboratories. All rights reserved.

Series 2 Device Security Features

Series 2 Device Security Features

Protecting IoT devices against security threats is central to a quality product. Silicon Labs offers several security options to help developers build secure devices, secure application software, and secure paths of communication to manage those devices. Silicon Labs' security offerings were significantly enhanced by the introduction of the Series 2 products that included a Secure Engine. The Secure Engine is a tamper-resistant component used to securely store sensitive data and keys and to execute cryptographic functions and secure services.

On Series 1 devices, the security features are implemented by the TRNG (if available) and CRYPTO peripherals.

On Series 2 devices, the security features are implemented by the Secure Engine and CRYPTOACC (if available). The Secure Engine may be hardware-based, or virtual (software-based). Throughout this document, the following abbreviations are used:

- HSE - Hardware Secure Engine
- VSE - Virtual Secure Engine
- SE - Secure Engine (either HSE or VSE)

Additional security features are provided by Secure Vault. Three levels of Secure Vault feature support are available, depending on the part and SE implementation, as reflected in the following table:

Level (1)	SE Support	Part (2)
Secure Vault High (SVH)	HSE only (HSE-SVH)	Refer to IoT Endpoint Security Fundamentals for details on supporting devices.
Secure Vault Mid (SVM)	HSE (HSE-SVM)	"
"	VSE (VSE-SVM)	"
Secure Vault Base (SVB)	N/A	"

Notes:

1. The features of different Secure Vault levels can be found in https://www.silabs.com/security.
2. IoT Endpoint Security Fundamentals.

Secure Vault Mid consists of two core security functions:

- Secure Boot: Process where the initial boot phase is executed from an immutable memory (such as ROM) and where code is authenticated before being authorized for execution.
- Secure Debug access control: The ability to lock access to the debug ports for operational security, and to securely unlock them when access is required by an authorized entity.

Secure Vault High offers additional security options:

- Secure Key Storage: Protects cryptographic keys by "wrapping" or encrypting the keys using a root key known only to the HSE-SVH.
- Anti-Tamper protection: A configurable module to protect the device against tamper attacks.
- Device authentication: Functionality that uses a secure device identity certificate along with digital signatures to verify the source or target of device communications.

A Secure Engine Manager and other tools allow users to configure and control their devices both in-house during testing and manufacturing, and after the device is in the field.

User Assistance

Copyright © 2025 Silicon Laboratories. All rights reserved.

In support of these products Silicon Labs offers whitepapers, webinars, and documentation. The following table summarizes the key security documents:

Document	Summary	Applicability
Series 2 Secure Debug	How to lock and unlock Series 2 debug access, including background information about the SE	Secure Vault Mid and High
Series 2 Secure Boot with RTSL	Describes the secure boot process on Series 2 devices using SE	Secure Vault Mid and High
AN1222: Production Programming of Series 2 Devices	How to program, provision, and configure security information using SE during device production	Secure Vault Mid and High
Anti-Tamper Protection Configuration and Use (this document)	How to program, provision, and configure the anti-tamper module	Secure Vault High
Authenticating Silicon Labs Devices using Device Certificates	How to authenticate a device using secure device certificates and signatures, at any time during the life of the product	Secure Vault High
Secure Key Storage	How to securely 'wrap' keys so they can be stored in non-volatile storage.	Secure Vault High

Key Reference

Public/Private keypairs along with other keys are used throughout Silicon Labs security implementations. Because terminology can sometimes be confusing, the following table lists the key names, their applicability, and the documentation where they are used.

Key Name	Customer Programmed	Purpose
Public Sign key (Sign Key Public)	Yes	Secure Boot binary authentication and/or OTA upgrade payload authentication
Public Command key (Command Key Public)	Yes	Secure Debug Unlock or Disable Tamper command authentication
OTA Decryption key (GBL Decryption key) aka AES-128 Key	Yes	Decrypting GBL payloads used for firmware upgrades
Attestation key aka Private Device Key	No	Device authentication for secure identity

SE Firmware

Silicon Labs strongly recommends installing the latest SE firmware on Series 2 devices to support the required security features. Refer to AN1222 for the procedure to upgrade the SE firmware and IoT Endpoint Security Fundamentals for the latest SE Firmware shipped with Series 2 devices and modules.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Introduction

Introduction

The HSE-SVH Anti-Tamper module is used to hamper or prevent both reverse engineering and re-engineering of proprietary software systems or applications.

Tamper attacks come from one or more vectors. Common attacks include voltage glitching, magnetic interference, and forced temperature adjustment. The HSE-SVH Anti-Tamper module provides fast hardware detection of external tamper signals such as case opening, glitching, and logical attacks allowing analysis and escalation up to and including bricking the device.

The anti-tamper module connects a number of hardware and software-driven tamper signals to a set of configurable hardware and software responses. This can be used to program the device to automatically respond to external events that could signal that someone is trying to tamper with the device, and very rapidly remove secrets stored in the HSE.

The available tamper signals range from signals based on failed authentication and secure boot to specialized glitch detectors. When any of these signals fire, the tamper block can be configured to trigger several different responses, ranging from triggering an interrupt to erasing the One-Time-Programmable (OTP) memory, removing all HSE secrets and resulting in a permanently destroyed device.

Silicon Labs provides Custom Part Manufacturing Service (CPMS) to protect the users' privacy by configuring the most effective tamper detection features at the Silicon Labs factory. For more information about CPMS, see the Custom Part Manufacturing Service User's Guide.

Some SVM devices (e.g. xG25A and xG27) and SVH devices (e.g. xG25B) feature an External Tamper Detect module which is used to detect signals such as case opening. The ETAMPDET signal on SVH devices is routed to the SE as an Anti-Tamper module tamper source, in addition to being a stand-alone module. For more information about ETAMPDET operation, refer to the device reference manual. Examples demonstrating how to use ETAMPDET can be found on the Silicon Labs peripheral_example github repository.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Secure Engine Manager

Secure Engine Manager

The Secure Engine Manager provides thread-safe APIs for the SE's mailbox interface. The SE Manager APIs related to tamper operations are listed in the following table.

For the SE's mailbox interface, see section *Secure Engine Subsystem* in Series 2 Secure Debug.

SE Manager API	Usage
sl_se_init_otp	Initialize SE OTP configuration (including tamper configuration on HSE-SVH devices).
sl_se_read_otp	Read SE OTP configuration (including tamper configuration on HSE-SVH devices).
sl_se_init_otp_key	Used during device initialization to upload the Public Command Key.
sl_se_read_pubkey	Read the stored Public Command Key.
sl_se_get_serialnumber	Read out the serial number (16 bytes) of the HSE device.
sl_se_get_challenge	Read out the current challenge value (16 bytes) for tamper disable.
sl_se_roll_challenge	Used to roll the current challenge value (16 bytes) to invalidate the Disable Tamper Token.
sl_se_disable_tamper	Temporarily disable tamper configuration using the Disable Tamper Token.
sl_se_get_status	Read the current HSE status (including recorded tamper status on HSE-SVH devices).
sl_se_get_reset_cause	Read the EMU->RSTCAUSE register from HSE devices after a tamper reset.
sl_se_get_tamper_reset_cause	Read the cached value of the EMU->TAMPERRSTCAUSE register after a tamper reset.
sl_se_enter_active_mode	Force the SE to remain active to enable the detection of glitch tamper events on the host Cortex-M33 core.(see fourth note below)
sl_se_exit_active_mode	Exit active mode and allow the SE to sleep when not performing operations. This will prevent the detection of glitch tamper events when the SE is sleeping. This API should only be used if active mode was entered by calling sl_se_enter_active_mode. If active mode is set through a DCI command, it can only be disabled through a DCI command. (see fourth note below)

Notes:

- The `sl_se_get_reset_cause` is only available on EFR32xG21B devices. The `EMU->RSTCAUSE` register can be directly read on other HSE-SVH devices.
- The `sl_se_get_tamper_reset_cause` is unavailable on EFR32xG21B devices, and SE firmware ≥ v2.2.1 is required.
- The SE Manager API document can be found at https://docs.silabs.com/gecko-platform/latest/service/api/group-sl-se-manager.
- Does not apply to EFR32MG21B parts.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Tamper Responses

Tamper Responses

A tamper source can lead to a series of different autonomous responses from the HSE. These responses are listed in the following table.

Level	Response	Description
0	Ignore	No action is taken
1	Interrupt	Triggers the SETAMPERHOST interrupt on the host
2	Filter	Increases a counter in the tamper filter
4	Reset	Resets the device
7	Erase OTP	Erases the device's OTP configuration

Notes:

1. Level 3, 5, and 6 are reserved.
2. These responses are cumulative:
   - If a filter response is triggered, it will also trigger an interrupt.
   - If a reset response is triggered, it will supersede the interrupt. The filter counter and interrupt flag are clear at reset.
   - If an erase OTP response is triggered, it will erase the OTP and reset the device. The device will fail to boot and become unusable.

Interrupt

If a tamper source is configured to respond with the interrupt response or higher (≥ level 1), the `SETAMPERHOST` interrupt line to the host Cortex-M33 will be pulsed and make the NVIC trigger the corresponding interrupt handler (`SETAMPERHOST_IRQHandler`).

After the interrupt has been handled, the tamper status can be found by reading the HSE status (using `sl_se_get_status` in the SE Manager), which contains a list of all the tamper sources that have been triggered since the last time the status was read. Reading HSE status clears the registered tamper sources.

> **Note**: Enabling the `SEMAILBOXHOST` clock for the tamper source is required to trigger the `SETAMPERHOST` interrupt in most HSE-SVH devices. EFR32xG21B does not require this.

Filter

The HSE has a filter to debounce spurious tamper events. The filter has a counter that is periodically reset. If a tamper source is configured to the filter response (level 2), when it is triggered, the counter is increased. If the counter value reaches a configurable threshold, the `Filter counter` tamper source (number 1) is triggered, which can configure to lead to any other responses (1, 4, or even 7).

Only a single shared filter counter is available, so the cumulative triggering of all tamper sources configured to the filter level will increase the same counter. The filter can be programmed to use one of the trigger thresholds and reset periods provided below. The filter counter is zero upon a tamper or normal reset.

Filter Trigger Threshold

- Value (n): 0 to 7
- Filter Trigger Threshold: $256/2^n$ (256 to 2)

Copyright © 2025 Silicon Laboratories. All rights reserved.

Filter Reset Period

- Value (n): 0 to 31
- Filter Reset Period: 32 ms * 2^n (32 ms to ~795.4 days)

Example Filter Configuration

For example, consider a device with a Filter Trigger Threshold of 3 and Filter Reset Period of 5. If that device detects 32 (256/2^3) Filter response events in 1.024 seconds (32 ms * 2^5), the `Filter counter` tamper source (number 1) will trigger.

Reset

The reset response resets the HSE and Cortex-M33. After a tamper reset, the last reset cause can be directly read from `EMU->RSTCAUSE` register or using `sl_se_get_rstcause` in the SE Manager. In cases where the reset was caused by a tamper response, the source of the tamper can be determined by calling `sl_se_get_tamper_reset_cause` in the SE Manager. (Note that this API is not available for EFR32xG21B-based parts). See Table Tamper Sources on Other HSE-SVH Devices for the list of tamper sources. Tamper reset occurs when the HSE sends a request to the Cortex-M33's EMU, which issues a hard reset.

If a tamper reset is triggered during boot, this can lead to a boot loop. To debug such a scenario, the HSE has a tamper reset counter and enters diagnostic mode if the counter reaches a programmable threshold. Users can issue a non-tamper reset to clear the tamper reset counter before the programmable threshold is reached.

In diagnostic mode, the Cortex-M33 is held in reset and only DCI commands are available. The device will remain in diagnostic mode until a power-on or pin reset occurs.

For more information on the SE's DCI, see section *Secure Engine Subsystem* in Series 2 Secure Debug.

Erase OTP

The Erase OTP response is the strongest reaction the HSE can take, and it will make the device and all wrapped secrets unrecoverable. After this response, the device will no longer be able to boot or connect to a debugger.

This response should typically only be used in situations where the device believes that it is under an actual attack, for instance through the detection of several voltage or digital glitches in a short time window.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Tamper Sources

Tamper Sources

The following tables list the available tamper sources and the default level on the EFR32xG21B and other HSE-SVH devices. The tamper sources with the default level higher than 0 (`Ignore`) are always in effect even if the user does not initialize the tamper configuration in HSE OTP. Users can keep or escalate the default tamper responses (≥ 0 for `Ignore` and ≥ 4 for `Reset`) of any sources when initially configuring the part.

Table: Tamper Sources on the EFR32xG21B Devices

Type	Number	Name	Description	Default Level
SE Hardware	0	Reserved	-	-
"	1	Filter counter	Filter counter reached the configured threshold value	0 (Ignore)
"	2	SE watchdog	Internal SE watchdog expired	4 (Reset)
"	3	Reserved	-	-
"	4	SE RAM CRC	A 2-bit, non-correctable error in the SE RAM has occurred.	4 (Reset)
"	5	SE hard fault	The SE core has encountered a hard fault exception indicating that an invalid memory access was attempted.	4 (Reset)
"	6	Reserved	-	-
SE Software	7	SE software assertion	SE firmware has triggered an assertion, indicating that one of several sanity checks has failed and that normal operation cannot continue without a reset.	4 (Reset)
"	8	SE secure boot	Secure boot of SE firmware failed	4 (Reset)
"	9	User secure boot	Secure boot of host firmware failed	0 (Ignore)
"	10	Mailbox authorization	Unauthorized command received over the Mailbox interface. This can be triggered by either (1) an incorrectly signed debug unlock or tamper disable token or (2) attempting to export a non-exportable key.	0 (Ignore)
"	11	DCI authorization	Unauthorized command received over the DCI interface. This can be triggered by either (1) an incorrectly signed debug un-lock or tamper disable token or (2) attempting to export a non-exportable key.	0 (Ignore)
"	12	OTP read	OTP or flash content could not be properly authenticated.	4 (Reset)
"	13	Reserved	-	-
"	14	Self test	A check of the integrity of the SE's internal storage failed during boot up.	4 (Reset)
"	15	TRNG monitor	The TRNG monitor performs a number of tests on the collected entropy data. If any of these tests fail, this tamper source is triggered.	0 (Ignore)
Hardware	16 - 23	PRS0 - 7 [1]	PRS consumer for SE Tamper 0 - 7 asserted	0 (Ignore)

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Number	Name	Description	Default Level
"	24	Decouple BOD [1]	Decouple Brown-Out-Detector threshold alert	4 (Reset)
"	25	Temperature sensor [1]	SE temperature is monitored to be within 5 degrees C of the limits for the device. If the limit is exceeded, this tamper source will be triggered.	0 (Ignore)
"	26	Voltage glitch falling	Voltage glitch detector detected a falling glitch	0 (Ignore)
"	27	Voltage glitch rising	Voltage glitch detector detected a rising glitch	0 (Ignore)
"	28	Secure lock	This tamper source indicates that the guarding mechanism (comparing the locks with their logical complement) of the debug port locks has failed	4 (Reset)
"	29	SE debug	Debug access to SE	0 (Ignore)
"	30	Digital glitch	Digital glitch detector detected an event	0 (Ignore)
"	31	SE ICACHE	The SE's instruction cache uses a checksum to verify the integrity of the data. This tamper source is triggered if the checksum is invalid.	4 (Reset)

Table: Tamper Sources on Other HSE-SVH Devices

Type	Number	Name	Description	Default Level
SE Hardware	0	Reserved	-	-
"	1	Filter counter	Filter counter reached the configured threshold value	0 (Ignore)
"	2	SE watchdog	Internal SE watchdog expired	4 (Reset)
"	3	Reserved	-	-
"	4	SE RAM ECC2	A 2-bit, non-correctable error in the SE RAM has occurred.	4 (Reset)
"	5	SE hard fault	The SE core has encountered a hard fault exception indicating that an invalid memory access was attempted.	4 (Reset)
"	6	Reserved	-	-
SE Software	7	SE software assertion	SE firmware has triggered an assertion, indicating that one of several sanity checks has failed and that normal operation cannot continue without a reset.	4 (Reset)
"	8	SE secure boot	Secure boot of SE firmware failed	4 (Reset)
"	9	User secure boot	Secure boot of host firmware failed	0 (Ignore)
"	10	Mailbox authorization	Unauthorized command received over the Mailbox interface. This can be triggered by either (1) an incorrectly signed debug unlock or tamper disable token or (2) attempting to export a non-exportable key.	0 (Ignore)
"	11	DCI authorization	Unauthorized command received over the DCI interface. This can be triggered by either (1) an incorrectly signed debug unlock or tamper disable token or (2) attempting to export a non-exportable key.	0 (Ignore)
"	12	OTP read	OTP or flash content could not be properly authenticated	4 (Reset)
"	13	Reserved	-	-

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Number	Name	Description	Default Level
"	14	Self test	A check of the integrity of the SE's internal storage failed during boot up.	4 (Reset)
"	15	TRNG monitor	The TRNG monitor performs a number of tests on the collected entropy data. If any of these tests fail, this tamper source is triggered.	0 (Ignore)
Hardware	16	Secure lock	This tamper source indicates that the guarding mechanism (comparing the locks with their logical complement) of the debug port locks has failed.	4 (Reset)
"	17	Digital glitch	Digital Glitch detector detected an event	0 (Ignore)
"	18	Voltage glitch	Voltage Glitch Detector detected an event	0 (Ignore)
"	19	SE ICACHE	The SE's instruction cache uses a checksum to verify the integrity of the data. This tamper source is triggered if the checksum is invalid.	4 (Reset)
"	20	SE RAM ECC1	SE RAM 1-bit ECC error occurred	0 (Ignore)
"	21	BOD [1]	Brown-Out-Detector threshold alert	4 (Reset)
"	22	Temperature sensor [1]	SE temperature is monitored to be within 5 degrees C of the limits for the device. If the limit is exceeded, this tamper source will be triggered.	0 (Ignore)
"	23	DPLL fall	DPLL lock failed low	0 (Ignore)
"	24	DPLL rise	DPLL lock failed high	0 (Ignore)
"	25	PRS0 or ETAMPDET	PRS consumer for SE Tamper 25 or ETAMPDET asserted.	0 (Ignore)
"	26 - 31	PRS1 - 6 or PRS0 - 5[1]	PRS consumer for SE Tamper 26 - 31 asserted	0 (Ignore)

Notes:

- [1] These tamper sources are available down to EM2. Other sources are available in EM1 and above.
- In EFR32xG21B devices, hardware tamper sources 24 to 27 can operate down to Energy Mode 3 (EM3), whereas other hardware tamper sources (16 - 23 and 28 - 31) can be active down to Energy Mode 1 (EM1).
- In other HSE-SVH devices, tamper sources 25 to 31 are used for External Tamper Detect (ETAMPDET) if present and PRS consumers. Devices with ETAMPDET (e.g. EFR32xG25B) will have 6 PRS consumers (26 to 31) and devices without ETAMPDET will have 7 PRS consumers (25 to 31).
- The **ETAMPDET** source gets triggered when any of the ETAMPDET channels are asserted.
- User configuration or tamper disable cannot reduce the tamper response below the default Level.
- The **User secure boot** source gets triggered if secure boot is enabled and host image verification fails. It is likely to put the device in the boot loop if users escalate the tamper response of this source to 4 (Reset).
- The **Mailbox and DCI authorizations** get triggered whenever one of the following conditions has occurred. The mailbox returns `SE_RESPONSE_AUTHORIZATION_ERROR`, and DCI returns `AUTH_ERROR = 2`.
    1. A mailbox or DCI command tries to exercise a key that it is not allowed to use (e.g., trying to export a non-exportable key).
    2. A secure debug access or tamper disable request over the mailbox or DCI is invalidly signed.
    3. A malformed HSE firmware upgrade over the mailbox or DCI is attempted.
- The **OTP read** gets triggered if there is an issue when decrypting and authenticating settings in HSE OTP or flash.
- The HSE has redundancy built into the locking mechanism, and the **Secure lock** source is used to detect errors in that redundancy.
- PRS inputs can allow user applications to implement additional tamper sources and feed them into the tamper response mechanism. The **PRS** tamper sources are under the control of the user application and could be reconfigured or disabled if the user application is compromised.
- The **Temperature sensor** source is not completely accurate and is generally only suitable for systems that expect to stay well within the specified temperature range. Users requiring a tighter temperature limit can implement their temperature

Copyright © 2025 Silicon Laboratories. All rights reserved.

monitor and provide the results as a tamper source via PRS.

- On EFR32xG23B and later devices, the default behavior is to detect tamper events only when the SE core is active. To detect tamper events when the SE is not performing operations, call `sl_se_enter_active_mode()`. This prevents the SE from sleeping and will result in higher current draw.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Anti-Tamper Configuration

Anti-Tamper Configuration

The user can provision the anti-tamper configuration in HSE OTP detailed in the following table through `sl_se_init_otp` in the SE Manager. Tamper configurations must be programmed with secure boot settings and are immutable once written.

For more information on enabling the OTP tamper configuration along with the secure boot settings, see *Enabling Secure Boot and Tamper Configuration* in AN1222: Production Programming of Series 2 Devices.

Setting	Description
Tamper response levels	A user response level for each tamper source (1)
Filter settings	The tamper filter counter has two settings: trigger threshold and reset period
Digital Glitch Detector Always On	Bit 1 of tamper flag: 0 – Digital glitch detector runs only when the HSE is executing a command; 1 – Digital glitch detector runs even when the HSE is not performing any operations (note that this leads to increased energy consumption)
Keep Tamper Alive During Sleep(2)	Bit 2 of tamper flag: 0 – The tamper module stops running in sleep mode; 1 – The tamper module keeps running in sleep mode (down to EM3)
Reset threshold	The number of consecutive tamper resets (up to 255) before the part enters diagnostic mode (3)

Notes:

1. The effective response of a tamper source is the maximum value between the default level and user level (`Active level = MAX(default level, user level)`). If the user sets the response of a tamper source to a level lower than the default level, the setting will still be saved to HSE OTP but does not take any effect. The HSE returns the user levels instead of active levels of all tamper sources when reading back (`sl_se_read_otp`) the tamper configuration from the HSE OTP.
2. This flag is not available on EFR32xG21B devices.
3. If the threshold is set to 0, the part will never enter the diagnostic mode due to tamper reset.

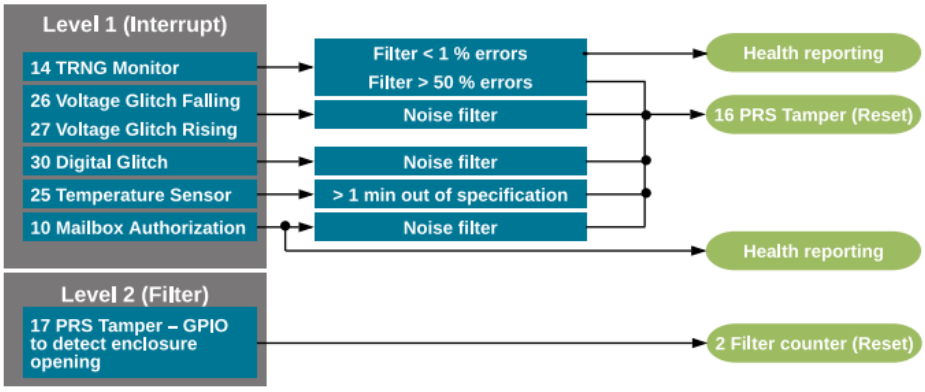
Copyright © 2025 Silicon Laboratories. All rights reserved.

Usage Example

Usage Example

Several of the available tamper sources report internal HSE errors. A number of these sources are configured to reset the device (level 4) by default. Custom handling of internal and external tamper sources (default level 0) can be configured to trigger an interrupt (level 1) on the Cortex-M33 or trigger an interrupt and increase a counter in the tamper filter (level 2) as in the following figure for EFR32xG21B devices.

Figure: Custom Handling of Tamper Sources (EFR32xG21B Devices)



> **Note**: The actions for level 1 on the right side are implemented by the tamper interrupt handler.

Usage example highlights:

- The response of the TRNG monitor depends on the failure rate due to lack of entropy.
- The voltage and digital glitch detectors can see spurious activations. They should typically not be used to drive a high-level tamper response directly. Instead, they should feed their signals into a tamper interrupt, which activates a high-level action (e.g., Reset in this example) through PRS tamper if a certain number of detections (noise filter) occur in a short time window.
- The operating conditions decide the time out of the specification filter for the temperature sensor. For some systems, any time out of specification should trigger a reset.
- Mailbox authorization is handled similarly for voltage and digital glitch detectors.
- A PRS tamper implements a high-level response for a tamper interrupt, which issues a tamper reset (level 4) to prevent or slow further attacks.
- In extreme cases, if the system identifies an attack with high confidence, a PRS tamper can be configured as Erase OTP (level 7) to brick the part and prevent further attacks. This is recommended only when the destruction of parts is acceptable and where high confidence of an attack can be achieved.
- Another PRS tamper detects enclosure opening from GPIO. This source feeds into the tamper filter counter (level 2), which will trigger an interrupt (cumulative effect) and activate a Filter counter (number 1) response (Reset in this example) if the filter counter reaches the trigger threshold within the filter reset period. This filter counter response approach is less flexible than the interrupt response approach since the trigger threshold and filter reset period are one-time programmable.

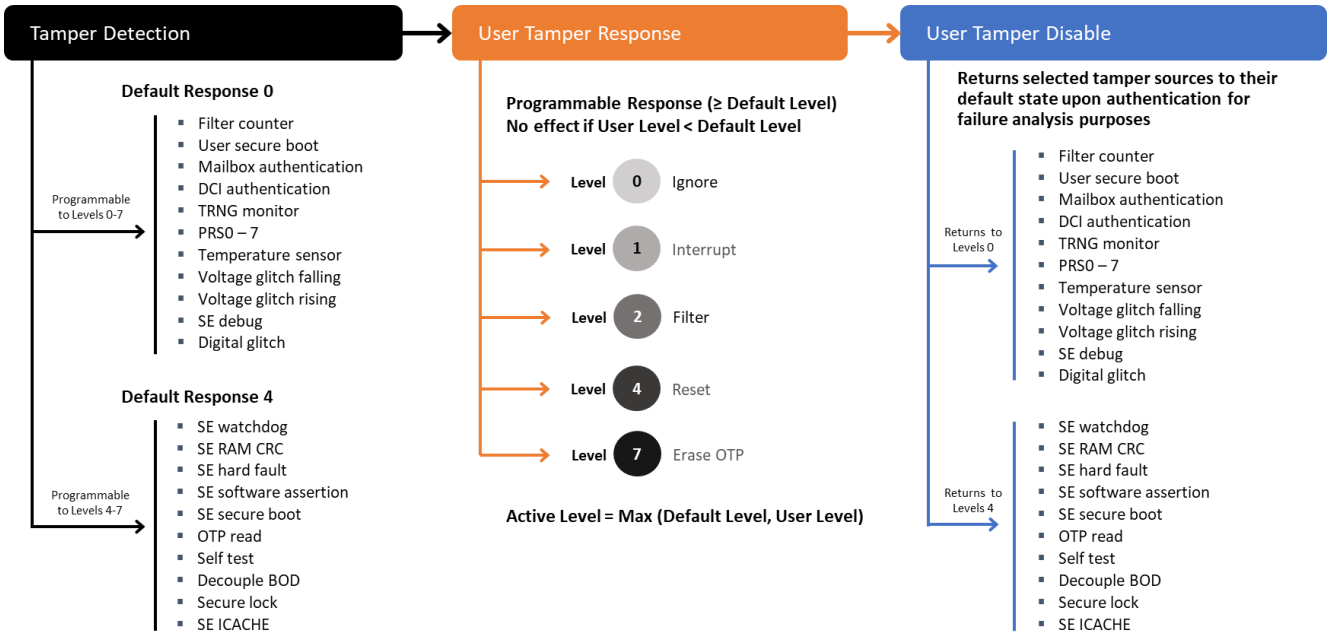
Copyright © 2025 Silicon Laboratories. All rights reserved.

Tamper Disable

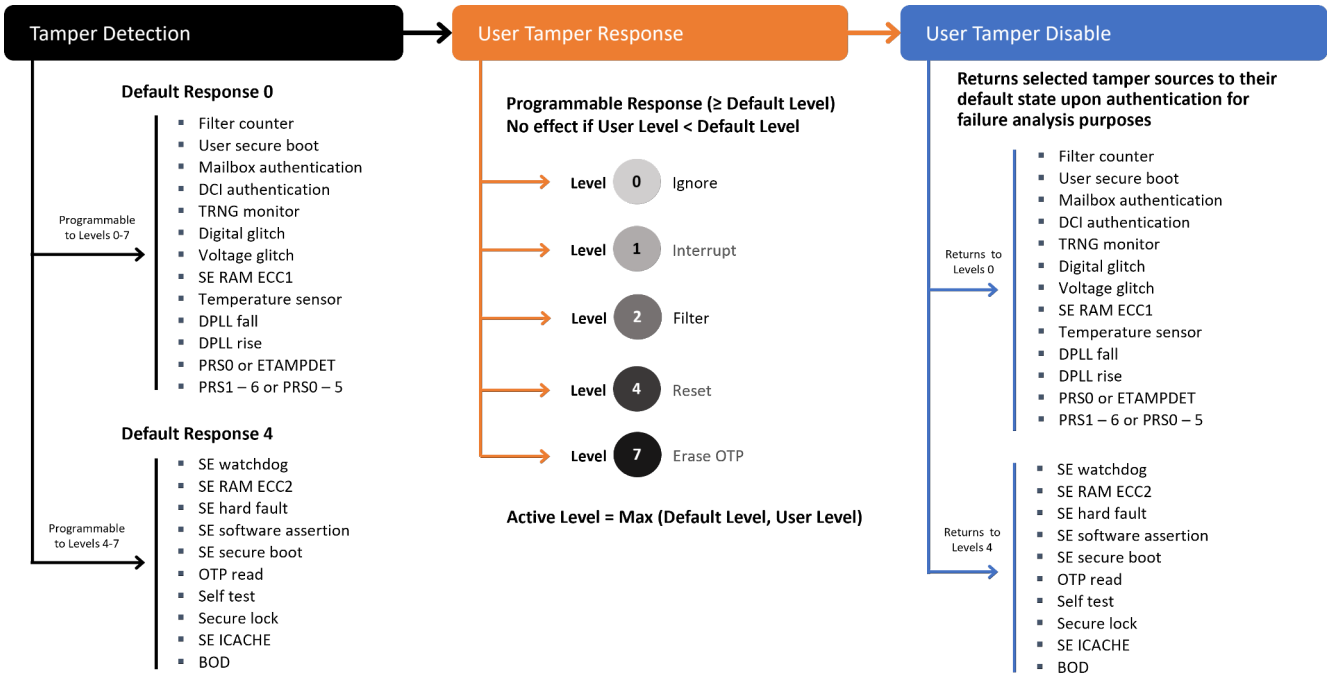
Tamper Disable

For diagnostic purposes, it may be necessary to disable the tamper response. For example, if a user has configured the part to Erase OTP on external tamper detection, disabling the tamper response is required to open the unit and perform failure analysis or field service activities.

After the tamper configuration has been initialized, users can temporarily restore the tamper response to default for a set of tamper sources via a Disable Tamper Token authenticated against the Public Command Key in HSE OTP (similar to secure debug unlock). This is only possible if the Public Command Key has been provisioned in the device.

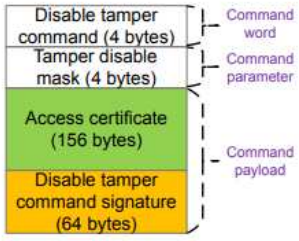


Copyright © 2025 Silicon Laboratories. All rights reserved.



Disable Tamper Token

The elements of the Disable Tamper Token are described in the following figures and table.



Element	Value	Description
Disable tamper command	0xfd020001	The command word of the Disable Tamper Token.
Tamper disable mask	Device-dependent	The command parameter of the Disable Tamper Token.
Access certificate (1)	Device-dependent	See section Access Certificate.
Disable tamper command signature (1)	Device-dependent	See section Challenge Response.

Note:

1. The disable tamper command payload consists of an access certificate and a disable tamper command signature.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Tamper Disable Mask																															
Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	Tamper source 31	Tamper source 30	Tamper source 29	Tamper source 28	Tamper source 27	Tamper source 26	Tamper source 25	Tamper source 24	Tamper source 23	Tamper source 22	Tamper source 21	Tamper source 20	Tamper source 19	Tamper source 18	Tamper source 17	Tamper source 16	Tamper source 15	Tamper source 14	Tamper source 13	Tamper source 12	Tamper source 11	Tamper source 10	Tamper source 9	Tamper source 8	Tamper source 7	Tamper source 6	Tamper source 5	Tamper source 4	Tamper source 3	Tamper source 2	Tamper source 1	Tamper source 0

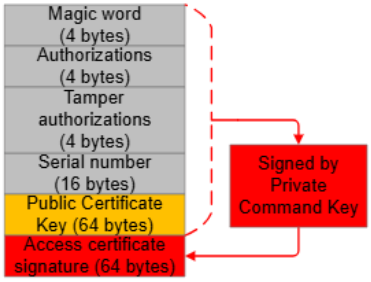
> **Note**: Set bit to restore the default response of the corresponding tamper source.

The Disable Tamper Token temporarily reverts all masked tamper sources in the figure above to the hard-coded configuration (Figure 8.1 Tamper Disable on the EFR32xG21B Devices on page 13 and Figure 8.2 Tamper Disable on Other HSE-SVH Devices on page 13).

The Disable Tamper Token can only restore the escalated user-level configuration to default. It cannot degrade the default level of a tamper source.

Access Certificate

The elements of the access certificate are described in the following figure and table.



Element	Value	Description
Magic word	0xe5ecce01	A constant value used to identify the access certificate.
Authorizations	0x0000003e (1)	A value used to authorize which bit in the debug mode request can be enabled for secure debug.
Tamper Authorizations	0xffffffb6 (2)	A value used to authorize which bit in the tamper disable mask can be set to disable the tamper response.
Serial number	Device-dependent	A number used to compare against the on-chip serial number for secure debug or tamper disable.
Public Certificate Key (3)	Device-dependent	The public key corresponding to the Private Certificate Key (3) used to generate the signature (ECDSA-P256-SHA256) in a challenge response.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Element	Value	Description
Access certificate signature	Device-dependent	All the content above is signed (ECDSA-P256-SHA256) by the Private Command Key corresponding to the Public Command Key in the HSE OTP.

Notes:

1. The value allows all debug options to be reset for secure debug. Note that the commands for debug unlock and tamper disable are separate, so the secure debug lock will not be disabled when issuing a tamper disable command.
2. Value that sets available bits in the tamper disable mask for tamper disable.
3. The Private/Public Certificate Key is a randomly generated key pair. It can be ephemeral or retainable.

The Private Certificate Key can be used repeatedly to generate the signature in a challenge response on one device until the Private/Public Certificate Key pair is discarded. This can reduce the frequency of access to the Private Command Key, allowing more restrictive access control on that key.

For more information about secure debug, see Series 2 Secure Debug.

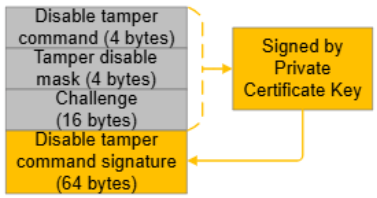
Tamper Authorizations																															
Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	Tamper disable mask 31	Tamper disable mask 30	Tamper disable mask 29	Tamper disable mask 28	Tamper disable mask 27	Tamper disable mask 26	Tamper disable mask 25	Tamper disable mask 24	Tamper disable mask 23	Tamper disable mask 22	Tamper disable mask 21	Tamper disable mask 20	Tamper disable mask 19	Tamper disable mask 18	Tamper disable mask 17	Tamper disable mask 16	Tamper disable mask 15	Tamper disable mask 14	Tamper disable mask 13	Tamper disable mask 12	Tamper disable mask 11	Tamper disable mask 10	Tamper disable mask 9	Tamper disable mask 8	Tamper disable mask 7	Tamper disable mask 6	Tamper disable mask 5	Tamper disable mask 4	Tamper disable mask 3	Tamper disable mask 2	Tamper disable mask 1	Tamper disable mask 0

Notes:

- Set the bit to enable the corresponding bit in the tamper disable mask.
- The Disable Tamper Token will restore the default response of the corresponding tamper source if the same bit is set in the tamper disable mask and tamper authorizations.

Challenge Response

The elements of the challenge response are described in the following figure and table.



Element	Value	Description
Disable tamper command	0xfd020001	The command word of Disable Tamper Token.
Tamper disable mask	Device-dependent	The command parameter of Disable Tamper Token.
Challenge	Device-dependent (1)	A random value generated by the HSE.

Copyright © 2025 Silicon Laboratories. All rights reserved.

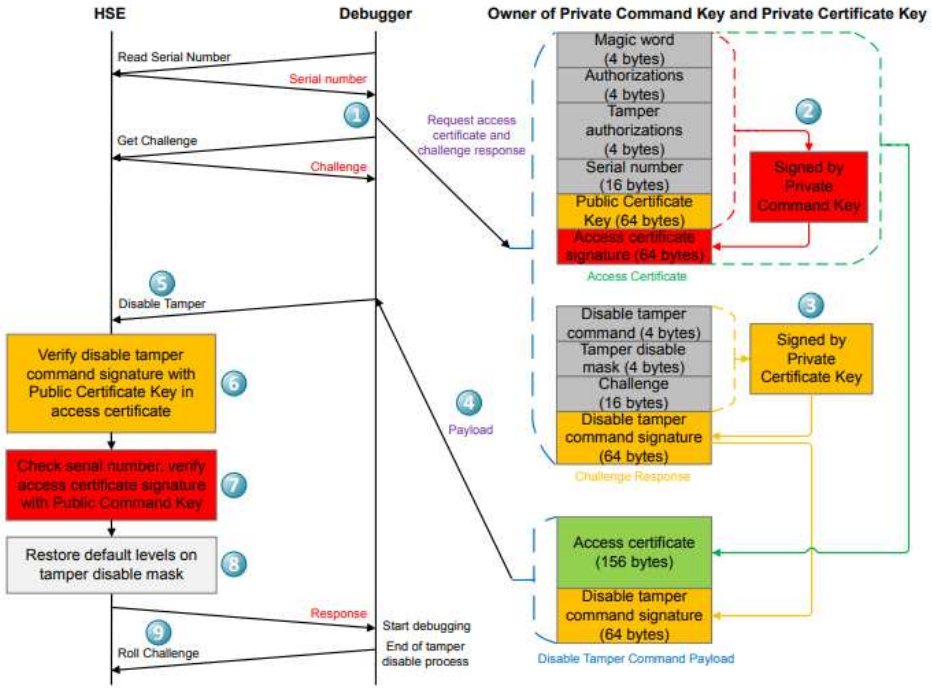
Element	Value	Description
Disable tamper command signature	Device-dependent (2)	All the content above is signed (ECDSA-P256-SHA256) by the Private Certificate Key corresponding to the Public Certificate Key in the access certificate.

Notes:

1. The challenge remains unchanged until it is updated to a new random value by rolling the challenge. The Private Certificate Key can be reused for signing when device challenge is refreshed.
2. This signature is the final argument of the Disable Tamper Token.

Tamper Disable Flow

The tamper disable flow is described in the following figure.



1. Get the serial number and challenge from the HSE.
2. Generate the access certificate with device serial number.
3. Generate the challenge response with device challenge.
4. Generate the disable tamper command payload with access certificate and disable tamper command signature.
5. Send the Disable Tamper Token to the HSE.
6. Verify the disable tamper command signature using the Public Certificate Key in the access certificate.
7. Verify the serial number and the access certificate signature using the on-chip serial number and Public Command Key in the HSE OTP.
8. Restore default levels on tamper disable mask until the next power-on or pin reset.
9. Roll the challenge to invalidate the current Disable Tamper Token.

> **Note**: Refer to the Simplicity Commander example for details on how to follow this flow using Simplicity Commander.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Examples

Examples

Overview

The examples for HSE-SVH Anti-Tamper module are described in the following table.

Example	Device (Radio Board)	HSE Firmware	Tool
Provision Tamper configuration	EFR32MG21B010F1024IM32 (BRD4181C)	Version 1.2.9	SE Manager
Provision Public Command Key & Tamper configuration	EFR32MG21B010F1024IM32 (BRD4181C)	Version 1.2.9	Simplicity Commander
"	EFR32MG21B010F1024IM32 (BRD4181C)	Version 1.2.9	Simplicity Studio 5
Tamper disable and Roll challenge	EFR32MG21B010F1024IM32 (BRD4181C)	Version 1.2.9	SE Manager
"	EFR32MG21B010F1024IM32 (BRD4181C)	Version 1.2.9	Simplicity Commander
Roll challenge	EFR32MG21B010F1024IM32 (BRD4181C)	Version 1.2.9	Simplicity Studio 5

> **Note**: Unless specified in the example, these examples can be applied to other HSE-SVH devices.

Using a Platform Example

Simplicity Studio 5 includes the SE Manager platform example for tamper. This application note uses platform examples of GSDK v4.1.0. The console output may be different on the other version of GSDK.

Refer to the corresponding readme file for details about each SE Manager platform example. This file also includes the procedures to create the project and run the example.

Using Simplicity Commander

1. This application note uses Simplicity Commander v1.14.6. The procedures and console output may be different on other versions of Simplicity Commander. The latest version of Simplicity Commander can be downloaded from .

```
commander --version
```

```
Simplicity Commander 1v14p6b1289

JLink DLL version: 7.70d
Qt 5.12.10 Copyright (C) 2017 The Qt Company Ltd.
EMDLL Version: 0v18p9b677
mbed TLS version: 2.16.6

DONE
```

2. The Simplicity Commander's Command Line Interface (CLI) is invoked by `commander.exe` in the Simplicity Commander folder. The location for Simplicity Studio 5 in Windows is `C:\SiliconLabs\SimplicityStudio\v5\developer\adapter_packs\commander`. For ease of use, it is highly recommended to add the path of `commander.exe` to the system `PATH` in Windows.

Copyright © 2025 Silicon Laboratories. All rights reserved.

If more than one Wireless Starter Kit (WSTK) is connected via USB, the target WSTK must be specified using the `--serialno \<J-Link serial number>` option.

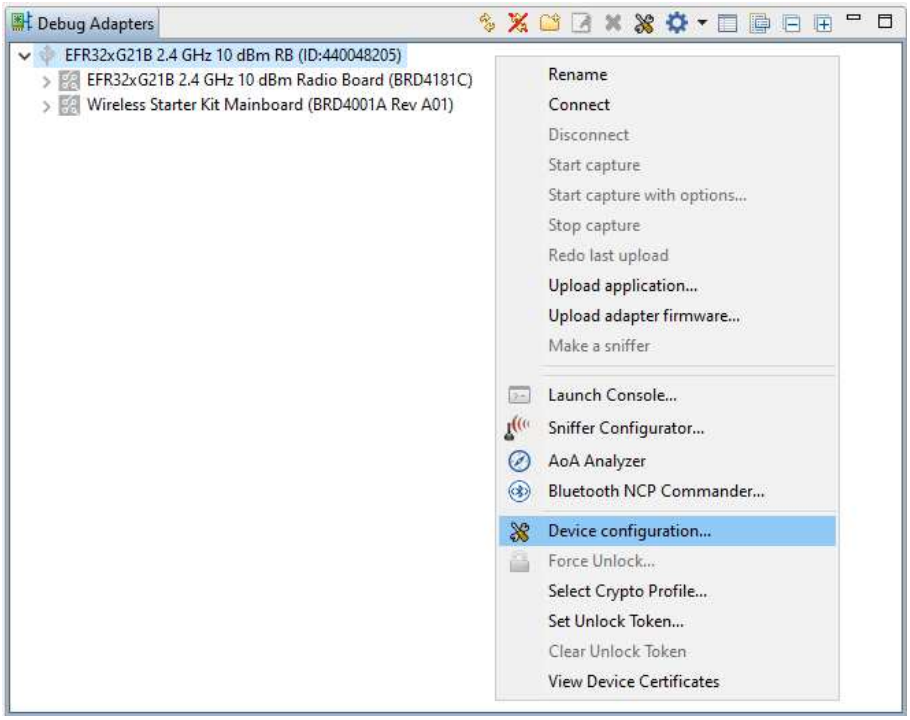
4. If the WSTK is in debug mode OUT, the target device must be specified using the `--device \<device name>` option.

For more information about Simplicity Commander, see the Simplicity Commander Reference Guide.

Using Simplicity Studio

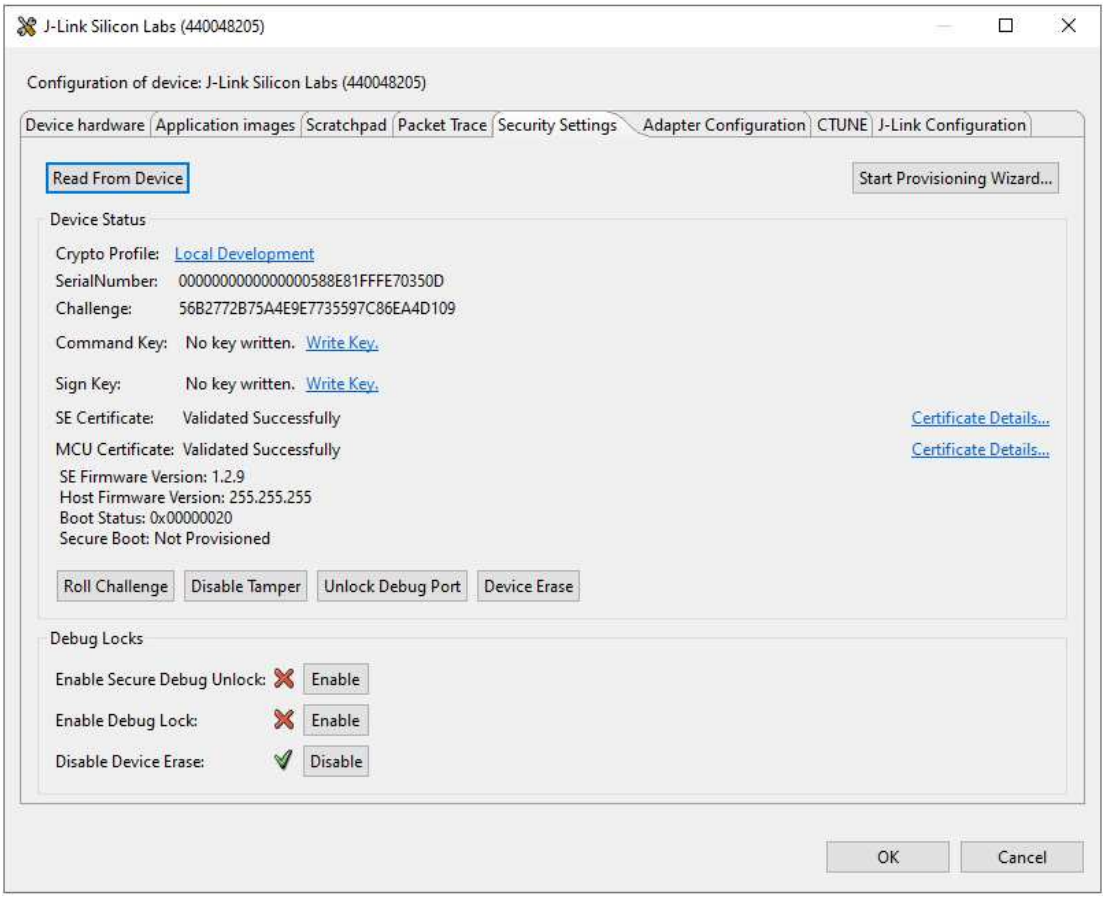
The security operations are performed in the Security Settings of Simplicity Studio. This application note uses Simplicity Studio v5.4.0.0. The procedures and pictures may be different on the other version of Simplicity Studio 5.

1. Right-click the selected debug adapter **RB (ID:J-Link serial number)** to display the context menu.



2. Click **Device configuration...** to open the **Configuration of device: J-Link Silicon Labs (serial number)** dialog box. Click the **Security Settings** tab to get the selected device configuration.

Copyright © 2025 Silicon Laboratories. All rights reserved.



Using an External Tool

The tamper disable example uses **OpenSSL** to sign the access certificate and challenge response. The Windows version of OpenSSL can be downloaded from Shining Light Productions. This application note uses OpenSSL Version 1.1.1h (Win64).

```
openssl version
```

```
OpenSSL 1.1.1h  22 Sep 2020
```

The OpenSSL's Command Line Interface (CLI) is invoked by `openssl.exe` in the OpenSSL folder. The location in Windows (Win64) is `C:\Program Files\OpenSSL-Win64\bin` . For ease of use, it is highly recommended to add the path of `openssl.exe` to the system `PATH` in Windows.

Provision Public Command Key and Tamper Configuration

The Public Command Key pair can be generated from the "unsafe" private key delivered with Simplicity Studio, by Simplicity Commander, or by a Hardware Security Module (HSM). Using an HSM is recommended for production systems.

Generated from "Unsafe" Key

External tools such as openssl can be used to generate a public key from the reference private key provided in Simplicity Studio. Note that this private key is well known and should not be used in production devices.

Run the `openssl ec` command to generate the Public Command Key from the Private Command Key.

```
openssl ec -in /c/SiliconLabs/SimplicityStudio/v5/developer/adapter_packs/secmgr/scripts/offline/cmd-unsafe-privkey.pem -pubout -out cmd-unsafe-pubkey.pem
```

Generated Using Simplicity Commander

Copyright © 2025 Silicon Laboratories. All rights reserved.

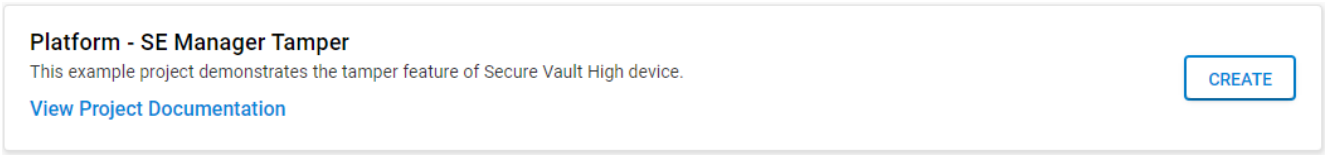
Run the util genkey command to generate the Public Command Key pair (command_key.pem and command_pubkey.pem) and Public Command Key token file (command_pubkey.txt).

```
commander util genkey --type ecc-p256 --privkey command_key.pem --pubkey command_pubkey.pem --tokenfile command_pubkey.txt
```

```
Generating ECC P256 key pair...
Writing private key file in PEM format to command_key.pem
Writing public key file in PEM format to command_pubkey.pem
Writing EC tokens to command_pubkey.txt...
DONE
```

SE Manager - Tamper Platform Example

Click the `View Project Documentation` link to open the readme file for instructions on creating the project and running the example.



1. Press `ENTER` two times to program the secure boot and tamper configuration to the HSE OTP of an uninitialized device.

```
SE Manager Tamper Example - Core running at 38000 kHz.
 . SE manager initialization... SL_STATUS_OK (cycles: 7 time: 0 us)
 . Read EMU RSTCAUSE register... SL_STATUS_OK (cycles: 3728 time: 98 us)
 + The EMU RSTCAUSE register (MSB..LSB): 00000043
 . Read SE OTP configuration... SL_STATUS_NOT_INITIALIZED (cycles: 7487 time: 197 us)
 + Cannot read SE OTP configuration.
 + Press ENTER to initialize SE OTP for tamper configuration or press SPACE to abort.
 + Warning: The OTP configuration cannot be changed once written!
 + Press ENTER to confirm or press SPACE to abort if you are not sure.
 + Initialize SE OTP for tamper configuration... SL_STATUS_OK (cycles: 267256 time: 7033 us)
 + Issue a power-on or pin reset to activate the new tamper configuration.
 . SE manager deinitialization... SL_STATUS_OK (cycles: 9 time: 0 us)
```

> **Note**: This example does not enable the secure boot.

2. Press the **RESET** button on the WSTK to restart the program. It will display the current tamper configuration of the device.

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
SE Manager Tamper Example - Core running at 38000 kHz.
 . SE manager initialization... SL_STATUS_OK (cycles: 10 time: 0 us)
 . Read EMU RSTCAUSE register... SL_STATUS_OK (cycles: 3736 time: 98 us)
 + The EMU RSTCAUSE register (MSB..LSB): 00000043
 . Read SE OTP configuration... SL_STATUS_OK (cycles: 7174 time: 188 us)
 + Secure boot: Disabled
 + Tamper source level
   Filter counter       : 1
   SE watchdog          : 4
   SE RAM CRC           : 4
   SE hard fault        : 4
   SE software assertion : 4
   SE secure boot       : 4
   User secure boot     : 0
   Mailbox authorization : 1
   DCI authorization    : 0
   OTP read             : 4
   Self test            : 4
   TRNG monitor         : 1
   PRS0                 : 1
   PRS1                 : 1
   PRS2                 : 2
   PRS3                 : 2
   PRS4                 : 4
   PRS5                 : 4
   PRS6                 : 7
   PRS7                 : 7
   Decouple BOD         : 4
   Temperature sensor   : 2
   Voltage glitch falling : 2
   Voltage glitch rising : 2
   Secure lock          : 4
   SE debug             : 0
   Digital glitch       : 2
   SE ICACHE            : 4
 + Reset period for the tamper filter counter: ~32 ms x 1024
 + Activation threshold for the tamper filter: 4
 + Digital glitch detector always on: Disabled
 + Tamper reset threshold: 5

 . Current tamper test is NORMAL.
 + Press SPACE to select NORMAL or TAMPER DISABLE, press ENTER to run.
```

Simplicity Commander

1. Run the security writekey command to provision the Public Command Key (e.g., command_pubkey.pem).

```
commander security writekey --command **command_pubkey.pem** --device EFR32MG21B010F1024 --serialno 440030580
```

```
Device has serial number 00000000000000014b457fffe0f77ce

================================================================================
Please look through any warnings before proceeding.
THIS IS A ONE-TIME command which permanently ties debug and tamper access to certificates signed by this key.
Type 'continue' and hit enter to proceed or Ctrl-C to abort:
================================================================================
continue
DONE
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

> **Note**: The Public Command Key cannot be changed once written.

2. Run the security readkey command to read the Public Command Key from the HSE OTP for verification with the key in step 1.

```
commander security readkey --command --device EFR32MG21B010F1024 --serialno 440030580
```

```
B1BC6F6FA56640ED522B2EE0F5B3CF7E5D48F60BE8148F0DC08440F0A4E1DCA4
7C04119ED6A1BE31B7707E5F9D001A659A051003E95E1B936F05C37EA793AD63
DONE
```

3. Run the security genconfig command to generate a default user_configuration.jsonfile for secure boot and tamper configuration.

```
commander security genconfig --nostore -o user_configuration.json --device EFR32MG21B010F1024 --serialno 440030580
```

```
Configuration file written to user_configuration.json
DONE
```

> **Note**: Simplicity Commander Version 1.14.6 or above is required to support tamper configuration for all HSE-SVH devices.

4. Use a text editor to modify the default tamper responses in `user_configuration.json` to the desired configuration as below.

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
{
   "OPN": "EFR32MG21B010F1024",
   "VERSION": "1.0.0",
   "mcu_flags": {
      "SECURE_BOOT_ANTI_ROLLBACK": false,
      "SECURE_BOOT_ENABLE": false,
      "SECURE_BOOT_PAGE_LOCK_FULL": false,
      "SECURE_BOOT_PAGE_LOCK_NARROW": false,
      "SECURE_BOOT_VERIFY_CERTIFICATE": false
   },
   "tamper_filter": {
      "FILTER_PERIOD": 10,
      "FILTER_THRESHOLD": 6,
      "RESET_THRESHOLD": 5
   },
   "tamper_flags": {
      "DGLITCH_ALWAYS_ON": false
   },
   "tamper_levels": {
      "DCI_AUTH": 0,
      "DECOUPLE_BOD": 4,
      "DGLITCH": 2,
      "FILTER_COUNTER": 1,
      "MAILBOX_AUTH": 1,
      "OTP_READ": 4,
      "PRS0": 1,
      "PRS1": 1,
      "PRS2": 2,
      "PRS3": 2,
      "PRS4": 4,
      "PRS5": 4,
      "PRS6": 7,
      "PRS7": 7,
      "SECURE_LOCK": 4,
      "SELF_TEST": 4,
      "SE_CODE_AUTH": 4,
      "SE_DEBUG": 0,
      "SE_HARDFAULT": 4,
      "SE_ICACHE": 4,
      "SE_RAM_CRC": 4,
      "SOFTWARE_ASSERTION": 4,
      "TEMP_SENSOR": 2,
      "TRNG_MONITOR": 1,
      "USER_CODE_AUTH": 0,
      "VGLITCH_FALLING": 2,
      "VGLITCH_RISING": 2,
      "WATCHDOG": 4
   }
}
```


> **Note**: This example does not enable the secure boot.

5. Run the `security writeconfig` command to program the secure boot and tamper configuration to the HSE OTP. This command can be executed once per device.

```
commander security writeconfig --configfile user_configuration.json --device EFR32MG21B010F1024 --serialno 440030580
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
================================================================================
THIS IS A ONE-TIME configuration: Please inspect file before confirming:
user_configuration.json
Type 'continue' and hit enter to proceed or Ctrl-C to abort:
================================================================================
continue
DONE
```

6. Run the `security readconfig` command to check the secure boot and tamper configuration of the device.

```
commander security readconfig --serialno 440030580
```

```
MCU Flags
Secure Boot                 : Disabled
Secure Boot Verify Certificate : Disabled
Secure Boot Anti Rollback      : Disabled
Secure Boot Page Lock Narrow   : Disabled
Secure Boot Page Lock Full     : Disabled

Tamper Levels
FILTER_COUNTER     : 1
WATCHDOG           : 4
SE_RAM_CRC         : 4
SE_HARDFAULT       : 4
SOFTWARE_ASSERTION : 4
SE_CODE_AUTH       : 4
USER_CODE_AUTH     : 0
MAILBOX_AUTH       : 1
DCI_AUTH           : 0
OTP_READ           : 4
SELF_TEST          : 4
TRNG_MONITOR       : 1
PRS0               : 1
PRS1               : 1
PRS2               : 2
PRS3               : 2
PRS4               : 4
PRS5               : 4
PRS6               : 7
PRS7               : 7
DECOUPLE_BOD       : 4
TEMP_SENSOR        : 2
VGLITCH_FALLING    : 2
VGLITCH_RISING     : 2
SECURE_LOCK        : 4
SE_DEBUG           : 0
DGLITCH            : 2
SE_ICACHE          : 4

Tamper Filter
Filter Period   : 10
Filter Threshold : 6
Reset Threshold  : 5

Tamper Flags
Digital Glitch Detector Always On: Disabled
DONE
```

Simplicity Studio

This example focuses on provisioning the Public Command Key and tamper configuration. It skips the procedures for provisioning of the Public Sign Key and Secure Boot Enabling.

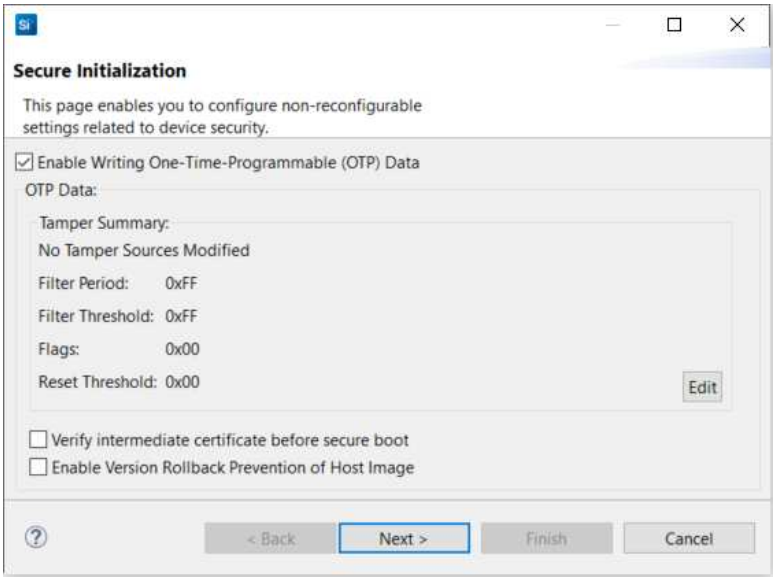
1. Run the `util keytotoken` command to convert the Public Command Key file (PEM format) into a text file (`command_pubkey.txt`).

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
commander util keytotoken command_pubkey.pem --outfile command_pubkey.txt
```

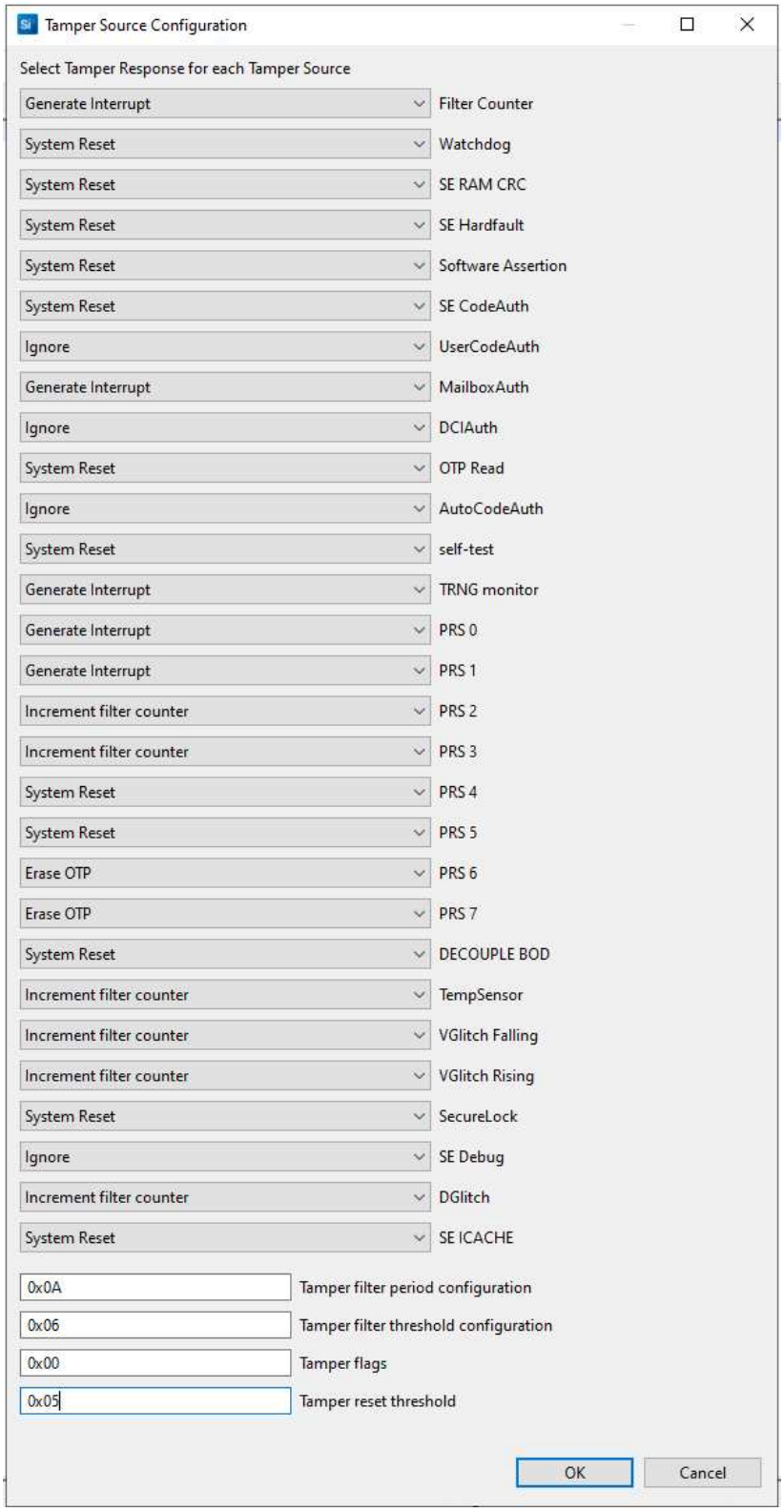
```
Writing EC tokens to command_pubkey.txt...
DONE
```

2. Open the **Security Settings** of the selected device as described in Using Simplicity Studio..
3. Click **[Start Provisioning Wizard...]** in the upper right corner to display the **Secure Initialization** dialog box.



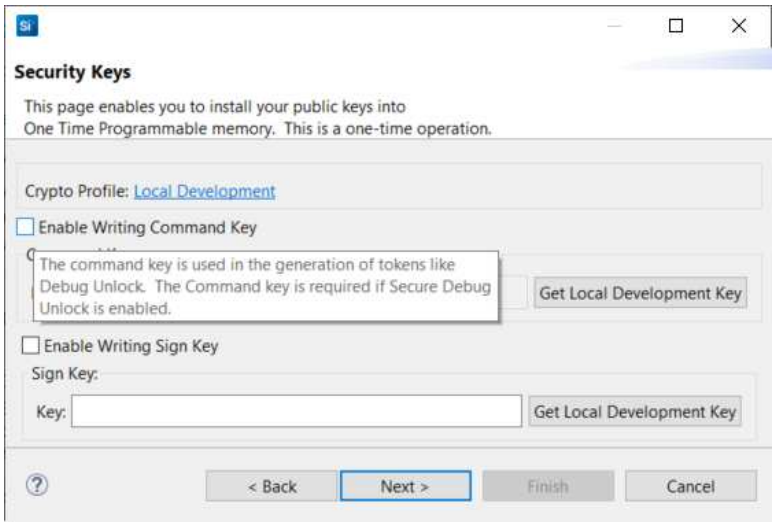
4. Click **[Edit]** to open the **Tamper Source Configuration** dialog box. Use the dropdown menus to modify the default tamper responses to the desired configuration. Click **[OK]** to exit.

Copyright © 2025 Silicon Laboratories. All rights reserved.



5. Click **[Next >]**. The **Security Keys** dialog box is displayed.

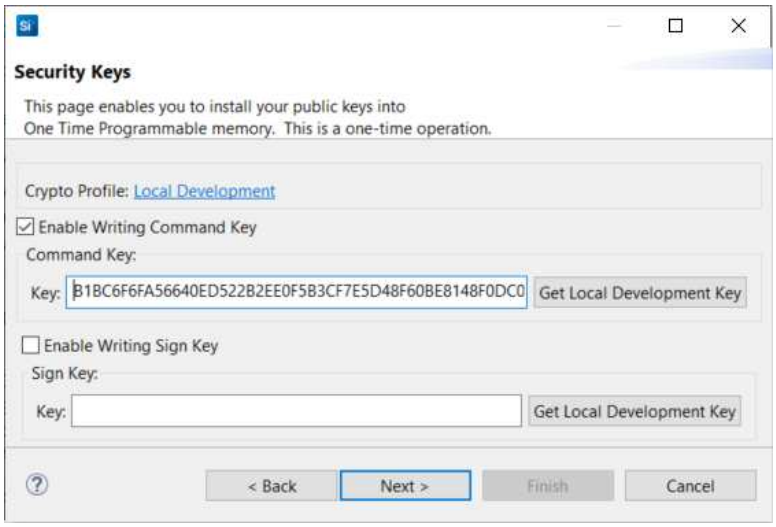
Copyright © 2025 Silicon Laboratories. All rights reserved.



6. Using a text editor, open the `command_pubkey.txt` file generated in step 1.

```
MFG_SIGNED_BOOTLOADER_KEY_X : B1BC6F6FA56640ED522B2EE0F5B3CF7E5D48F60BE8148F0DC08440F0A4E1DCA4
MFG_SIGNED_BOOTLOADER_KEY_Y : 7C04119ED6A1BE31B7707E5F9D001A659A051003E95E1B936F05C37EA793AD63
```

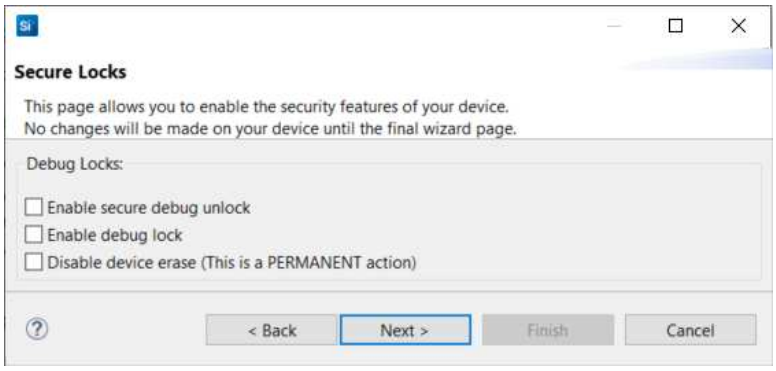
7. Check **Enable Writing Command Key**. Copy the Public Command Key (X-point `B1BC...` first, then Y-point `7C04...`) to the **Key:** box under **Command Key:**.



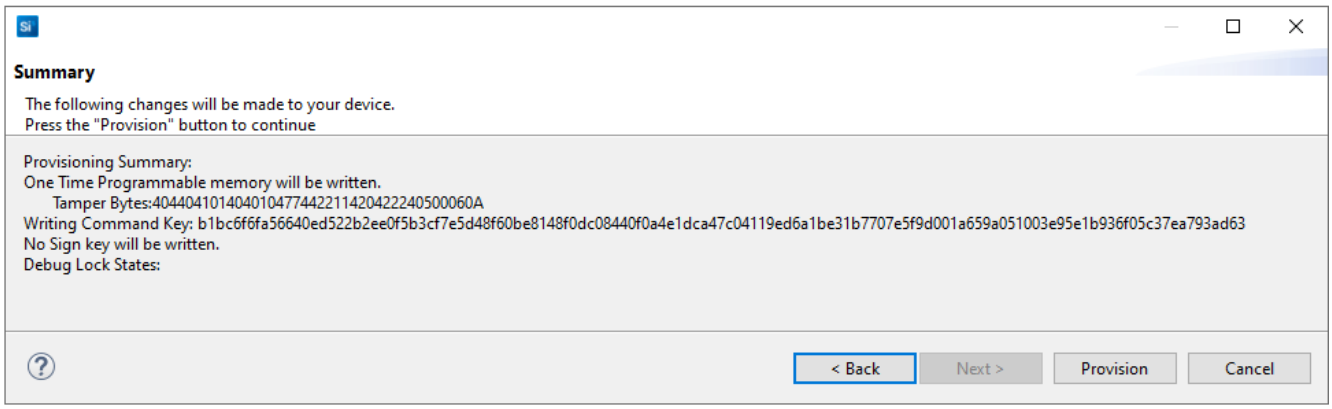
> **Note**: This example does not enable the secure boot (not checking **Enable Writing Sign Key** option).

8. Click **[Next >]**. The **Secure Locks** dialog box is displayed. The **Debug locks** are set by default. Uncheck **Enable secure debug unlock** and **Enable debug lock**.

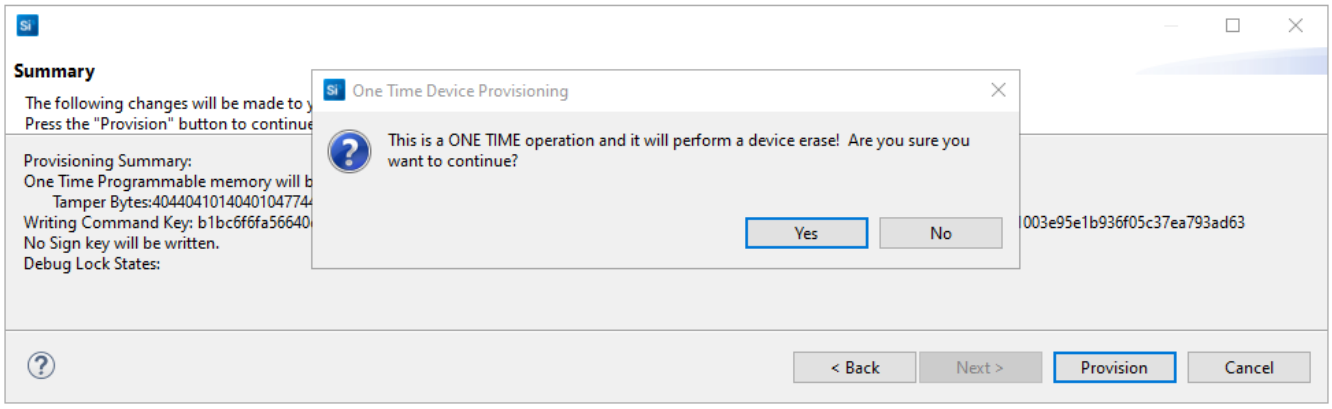
Copyright © 2025 Silicon Laboratories. All rights reserved.



9. Click **[Next >]** to display the **Summary** dialog box. Verify the tamper configuration and Public Command Key in the **Provisioning Summary** are correct.



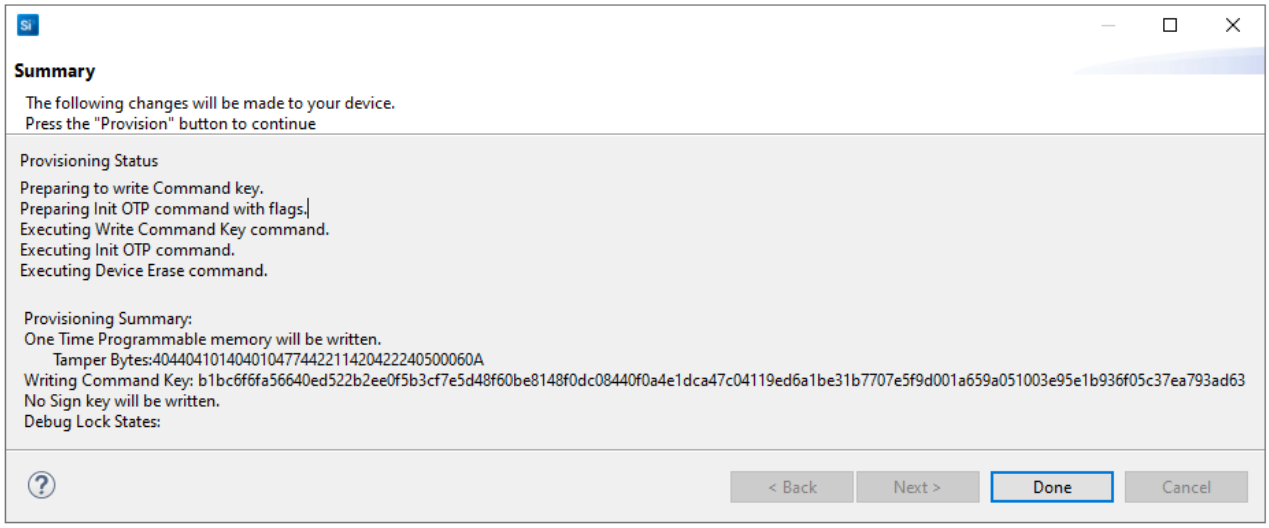
10. If the information displayed is correct, click **[Provision]**. Click **[Yes]** to confirm.



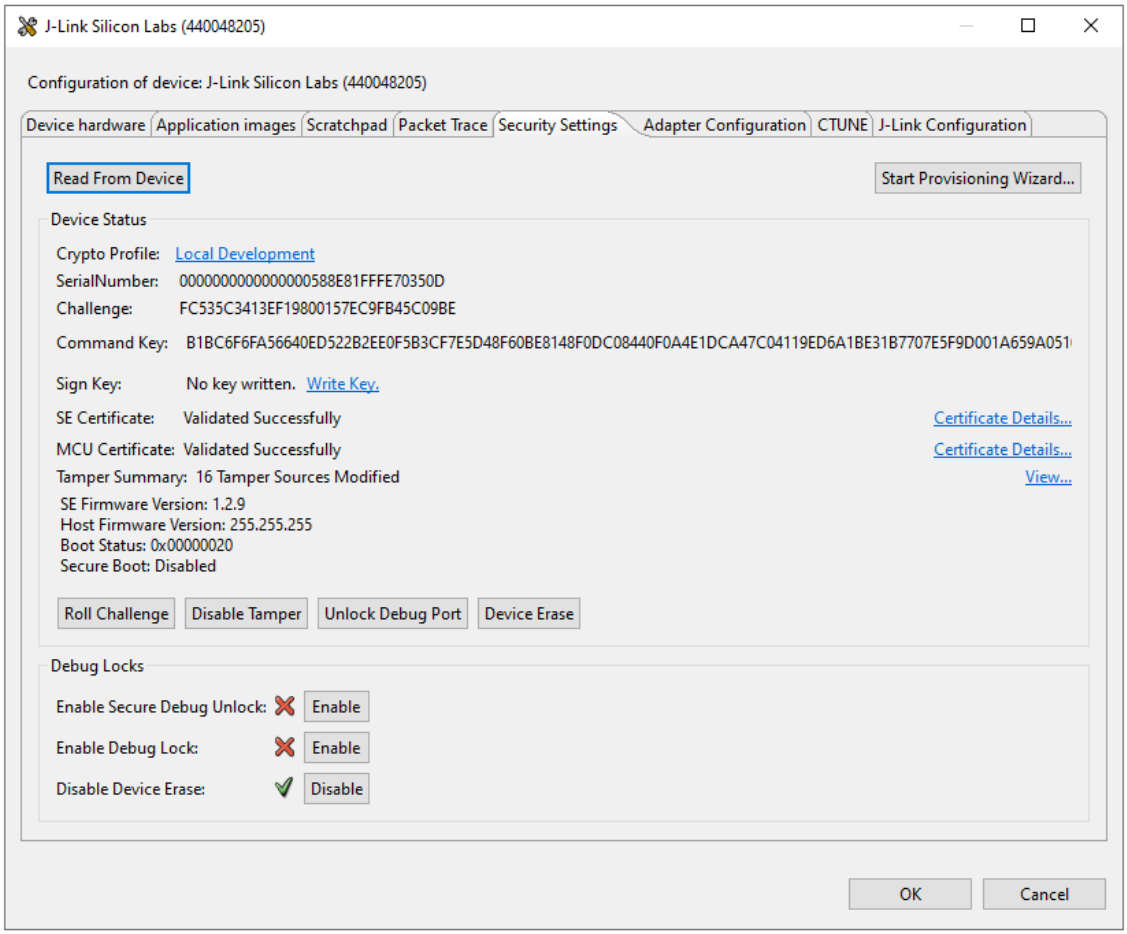
> **Note**: The Public Command Key and tamper configuration cannot be changed once written.

11. The **Provisioning Status** is displayed in the **Summary** dialog box.

Copyright © 2025 Silicon Laboratories. All rights reserved.



12. Click **[Done]** to exit the provisioning process. The device configuration is updated.



13. Click the **View...** link to check the tamper configuration or click **[OK]** to exit.

Tamper Disable and Roll Challenge

PRS Tamper Sources

Copyright © 2025 Silicon Laboratories. All rights reserved.

The tamper configuration in the SE Manager Tamper platform example is used to demonstrate the tamper disable on HSE-SVH devices. The following tables list the PRS tamper source usage on EFR32xG21B and other HSE-SVH devices on this example. The push buttons PB0 and PB1 are on the Wireless Starter Kit (WSTK) Mainboard.

Table: PRS Tamper Source Usage on EFR32xG21B Devices

Source (Bit)	Default Level (Response)	User Level (Response)	PRS Producer	Tamper Disable Mask (1)
PRS0 (16)	0 (Ignore)	1 (Interrupt)	Push button PB0	0
PRS1 (17)	0 (Ignore)	1 (Interrupt)	-	1
PRS2 (18)	0 (Ignore)	2 (Filter)	Push button PB0	0
PRS3 (19)	0 (Ignore)	2 (Filter)	-	1
PRS4 (20)	0 (Ignore)	4 (Reset)	Push button PB1	1
PRS5 (21)	0 (Ignore)	4 (Reset)	Software (2)	1
PRS6 (22)	0 (Ignore)	7 (Erase OTP)	-	1
PRS7 (23)	0 (Ignore)	7 (Erase OTP)	-	1

Notes:

1. The tamper disable mask is `0x00fa0000` to restore the tamper sources PRS1, PRS3, PRS4, PRS5, PRS6, and PRS7 to default response (Ignore).
2. The Software PRS triggers the tamper source PRS5 to reset the device if the filter counter reaches the trigger threshold (4) within the filter reset period (~32 ms x 1024).

Table: PRS Tamper Source Usage on Other HSE-SVH Devices

Source (Bit) (1)	Default Level (Response)	User Level (Response)	PRS Producer	Tamper Disable Mask (2)
PRS0 (25 or 26)	0 (Ignore)	1 (Interrupt)	-	1
PRS1 (26 or 27)	0 (Ignore)	1 (Interrupt)	Push button PB0	0
PRS2 (27 or 28)	0 (Ignore)	2 (Filter)	Push button PB0	0
PRS3 (28 or 29)	0 (Ignore)	2 (Filter)	-	1
PRS4 (29 or 30)	0 (Ignore)	4 (Reset)	Push button PB1	1
PRS5 (30 or 31)	0 (Ignore)	4 (Reset)	Software (3)	1
PRS6 (31 or -)	0 (Ignore)	7 (Erase OTP)	-	1

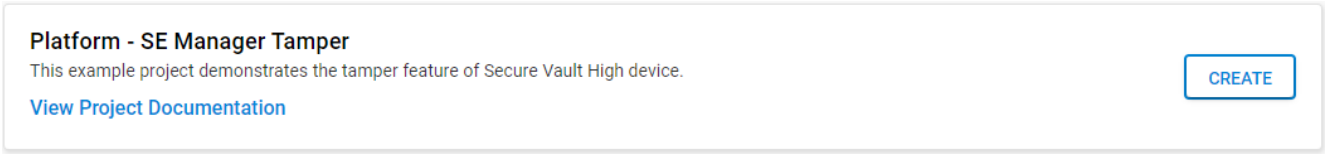
Notes:

1. The HSE-SVH devices with ETAMPDET peripheral only have PRS0 (bit 26) to PRS5 (bit 31).
2. The tamper disable mask depends on whether the HSE-SVH device has an ETAMPDET peripheral.
    1. Without ETAMPDET peripheral, the tamper disable mask is `0xf2000000` to restore the tamper sources PRS0, PRS3, PRS4, PRS5, and PRS6 to default response (Ignore).
    2. With ETAMPDET peripheral, the tamper disable mask is `0xe4000000` to restore the tamper sources PRS0, PRS3, PRS4, and PRS5 to default response (Ignore).
3. The Software PRS triggers the tamper source PRS5 to reset the device if the filter counter reaches the trigger threshold (4) within the filter reset period (~32 ms x 1024).

SE Manager - Tamper Platform Example

Click the `View Project Documentation` link to open the readme file for instructions on creating the project and running the example.

Copyright © 2025 Silicon Laboratories. All rights reserved.



Follow the procedures in SE Manager - Tamper Platform Example if the HSE OTP is uninitialized. The following sections describe an initialized device that runs in Normal and Tamper Disable modes.

Normal

1. Press `ENTER` to run the `NORMAL` tamper demo. Follow the instructions to go through the example.

```
. Current tamper test is NORMAL.
+ Press SPACE to select NORMAL or TAMPER DISABLE, press ENTER to run.

. Normal tamper test instructions:
+ Press PB0 to increase filter counter and tamper status is displayed.
+ PRS will issue a tamper reset if filter counter reaches 4 within ~32 ms x 1024.
+ Press PB1 to issue a tamper reset.
+ Device will enter diagnostic mode if tamper reset reaches 5.
```

2. Press PB0 to trigger PRS0 (Interrupt) and PRS2 (Filter) to issue an interrupt. The active tamper sources (`0x00050000`) of the EFR32xG21B device are PRS0 (`bit 16`) and PRS2 (`bit 18`).

```
. Get tamper status... SL_STATUS_OK (cycles: 11937 time: 314 us)
+ Recorded tamper status (MSB..LSB): 00050001
+ Currently active tamper sources (MSB..LSB): 00050000
```

3. Press PB0 (Filter on PRS2) 4 times within ~32 ms x 1024 to trigger an interrupt when reaching the filer counter threshold. The program will use software PRS to issue a tamper reset through the PRS5 tamper source. The active tamper sources (`0x00050002`) of the EFR32xG21B device are Filter (`bit 2`), PRS0 (`bit 16`), and PRS2 (`bit 18`).

```
. Get tamper status... SL_STATUS_OK (cycles: 11725 time: 308 us)
+ Recorded tamper status (MSB..LSB): 00050002
+ Currently active tamper sources (MSB..LSB): 00050002
+ Tamper filter threshold is reached, issue a reset through PRS
```

4. Press PB1 to trigger PRS4 (Reset) to issue a tamper reset.
5. After a tamper reset, the `SETAMPER` (bit 13) in `EMU->RSTCAUSE` register is set. Note that bit 1 indicates a pin reset and will also be set.

```
. Read EMU RSTCAUSE register... SL_STATUS_OK (cycles: 4071 time: 107 us)
+ The EMU RSTCAUSE register (MSB..LSB): 00002002
+ The tamper reset is observed
```

6. After five consecutive tamper resets (reset threshold in this example), the device will enter diagnostic mode until a power-on or pin reset.

Tamper Disable

This example uses the tamper disable mask (`0x00fa0000`) to restore the tamper sources PRS1, PRS3, PRS4, PRS5, PRS6, and PRS7 of EFR32xG21B device to default response (Ignore).

1. Press `SPACE` to select `TAMPER DISABLE` , press `ENTER` to run.

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
. Current tamper test is NORMAL.
+ Press SPACE to select NORMAL or TAMPER DISABLE, press ENTER to run.
+ Current tamper test is TAMPER DISABLE.
```

2. This example will prompt to program the default Public Command Key in flash to the HSE OTP if this key does not exist. Press `ENTER` two times to confirm and `ENTER` again to restore the default tamper level. Follow the instructions shown in step 3 to go through the example (steps 4 to 6).

```
. Verify the device public command key in SE OTP.
+ Exporting a public command key from a hard-coded private command key... SL_STATUS_OK (cycles: 210999 time: 5552 us)
+ Reading the public command key from SE OTP... SL_STATUS_NOT_INITIALIZED (cycles: 7763 time: 204 us)
+ Press ENTER to program public command key in SE OTP or press SPACE to abort.
+ Warning: The public command key in SE OTP cannot be changed once written!
+ Press ENTER to confirm or press SPACE to skip if you are not sure.
+ Programming a public command key to SE OTP... SL_STATUS_OK (cycles: 79656 time: 2096 us)
+ Press ENTER to disable tamper signals or press SPACE to exit.
```

3. Press `ENTER` to restore the default tamper level if the default Public Command Key in flash matches with the key in the HSE OTP. Follow the instructions to go through the example (steps 4 to 6).

```
. Verify the device public command key in SE OTP.
+ Exporting a public command key from a hard-coded private command key... SL_STATUS_OK (cycles: 200804 time: 5284 us)
+ Reading the public command key from SE OTP... SL_STATUS_OK (cycles: 7134 time: 187 us)
+ Comparing exported public command key with SE OTP public command key... OK
+ Press ENTER to disable tamper signals or press SPACE to exit.

. Start the tamper disable processes.
+ Creating a private certificate key in a buffer... SL_STATUS_OK (cycles: 214059 time: 5633 us)
+ Exporting a public certificate key from a private certificate key... SL_STATUS_OK (cycles: 206545 time: 5435 us)
+ Read the serial number of the SE and save it to access certificate... SL_STATUS_OK (cycles: 7930 time: 208 us)
+ Signing the access certificate with private command key... SL_STATUS_OK (cycles: 222650 time: 5859 us)
+ Request challenge from the SE and save it to challenge response... SL_STATUS_OK (cycles: 4208 time: 110 us)
+ Signing the challenge response with private certificate key... SL_STATUS_OK (cycles: 223559 time: 5883 us)
+ Creating a tamper disable token to disable tamper signals... SL_STATUS_OK (cycles: 946431 time: 24906 us)
+ Success to disable the tamper signals!

. Tamper disable test instructions:
+ Press PB0 to increase filter counter and tamper status is displayed.
+ PRS will NOT issue a tamper reset even filter counter reaches 4 within ~32 ms x 1024.
+ Press PB1 will NOT issue a tamper reset.
+ Issue a power-on or pin reset to re-enable the tamper signals.
+ Press ENTER to roll the challenge to invalidate the current tamper disable token or press SPACE to exit.
```

4. Press PB0 to verify tamper sources PRS0 (Interrupt) and PRS2 (Filter) of EFR32xG21B device can still issue an interrupt.

```
. Get tamper status... SL_STATUS_OK (cycles: 11259 time: 296 us)
+ Recorded tamper status (MSB..LSB): 00050001
+ Currently active tamper sources (MSB..LSB): 00050000
```

5. The PRS5 tamper source (configured as Reset) was restored to the default (Ignore), so it cannot issue a tamper reset even if users press PB0 (Filter on PRS2) 4 times within ~32 ms x 1024.
6. The PRS4 tamper source (configured as Reset) was restored to the default (Ignore), so it cannot issue a tamper reset even if users press PB1.
7. Issue a power-on or pin reset to exit the tamper disable state or press `ENTER` to roll the challenge.

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
. Check and roll the challenge.
+ Request current challenge from the SE... SL_STATUS_OK (cycles: 0 time: 0 us)
+ The current challenge (16 bytes):
  AA C1 79 FC FC C5 78 8E A0 3F 91 AB 5D A9 C5 04
+ Rolling the challenge... SL_STATUS_OK (cycles: 0 time: 0 us)
+ Request rolled challenge from the SE... SL_STATUS_OK (cycles: 0 time: 0 us)
+ The rolled challenge (16 bytes):
  0F 63 9C 44 46 E4 7C B2 C9 CA 66 13 34 34 92 8E
+ Issue a power-on or pin reset to activate the rolled challenge.

. SE manager deinitialization... SL_STATUS_OK (cycles: 0 time: 0 us)
```

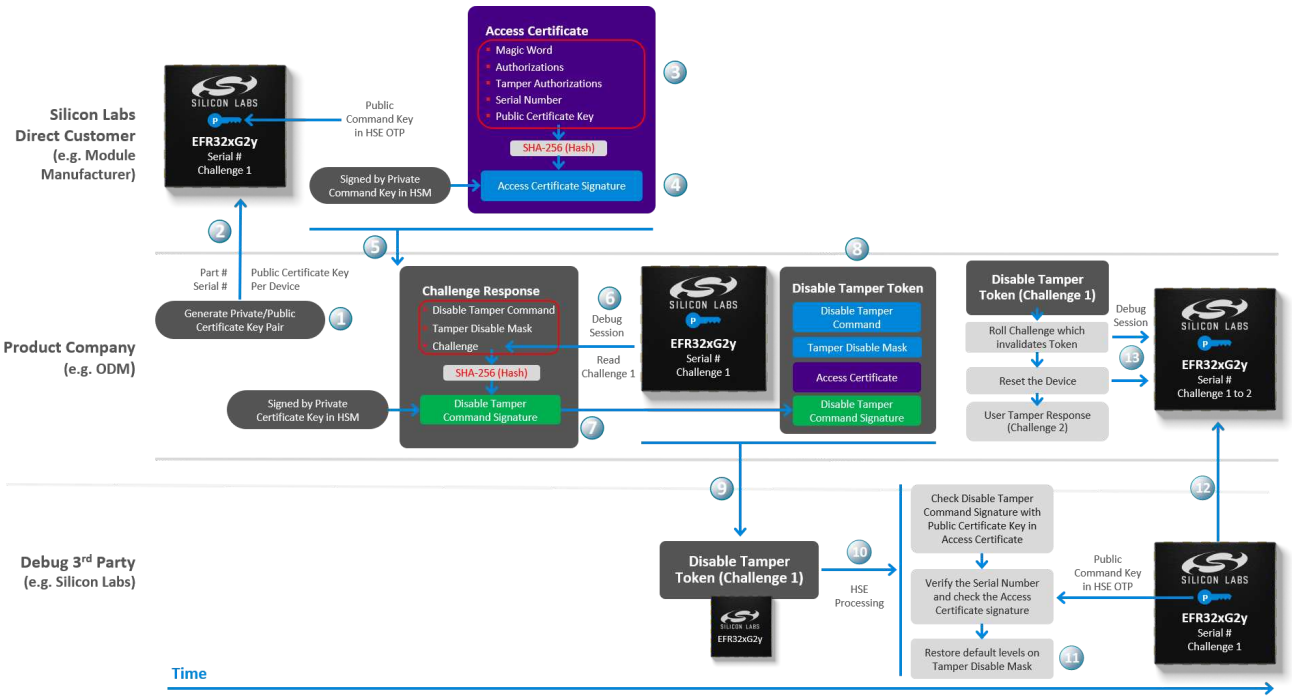
Simplicity Commander

The tamper disable was designed with three organizations in mind:

1. The Direct Customer to whom Silicon Labs sells the chip. This chip has the Public Command Key installed in the SE OTP.
2. The Product Company is a customer of the Direct Customer. This is applicable if the Direct Customer is creating a white-labeled product for another company or a sub-component that goes into another company's product.
3. The Debug 3^{rd} Party could be anyone, internal or external, that the Product Company decides is qualified to debug the device.

Because the Public Command Key is installed into the SE OTP of a large number of devices and cannot be changed, the corresponding Private Command Key must be guarded by a very stringent process. If this Private Command Key is ever leaked, all the devices programmed with the corresponding Public Command Key will be compromised.

A tamper disable use case is described in the following figure, and the signing process is performed by a Hardware Security Module (HSM).



The tamper disable flow moving across the time axis from left to right is explained below:

Copyright © 2025 Silicon Laboratories. All rights reserved.

1. The Product Company creates a Private/Public Certificate Key pair for each device. Because the key pair is assigned only to a single device, the company may not need to protect the Private Certificate Key as securely as the Private Command Key by the Direct Customer.
   In this example, the Private/Public Certificate Key pair (`cert_key.pemand` and `cert_pubkey.pem`) is generated by running the `util genkey` command.

```
commander util genkey --type ecc-p256 --privkey cert_key.pem --pubkey cert_pubkey.pem
```

```
Generating ECC P256 key pair...
Writing private key file in PEM format to cert_key.pem
Writing public key file in PEM format to cert_pubkey.pem
DONE
```

2. The Public Certificate Key (`cert_pubkey.pem`) for each device is passed to the Silicon Labs Direct Customer. The part number and serial number are also required if Direct Customer cannot access the device.
   Run the `security status` command to get the device serial number. The `--serialno` option is for the J-Link serial number of the WSTK.

```
commander security status --device EFR32MG21B010F1024 --serialno 440030580
```

```
SE Firmware version   : 1.2.9
Serial number         : 00000000000000014b457fffe0f77ce
Debug lock            : Disabled
Device erase          : Enabled
Secure debug unlock   : Disabled
Tamper status         : OK
Secure boot           : Disabled
Boot status           : 0x20 - OK
Command key installed : True
Sign key installed    : False
DONE
```

3. The Direct Customer then places that Public Certificate Key in the access certificate. The access certificate is per device because it contains the unique device serial number. This certificate is generated once upon creation of the device, and thereafter, is generally only modified when the Private/Public Certificate Key pair is changed by the Product Company.
   The following two steps are **OPTIONAL** for customization of Authorizations and Tamper Authorizations.
   a. (**Optional**) Run the `security genauth` command to generate the default certificate authorization file (`certificate_authorization.json`).

```
commander security genauth -o certificate_authorizations.json --nostore --serialno 440030580
```

```
DONE
```

   b. (**Optional**) Use a text editor to modify the default Authorizations and Tamper Authorizations in the `json` file.
   Run the `security gencert` command with the following parameters from the Product Company to generate an unsigned access certificate (`access_certificate.extsign`) in Security Store:
   - Device part number
   - Device serial number
   - Public Certificate Key

```
commander security gencert --device EFR32MG21B010F1024 --deviceserialno 00000000000000014b457fffe0f77ce
--cert-pubkey cert_pubkey.pem --extsign
```

```
Authorization file written to Security Store:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000014b457fffe0f77ce/certificate_authoriz

Cert key written to Security Store:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000014b457fffe0f77ce/cert_pubkey.pem

Created an unsigned certificate in Security Store:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000014b457fffe0f77ce/access_certificate.
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Notes:

- The `--extsign` option to create an unsigned access certificate is only available in Simplicity Commander Version 1.11.2 or above.
- The unsigned access certificate is generated with the default certificate authorization file (`certificate_authorization.json`) which uses `0x0000003e` for Authorizations and `0xffffffb6` (HSE-SVH device) for Tamper Authorizations (Table Elements of the Access Certificate on page).
- **(Optional)** Use `--authorization` option if the customized `json` file generated in the above optional steps (a) and (b) is used.

```
commander security gencert --device EFR32MG21B010F1024 --authorization certificate_authorizations.json
--deviceserialno 00000000000000014b457fffe0f77ce --cert-pubkey cert_pubkey.pem --extsign
```

4. The signing of the access certificate can be done by passing an unsigned access certificate to a Hardware Security Module (HSM) containing the Private Command Key.
   In this example, the OpenSSL is used to sign the access certificate (`access_certificate.extsign`) in Security Store with the Private Command Key (`command_key.pem`). The access certificate signature is in the `cert_signature.bin` file.

```
openssl dgst -sha256 -binary -sign command_key.pem -out cert_signature.binaccess_certificate.extsign
```

   Run the `util signcert` command with the following parameters to verify the signature and generate the signed access certificate (`access_certificate.bin`):
   - Unsigned access certificate
   - Access certificate signature
   - Public Command Key

```
commander util signcert access_certificate.extsign --cert-type access --signature cert_signature.bin
--verify command_pubkey.pem --outfile access_certificate.bin
```

```
R = 76CDC5BA18E5248FDA5418002F250F149B449829A005D6F0726268016CC53ED4
S = E4B8ABA2CF742B0E6CC5BA2C1023D76BEEF3C4A11DA97CC4D23459F32237A206
Successfully verified signature
Successfully signed certificate
DONE
```

   Notes:
   - Put the required files in the same folder to run the command.
   - The `util signcert` command for access certificate is only available in Simplicity Commander Version 1.11.2 or above.
   - The access certificate signature can be in a Raw or Distinguished Encoding Rules (DER) format.
5. The access certificate is passed to the Product Company. The purpose of the access certificate is to grant overall debug access capabilities to the Product Company and authorize them to allow third parties to debug the device. The Product Company can now use the access certificate to generate the Disable Tamper Token. The same access certificate can be used to generate as many Disable Tamper Tokens as necessary without having to ever go back to the Direct Customer.
6. To create the Disable Tamper Token, a debug session must be started with the device and the challenge value (which is a random number `Challenge 1` in this example) should be read out to generate the challenge response.
   Run the `security gencommandcommand` to generate the challenge response without disable tamper command signature and store it in a file (`command_unsign.bin`).

```
commander security gencommand --action disable-tamper --disable-param 0x00fa0000 -o command_unsign.bin
--nostore --device EFR32MG21B010F1024 --serialno 440030580
```

```
Unsigned command file written to:
command_unsign.bin
DONE
```

   - The tamper disable mask (`0x00fa0000`) is based on the Tamper platform example on EFR32xG21B devices (Table PRS Tamper Source Usage on EFR32xG21B Devices).
   - If the `--disable-param` option is not provided, it will restore all tamper sources (`0xffffffb6`) by default.
7. The challenge response is then cryptographically hashed (SHA-256) to create a digest. The digest is then signed by the Private Certificate Key to generate the disable tamper command signature.
   The signing of the challenge response can be done by passing an unsigned challenge response to a Hardware Security Module (HSM) containing the Private Certificate Key.

Copyright © 2025 Silicon Laboratories. All rights reserved.

In this example, the OpenSSL is used to sign the challenge response (`command_unsign.bin`) with the Private Certificate Key (`cert_key.pem`). The disable tamper command signature is in the `command_signature.bin` file.

```
openssl dgst -sha256 -binary -sign cert_key.pem -out command_signature.bincommand_unsign.bin
```

8. Run the `security disabletamper` command with the access certificate (`access_certificate.bin`) from Direct Customer and disable tamper command signature (`command_signature.bin`) in step 7 to generate the Disable Tamper Token.

```
commander security disabletamper --disable-param  0x00fa0000 --cert access_certificate.bin
--command-signature command_signature.bin EFR32MG21B010F1024 --serialno 440030580
```

```
Certificate written to Security Store:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000014b457fffe0f77ce/access_certificate.

R = A70834D97640A92510D151765F0EED6C6A05CB8BE81E06E905C230ED24E71659
S = 9B69C113C2B7DEE60BF0BC7D72719F7F9465840D68EADBBB4F9BCE7A1267B936
Command signature is valid
Tamper successfully disabled.
Command disable tamper payload was stored in Security Store
DONE
```

Notes:

- Put the required files in the same folder to run the command.
- The disable tamper command signature can be in a Raw or Distinguished Encoding Rules (DER) format.
- Simplicity Commander Version 1.11.2 or above is required to support signature in DER format.

9. **(Alternative)** Key protection is not required if the Private Certificate Key is ephemeral. Steps 6 to 8 can be implemented by running the `security disabletamper` command with the access certificate (`access_certificate.bin`) from the Direct Customer and Private Certificate Key (`cert_key.pem`) to generate the Disable Tamper Token.

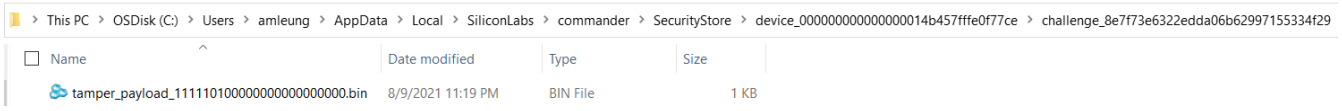
```
commander security disabletamper --disable-param 0x00fa0000 --cert access_certificate.bin --cert-privkey cert_key.pem --device
EFR32MG21B010F1024 --serialno 440030580
```

```
Certificate written to Security Store:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000014b457fffe0f77ce/access_certificate.

Cert key written to Security Store:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000014b457fffe0f77ce/cert_pubkey.pem

Created unsigned disable tamper command
Signed disable tamper command using
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000014b457fffe0f77ce/cert_key.pem
Tamper successfully disabled.
Command disable tamper payload was stored in Security Store
DONE
```

10. The Disable Tamper Token (aka `Command disable tamper payload`) file (`tamper_payload_111110100000000000000000.bin` , where `111110100000000000000000` is `0x00fa0000` for tamper disable mask) is stored in the Security Store. The location in Windows is `C:\Users\<PC user name>\AppData\Local\SiliconLabs\commander\SecurityStore\device_<Serial number>\challenge_<Challenge value>` .



Users can also use the `security getpath` command to get the path of the Security Store or a specified device.

```
commander security getpath
```

```
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore
DONE
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_000000000000000588e81fffe70350d
DONE
```

11. The Disable Tamper Token and the device are now delivered to the Debug 3^{rd} Party.
    Run the `security gencommand` command to create the Security Store to place the Disable Tamper Token file.

```
commander security gencommand --action disable-tamper --disable-param 0x00fa0000 --device EFR32MG21B010F1024 --serialno 440030580
```

```
Unsigned command file written to Security Store:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000014b457fffe0f77ce/challenge_8e7f73e

DONE
```

    Copy the Disable Tamper Token file (`tamper_payload_11111010000000000000000.bin`) from Product Company to the Windows Security Store `challenge_<Challenge value>` folder located in `C:\Users\<PC username>\AppData\Local\SiliconLabs\commander\SecurityStore\device_<Serial number>\challenge_<Challenge value>` .
12. The device compares the Disable Tamper Token contents with its internal serial number, challenge value, and Public Command Key to determine the token's authenticity. If authentic, it will execute the disable tamper command to restore the default levels on the tamper disable mask (`0xfa000000`) ; otherwise, it will ignore the command.
    Run the `security disabletamper` command to disable the tamper.

```
commander security disabletamper --disable-param 0x00fa0000 --device EFR32MG21B010F1024 --serialno 440030580
```

```
Disabling tamper with tamper payload:
C:/Users/<username>/AppData/Local/SiliconLabs/commander/SecurityStore/device_00000000000000014b457fffe0f77ce/challenge_8e7f73e
 successfully disabled.
DONE
```

> **Note**: Users can verify the Disable Tamper Token by following steps 4 to 6 in Tamper Disable if the EFR32xG21B device is running in the Normal mode of the SE Manager Tamper platform example.

13. The Debug 3^{rd} Party can now use this same Disable Tamper Token to disable the tamper (step 12), over and over again after each power-on or pin reset, until they have finished debugging the device.
14. Once the Debug 3^{rd} Party has finished debugging, they will send the device back to the Product Company.
15. Once the Product Company receives the device, they will immediately start a debug session to roll the challenge (from `Challenge 1` to `Challenge 2` in this example). Rolling the challenge will effectively invalidate any Disable Tamper Token that has been previously given to any third party.
    Run the `security rollchallenge` command and reset the device to invalidate the current Disable Tamper Token. The challenge cannot be rolled before it has been used at least once — that is, by running the `security disabletamper` or `security unlock` command.

```
commander security rollchallenge --device EFR32MG21B010F1024 --serialno 440030580
```

```
Challenge was rolled successfully.
DONE
```

    The unlock token is invalidated after rolling the challenge because any previously issued Disable Tamper Token now contains a different challenge value (`Challenge 1`) than the challenge value currently in the device (`Challenge 2`).
    The validation process of any previously issued Disable Tamper Token will always fail until a new Disable Tamper Token is issued with a current matching challenge value (`Challenge 2`).

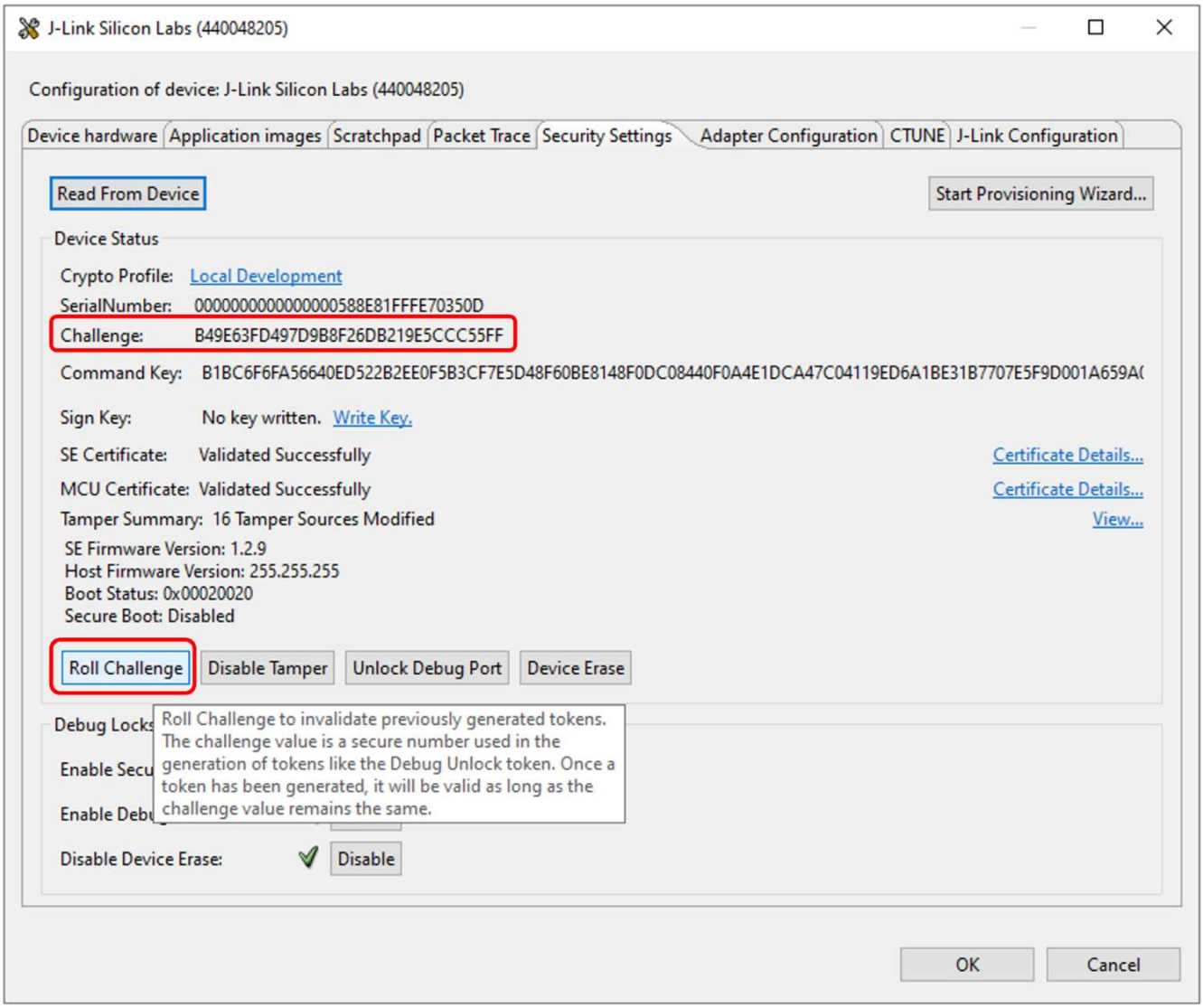
Copyright © 2025 Silicon Laboratories. All rights reserved.

> **Note**: Direct Customer can directly use the Private Command Key on the connected chip to generate the Disable Tamper Token in Security Store. But it has a high risk (cannot use HSM) to leak the Private Command Key to a 3^{rd} party when using this approach.

```
commander security disabletamper --disable-param 0x00fa0000 --command-key command_key.pem
--device EFR32MG21B010F1024 --serialno 440030580
```

Simplicity Studio

1. Open **Security Settings** of the selected device as described in Using Simplicity Studio.
2. Click **[Roll Challenge]** to generate a new challenge value to invalidate the Disable Tamper Token for tamper disable. Click **[OK]** to exit.



Copyright © 2025 Silicon Laboratories. All rights reserved.

Authenticating Silicon Labs Devices using Device Certificates

Authenticating Silicon Labs Devices Using Device Certificates

Note: This section replaces *AN1268: Authenticating Silicon Labs Devices Using Device Certificates*. Further updates to this application note will be provided here.

This application note describes how to authenticate a device as a genuine Silicon Labs product at any time during its life. The digital certificates for secure identity are stored in the device and the Silicon Labs Server. This secure identity feature is only available on Secure Vault High devices.

Key Points

- Secure identity on Secure Vault High devices
- Device certificate options
- Entity Attestation Token (EAT)
- Remote authentication process
- Examples for certificate chain verification and remote authentication

Copyright © 2025 Silicon Laboratories. All rights reserved.

Series 2 Device Security Features

Series 2 Device Security Features

Protecting IoT devices against security threats is central to a quality product. Silicon Labs offers several security options to help developers build secure devices, secure application software, and secure paths of communication to manage those devices. Silicon Labs' security offerings were significantly enhanced by the introduction of the Series 2 products that included a Secure Engine. The Secure Engine is a tamper-resistant component used to securely store sensitive data and keys and to execute cryptographic functions and secure services.

On Series 1 devices, the security features are implemented by the TRNG (if available) and CRYPTO peripherals.

On Series 2 devices, the security features are implemented by the Secure Engine and CRYPTOACC (if available). The Secure Engine may be hardware-based, or virtual (software-based). Throughout this document, the following abbreviations are used:

- HSE - Hardware Secure Engine
- VSE - Virtual Secure Engine
- SE - Secure Engine (either HSE or VSE)

Additional security features are provided by Secure Vault. Three levels of Secure Vault feature support are available, depending on the part and SE implementation, as reflected in the following table:

Level (1)	SE Support	Part (2)
Secure Vault High (SVH)	HSE only (HSE-SVH)	EFR32xG2yB (3)
Secure Vault Mid (SVM)	HSE (HSE-SVM)	EFR32xG2yA (3)
"	VSE (VSE-SVM)	EFR32xG2y, EFM32PG2y (4)
Secure Vault Base (SVB)	N/A	MCU Series 1 and Wireless SoC Series 1

Notes:

1. The features of different Secure Vault levels can be found in https://www.silabs.com/security.
2. The x is a letter (B, F, M, or Z).
3. At the time of this writing, the y is a digit (1, 3, or 4).
4. At the time of this writing, the y is a digit (2).

Secure Vault Mid consists of two core security functions:

- **Secure Boot**: Process where the initial boot phase is executed from an immutable memory (such as ROM) and where code is authenticated before being authorized for execution.
- **Secure Debug access control**: The ability to lock access to the debug ports for operational security, and to securely unlock them when access is required by an authorized entity.

Secure Vault High offers additional security options:

- **Secure Key Storage**: Protects cryptographic keys by "wrapping" or encrypting the keys using a root key known only to the HSE-SVH.
- **Anti-Tamper protection**: A configurable module to protect the device against tamper attacks.
- **Device authentication**: Functionality that uses a secure device identity certificate along with digital signatures to verify the source or target of device communications.

A Secure Engine Manager and other tools allow users to configure and control their devices both in-house during testing and manufacturing, and after the device is in the field.

User Assistance

Copyright © 2025 Silicon Laboratories. All rights reserved.

In support of these products Silicon Labs offers whitepapers, webinars, and documentation. The following table summarizes the key security documents:

Document	Summary	Applicability
Series 2 Secure Debug	How to lock and unlock Series 2 debug access, including background information about the SE	Secure Vault Mid and High
Series 2 Secure Boot with RTSL	Describes the secure boot process on Series 2 devices using SE	Secure Vault Mid and High
Anti-Tamper Protection Configuration and Use	How to program, provision, and configure the anti-tamper module	Secure Vault High
Authenticating Silicon Labs Devices using Device Certificates (this document)	How to authenticate a device using secure device certificates and signatures, at any time during the life of the product	Secure Vault High
Secure Key Storage	How to securely 'wrap' keys so they can be stored in non-volatile storage.	Secure Vault High
AN1222: Production Programming of Series 2 Devices	How to program, provision, and configure security information using SE during device production	Secure Vault Mid and High

Key Reference

Public/Private keypairs along with other keys are used throughout Silicon Labs security implementations. Because terminology can sometimes be confusing, the following table lists the key names, their applicability, and the documentation where they are used.

Key Name	Customer Programmed	Purpose
Public Sign key (Sign Key Public)	Yes	Secure Boot binary authentication and/or OTA upgrade payload authentication
Public Command key (Command Key Public)	Yes	Secure Debug Unlock or Disable Tamper command authentication
OTA Decryption key (GBL Decryption key) aka AES-128 Key	Yes	Decrypting GBL payloads used for firmware upgrades
Attestation key aka Private Device Key	No	Device authentication for secure identity

Device Compatibility

This application note applies to Series 2 HSE-SVH device families. Refer to IoT Endpoint Security Fundamentals for details on supporting devices.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Introduction

Introduction

One of the biggest challenges for connected devices is post-deployment authentication. Silicon Labs' factory trust provisioning service with optional secure programming provides a secure device identity certificate, analogous to a birth certificate, for each individual silicon die during integrated circuit (IC) manufacturing. This enables post-deployment security, authenticity, and attestation-based health checks. The device certificate guarantees the authenticity of the device for its lifetime. When the certificate is checked, a digital signature confirms that the certificate received has not been tampered with.

Certificates can now be used to authenticate Internet of Things (IoT) devices as well as Internet servers, now that Silicon Labs' HSE-SVH devices have both cryptographic acceleration in hardware and tamper-resistant storage to handle digital certificate operations.

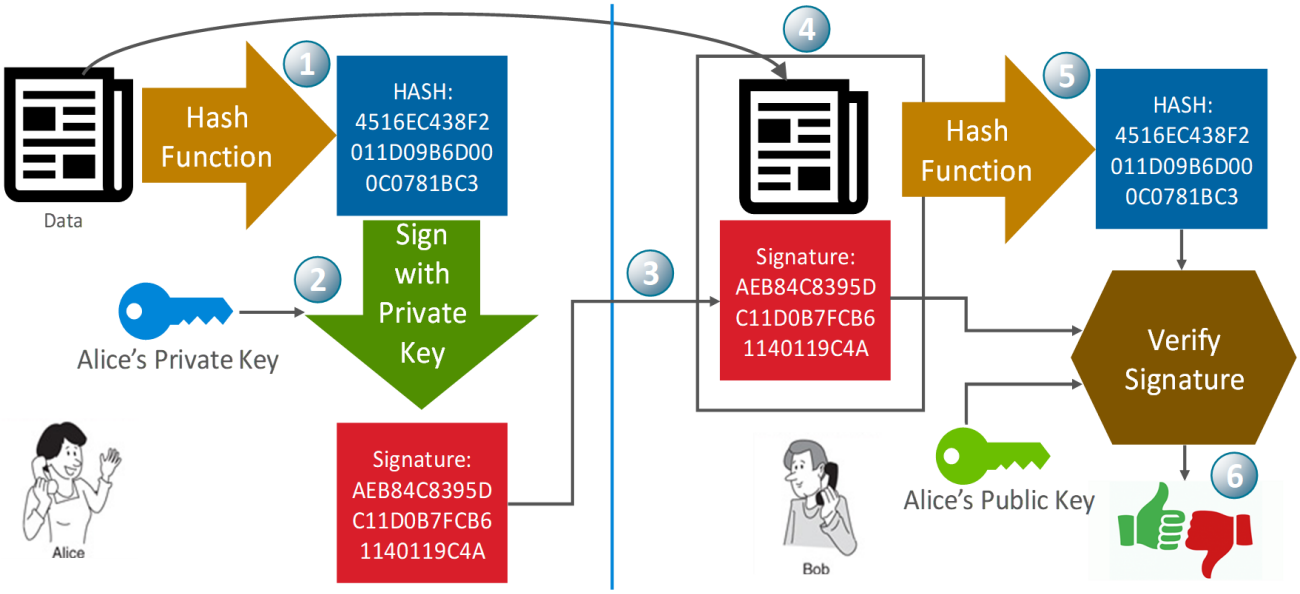
The digital signature and certificates are major cryptographic tools to verify the device is authentic. These tools are described in the following sections.

Digital Signature

The digital signature is used to protect integrity and authenticity of an electronic message.

Digital Signature Example

Alice wants to give data to Bob, and Bob wants to make sure that the data came from Alice and has not been tampered with. Alice has a private/public key pair, and has previously given Bob her public key.



1. Alice generates the hash (for example SHA256) of the data.
2. Alice's private key is used to sign the hash to create a signature. The hash is signed instead of the data itself because the signing operation is slow. Therefore it is more efficient to sign the hash instead of the arbitrarily large data.
3. The signature is attached to the end of the data.
4. The data and signature are given to Bob.

Copyright © 2025 Silicon Laboratories. All rights reserved.

5. Bob independently generates the hash of the data.
6. The signature is verified with the hash and Alice's public key, which results in a true or false outcome indicating if the data is valid.

> **Note**: This scheme requires distribution of Alice's public key.

Digital Certificates and Chain of Trust

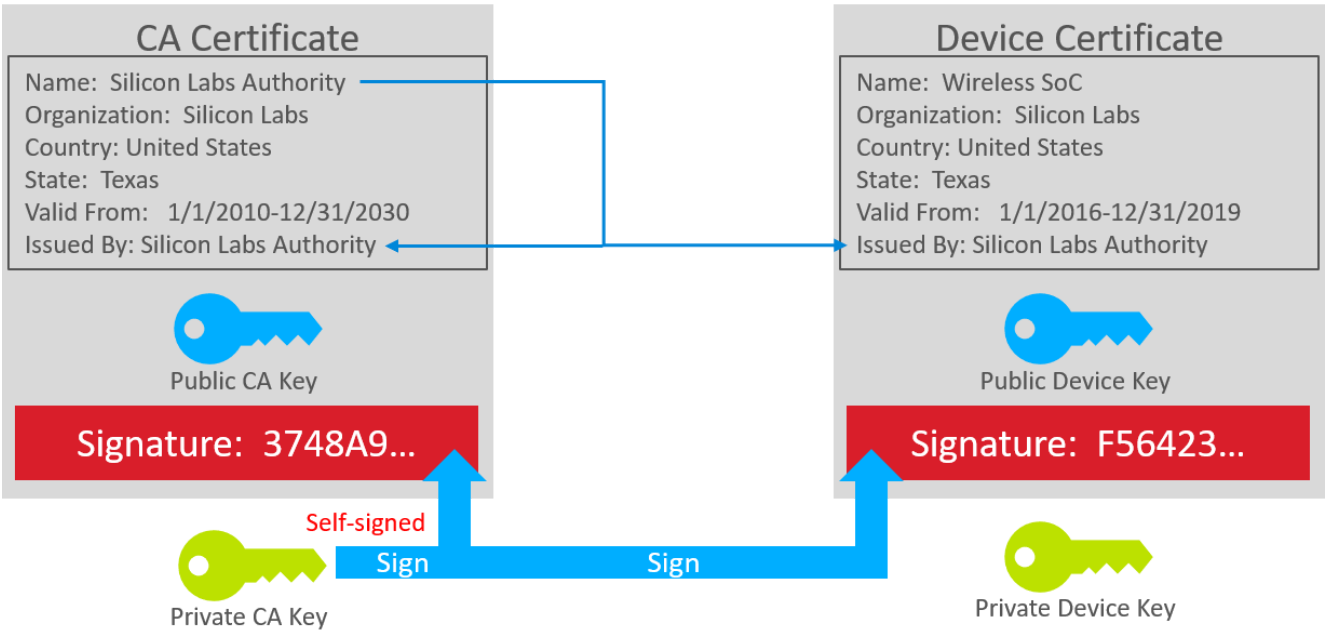
In Digital Signature Example, Bob already had access to Alice's public key, which he trusted. However, it is not always feasible to pre-share a public key with everyone for secure identity verification, and no mechanism exists to revoke or inactivate the public key in case it gets stolen.

A digital certificate is simply a small, verifiable data file that contains identity credentials and a public key. That data is then signed either with the corresponding private key, or a different private key. The digital certificate can be used to prove the ownership of a public key.

- If it is signed using the corresponding private key, it is called a self-signed certificate.
- If it is signed by another private key, the owner of that private key is acting as a Certificate Authority (CA).
- A Certificate Authority (CA) is a trusted third party by both the owner and party relying on the certificate.

Concatenation of digital certificates builds a chain of trust.

- At the root of the chain is a self-signed certificate called a root certificate or a CA certificate.
- The root or CA certificate can be used to sign another certificate.



> **Note**: The private key is never included as part of the certificate – it must be stored separately and kept private. The security of the scheme relies on protecting the private keys.

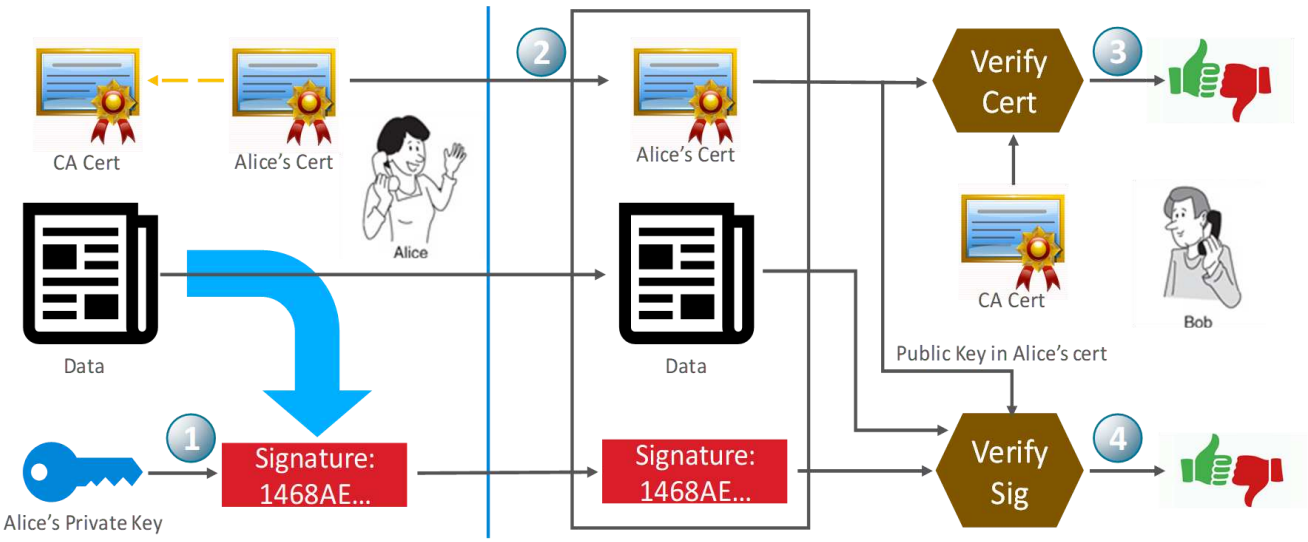
Digital Certificates Verification

This section illustrates the process shown in Digital Signature Example, but using digital certificates.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Digital Certificates Verification Example

Alice wants to give data to Bob, signed with her private key. Alice has a digital certificate signed by a trusted third party (CA) in addition to her private key. Bob has a certificate from the trusted CA but nothing else is previously shared.



1. Alice uses her private key to sign the data.
2. Alice gives the data, signature, and her certificate to Bob.
3. Bob first verifies that Alice's certificate is valid, to prove Alice is the owner of the certificate's public key. This is done by verifying that Alice's certificate contains a valid signature created by the CA.
4. Bob then verifies the signature of the data using the public key in Alice's certificate.

> **Note**: The hash process in Digital Signature Example is skipped in this example.

Copyright © 2025 Silicon Laboratories. All rights reserved.

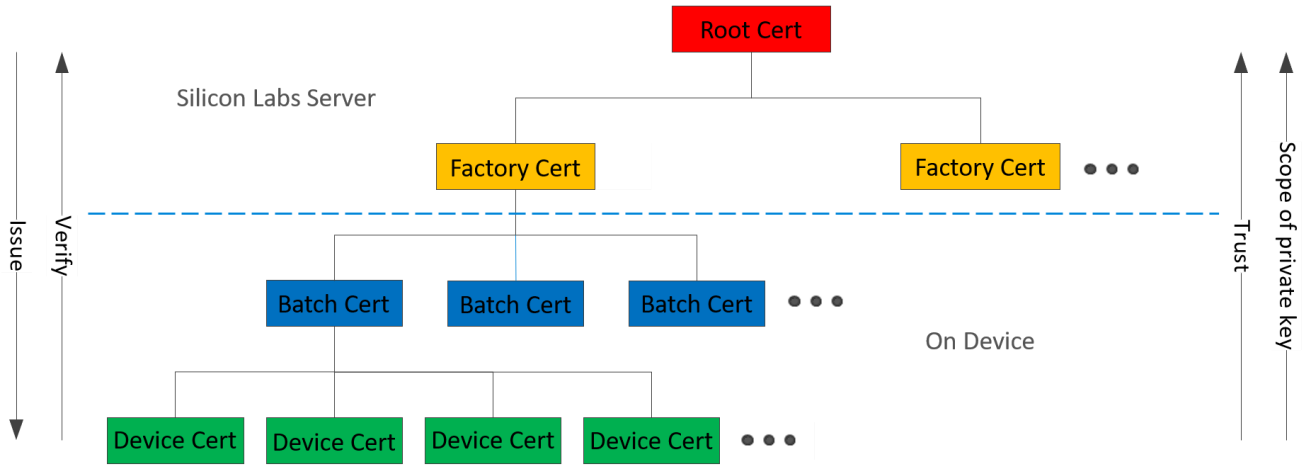
Secure Identification on HSE-SVH Devices

Secure Identification on HSE-SVH Devices

The goal of secure identification is to prove the ownership of a device's unique public key to an external service. It enables the external service to identify the device as legitimate and to authenticate device-generated data or messages.

Chain of Trust

The chain of trust on HSE-SVH devices is illustrated in the following figure.



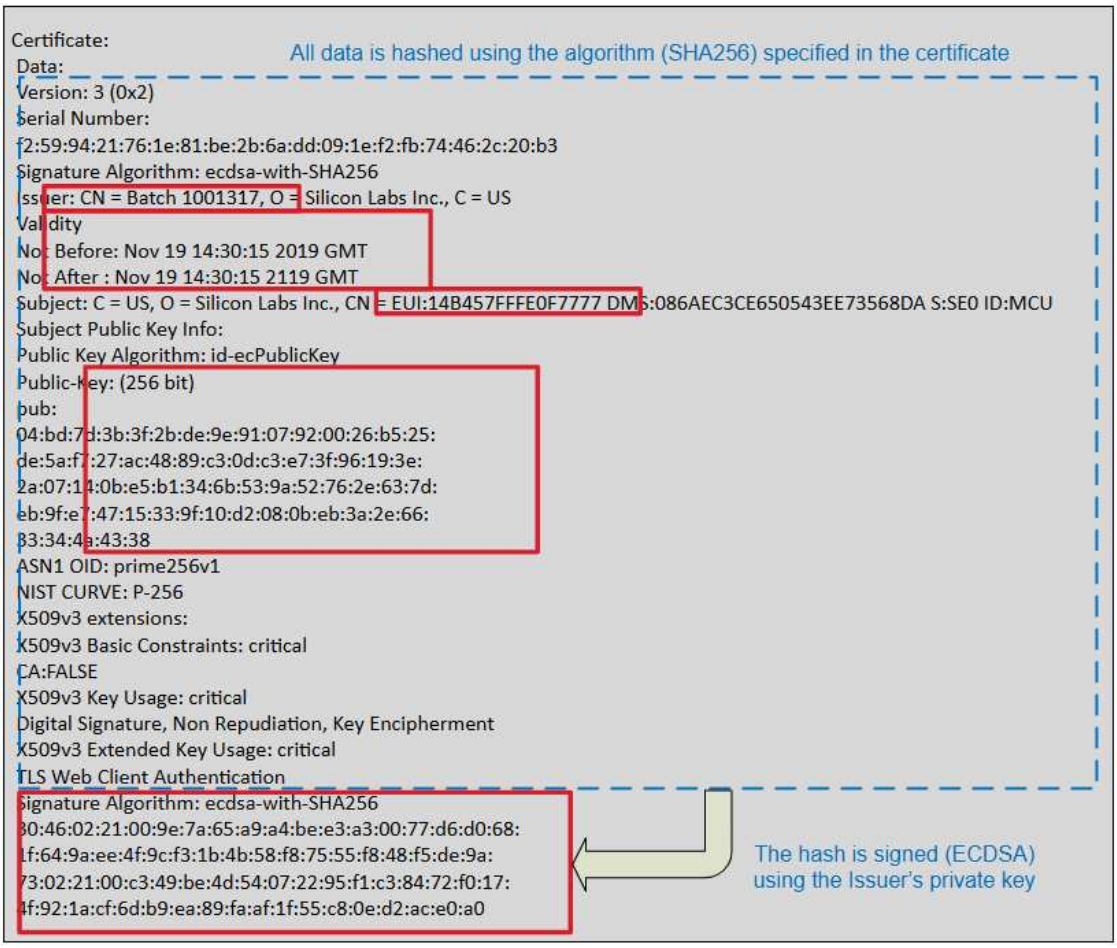
- Silicon Labs is a Certificate Authority (CA).
- The root certificate and factory certificate are stored in the Silicon Labs Server.
- The factory certificate is static per factory.
- The batch certificate and device certificate are stored on the device.
- The batch certificate is rolled per production batch.
- The device certificate is a unique cryptographic identity.
- All certificates are X.509 standard format.
  - TLS-compliant: Standard endpoint authentication methods are used in internet communications
  - Signature algorithm: ECDSA-prime256v1 with SHA256
- Each certificate in the chain is signed by the certificate above it (Signing and Verification figure).

> **Note**: A certificate can be revoked if needed, for instance if security issues arise. The certificate revocation lists are stored in the Silicon Labs Server.

Device Certificate

The device certificate example is described in the following figure.

Copyright © 2025 Silicon Laboratories. All rights reserved.

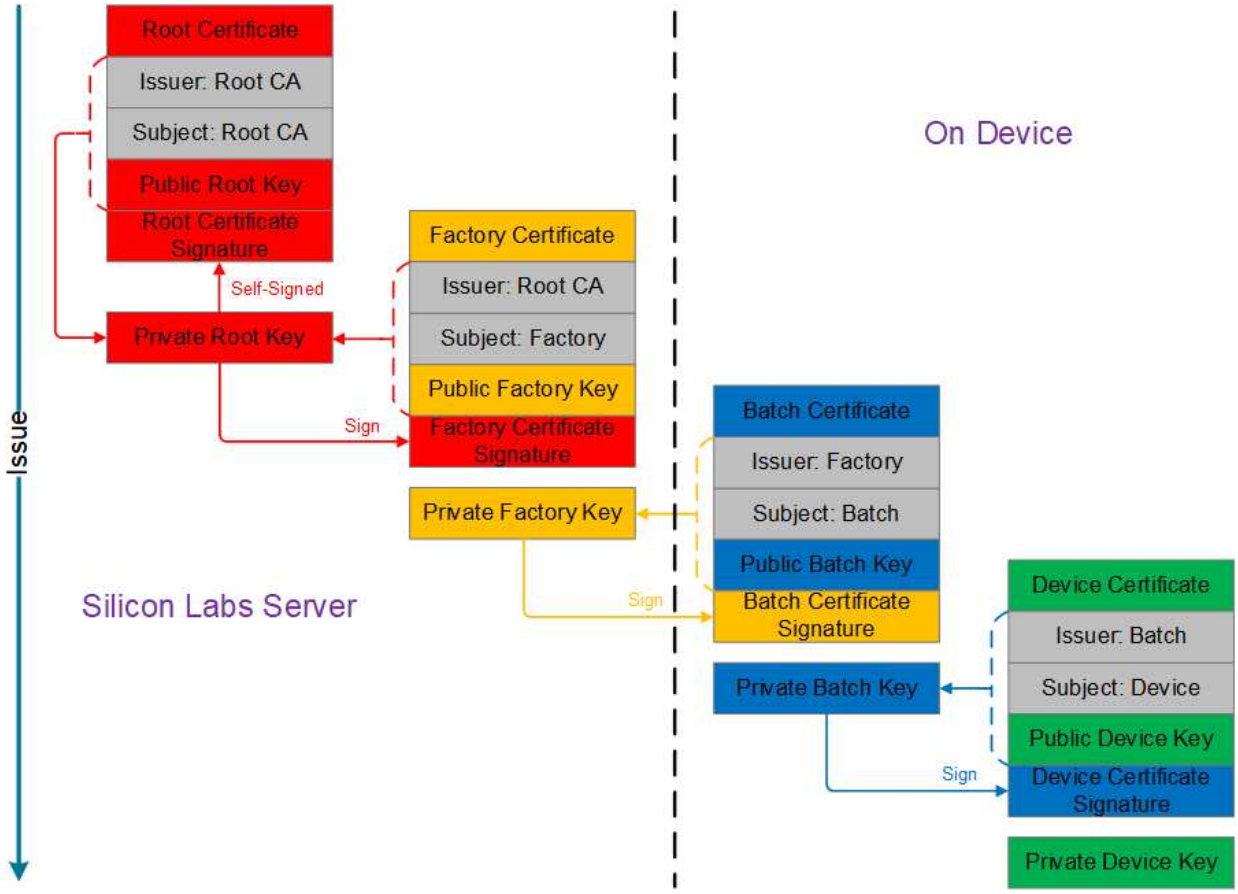


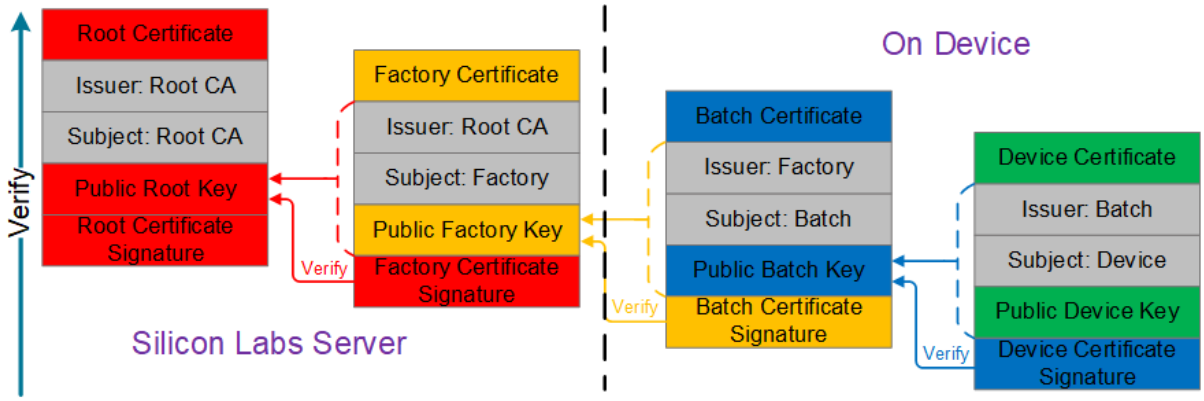
- The device certificate is in X.509 DER format (~0.5 kB).
- The device certificate is stored in HSE one-time programmable memory (OTP). It cannot be modified once programmed.
- The batch number (`Issuer: CN = Batch` field) identifies the factory and batch in which the device was produced.
- The validity period is 100 years from device manufacture date.
- The device 64-bit hard-coded unique ID (EUI) is encoded in the `Subject: CN` field, which blinds this certificate to the device.
- The device-specific public key is embedded in the device certificate and the corresponding private key is securely stored in the Secure Key Storage on the chip.
- The Issuer's private key is used to sign the hash of the certificate data to create a device certificate signature.

Signing and Verification

Signing and verification for certificates on HSE-SVH devices are described in the following figures.

Copyright © 2025 Silicon Laboratories. All rights reserved.





Copyright © 2025 Silicon Laboratories. All rights reserved.

Device Certificate Options

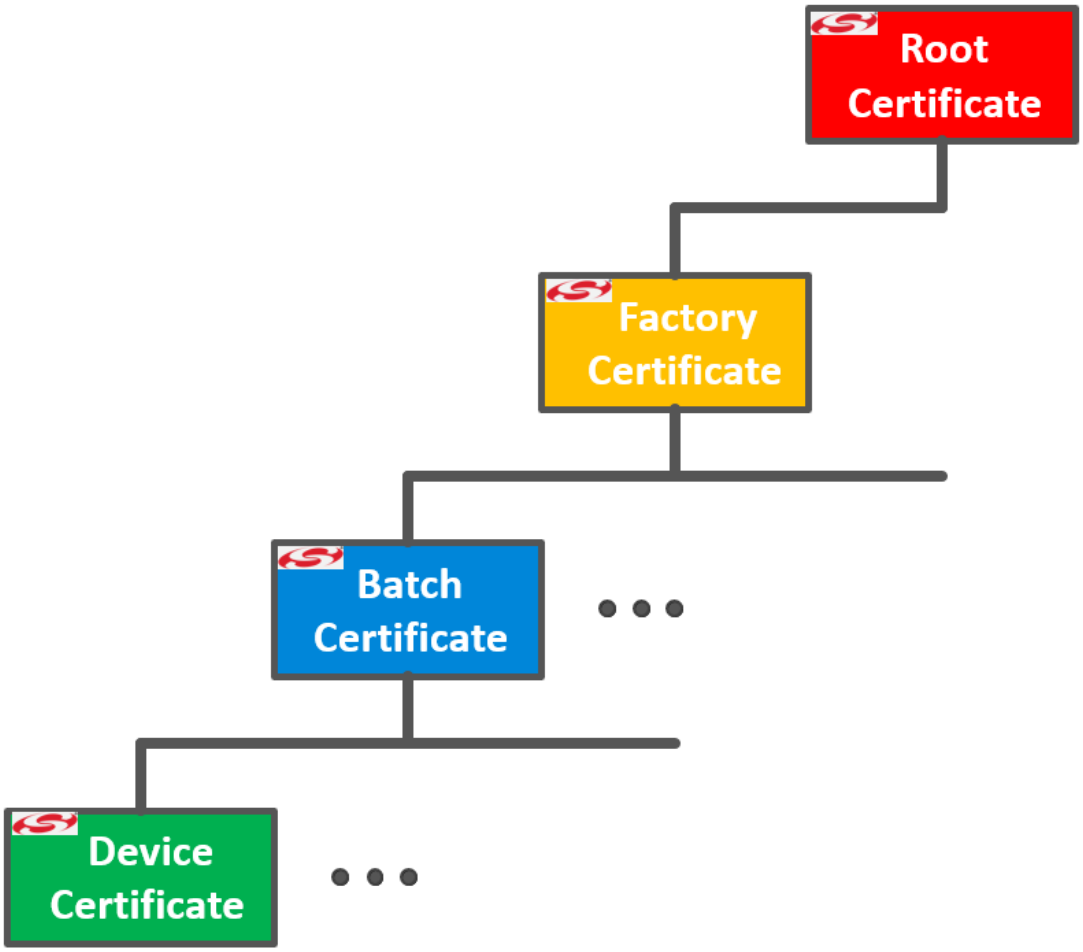
Device Certificate Options

The HSE-SVH devices are each programmed with a device certificate during IC production. The device certificate is signed with a Public Device Key, using a Private Batch Key that can be validated against a Silicon Labs certificate chain Verification for Certificates and Certificate Chain Verification. The device private key never leaves the Secure Key Storage on the chip. Customers can create their own device certificates during their production.

Three device certificate options (standard, modified, and external) are provided to meet different requirements. Silicon Labs provides Custom Part Manufacturing Service (CPMS) to program custom certificates on your chips at the Silicon Labs factories. For more information about CPMS, see UG519: Custom Part Manufacturing Service User's Guide.

Standard Device Certificate

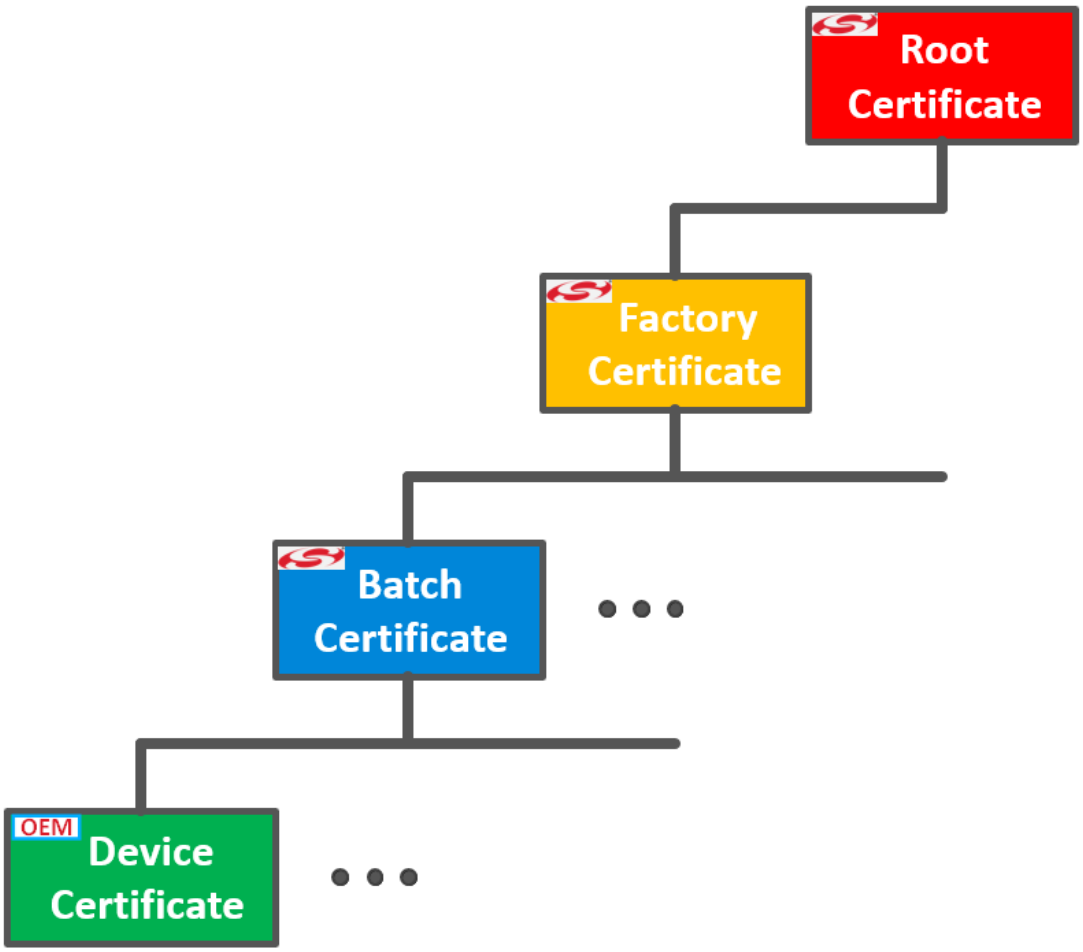
- Comes standard with HSE-SVH devices.
- Cryptographically proves the device is an authentic Silicon Labs device.
- Does not protect against overproduction or counterfeit products that are built with authentic Silicon Labs devices.
- Signed to a Silicon Labs Certificate Authority (CA).
- The device can prove that it possesses the private key associated with the public key in its certificate by signing the response to a given challenge (Remote Authentication Process and Certificate Chain Verification and Remote Authentication).



Copyright © 2025 Silicon Laboratories. All rights reserved.

Modified Device Certificate

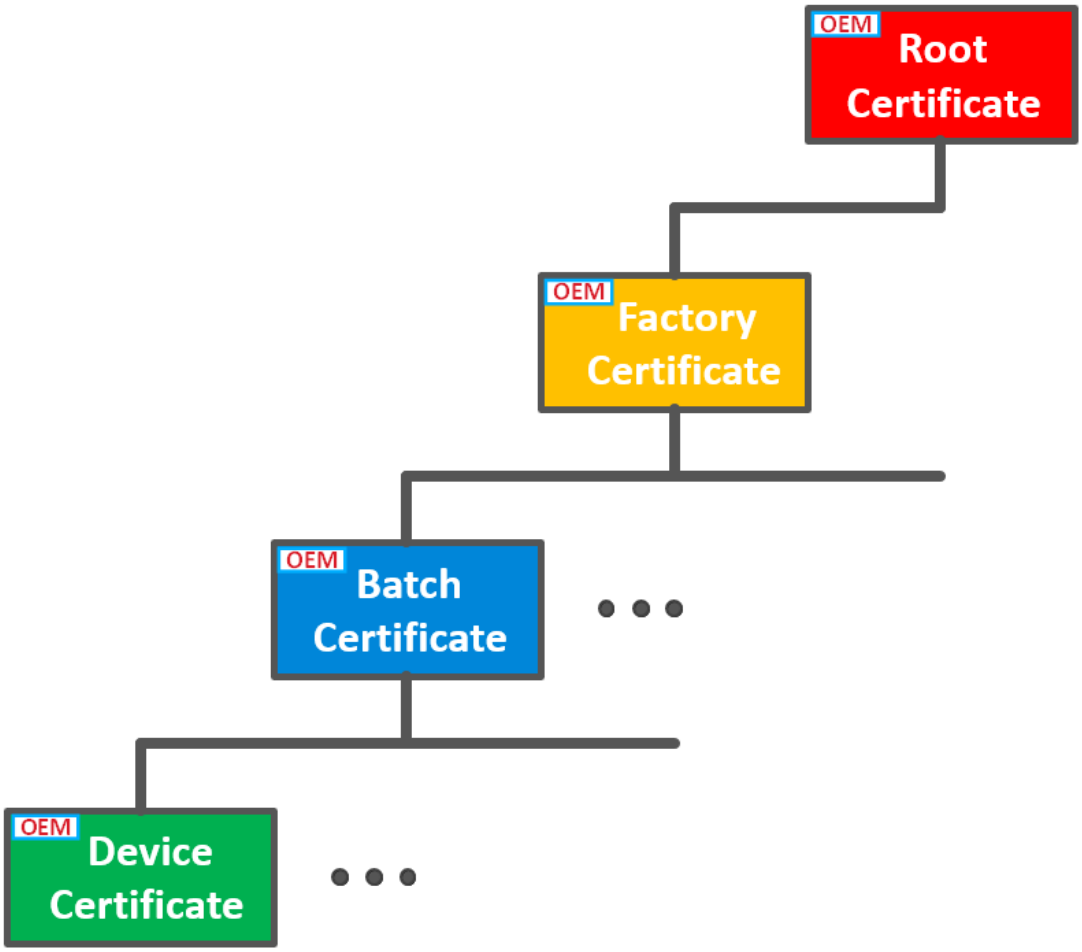
- Available as a customization service on HSE-SVH devices (OEM custom part number).
- Cryptographically proves the device is an authentic Silicon Labs device that was produced for a specific OEM.
- Protects against overproduction by Contract Manufacturer (CM).
- Device Certificate X.509 fields can be specified, with restrictions.
- Signed to a Silicon Labs Certificate Authority (CA).



External Device Certificate

- Available as a customization service on HSE-SVH devices (OEM custom part number).
- Cryptographically proves the device is an authentic Silicon Labs device that was produced for a specific OEM.
- Protects against overproduction by Contract Manufacturer (CM).
- Factory Certificate is custom for each OEM.
- Device Certificate and Factory Certificate X.509 fields can be specified, with restrictions.
- Signed to a OEM Certificate Authority (CA).
- Root Certificate Authority is OEM-specified and is optional.
- Electronic delivery of all batch and device certificates signed under this OEM factory certificate is supported.

Copyright © 2025 Silicon Laboratories. All rights reserved.



Copyright © 2025 Silicon Laboratories. All rights reserved.

Entity Attestation Token (EAT)

Entity Attestation Token (EAT)

The device attestation service creates a token that contains a fixed set of device-specific data when requested from the caller. The device must contain an attestation key pair, which is unique per device, to sign the token. The HSE-SVH device uses the Private Device Key (aka attestation key) to sign the token, and the caller uses the Public Device Key to verify the token's authenticity.

An Entity Attestation Token (EAT) is a mini-report that is cryptographically signed. An EAT is encoded in either one of two standardized data formats: a Concise Binary Object Representation (CBOR) or in the text-based format JSON. A digital signature is then used to protect its content. The technical specification defining the content of the EAT, which are claims about the hardware and the software running on a device, is specified by the Internet Engineering Task Force (IETF).

An EAT is a collection of Key ID-Value pairs relating to device pedigree or any other information one wants the device to attest. Collected data can originate from the Root of Trust (RoT), any protected area, or non-protected areas.

The EAT must be signed following the structure of the CBOR Object Signing and Encryption (COSE) specification. For asymmetric key algorithms, the signature structure must be COSE-Sign1. A COSE-Sign1 is a CBOR encoded, self-secured data blob that contains headers, a payload, and a signature.

The primary need for EAT verification is to check correct formatting and verify signatures as for any token. In addition, though, the verifier can operate a policy where values of some of the claims in this profile can be compared to reference values, registered with the verifier for a given deployment, to confirm that the device is endorsed by the manufacturer supply chain.

The HSE can generate the PSA attestation token or security configuration token when requested from the caller with a challenge (Auth challenge claim below). The following tables describe EAT claims that are used in the PSA attestation token and security configuration token.

> **Note**: The actual claims returned from the tokens are HSE firmware version dependent.

Claims of PSA Attestation Token

Key ID	Claim	Description	Value
-75000	Profile definition	Name of a document that describes the profile of the report.	PSA_IOT_PROFILE_1
-75001	Client ID	Represents the Partition ID of the caller.	See note below
-75002	Security lifecycle	Represents the current life cycle stage of the PSA RoT.	Device dependent
-75003	Implementation ID	Uniquely identifies the underlying immutable PSA RoT.	Device dependent (32 bytes)
-75004	Boot seed	Represents a random value created at system boot time.	Random bytes (32 bytes)
-75006	Software components	A list of Software components represents all the software loaded by PSA RoT.	See the software components table below.
-75008	Auth challenge	Input object from the caller. For example, this can be a cryptographic nonce or a hash of locally attested data. The length must be 32, 48, or 64 bytes.	Random bytes or hash (32/48/64 bytes)

Copyright © 2025 Silicon Laboratories. All rights reserved.

Key ID	Claim	Description	Value
-75009	Instance ID	Unique identifier of the instance.	Device EUI-64 unique ID with type byte 0x06 (9 bytes)

Note:

- Key ID 75001: Client ID if present. Otherwise the value 1 for a token requested by a secure bus master and -1 for a non-secure master.
- Key ID 75002 (For the definitions of these lifecycle states, please refer to the ARM Platform Security Model):
  - UNKNOWN (0x0000)
  - ASSEMBLY_AND_TEST (0x1000)
  - PSA_ROT_PROVISIONING (0x2000)
  - SECURED (0x3000)
  - NON_PSA_ROT_DEBUG (0x4000)
  - RECOVERABLE_PSA_ROT_DEBUG (0x5000)
  - DECOMMISSIONED (0x6000)
- Key ID 75003:
  - Word[0]: Die revision
  - Word[1]: HSE OTP version
  - Word[2]: Bit indicating it is an HSE-SVH device
  - Word[3]: Production version
  - Word[4:7]: Reserved (zeros)

Software Components

Key ID	Type	Description	Value
1	Measurement type	A short string represents the role of this software component.	See note below
2	Measurement value	Represents a hash of the invariant software component in memory at startup time.	See note below
4	Version	The issued software version is in the form of a text string.	See note below

Notes:

- Key ID 1:
  - HSE always exists — **PRoT**
  - If secure booted Gecko Bootloader exists at flash starting address — **BL**
  - If secure booted application exists at flash starting address — **ARoT**
- Key ID 2: SHA-256 hash (32 bytes) of the firmware (HSE, Gecko Bootloader, or application)
- Key ID 4: Version of the firmware (HSE, Gecko Bootloader, or application)

Claims of Security Configuration Token

Key ID	Claim	Description	Value
-75000	Profile definition	Name of a document that describes the profile of the report.	SILABS_1
-75008	Auth challenge	Input object from the caller. For example, this can be a cryptographic nonce or a hash of locally attested data. The length must be 32 bytes.	Random bytes or hash (32 bytes)
-75009	Instance ID	Unique identifier of the instance.	Device EUI-64 unique ID with type byte 0x06 (9 bytes)
-76000	SE status	Device HSE status.	Device dependent (36 bytes)

Copyright © 2025 Silicon Laboratories. All rights reserved.

Key ID	Claim	Description	Value
-76001	OTP configuration	Device HSE OTP configuration if provisioned.	Device dependent (24 bytes)
-76002	Sign Key	Public Sign Key in HSE OTP if provisioned.	Device dependent (64 bytes)
-76003	Command Key	Public Command Key in HSE OTP if provisioned.	Device dependent (64 bytes)
-76004	Tamper settings	Current applied tamper settings.	Device dependent (16 bytes)

Notes:

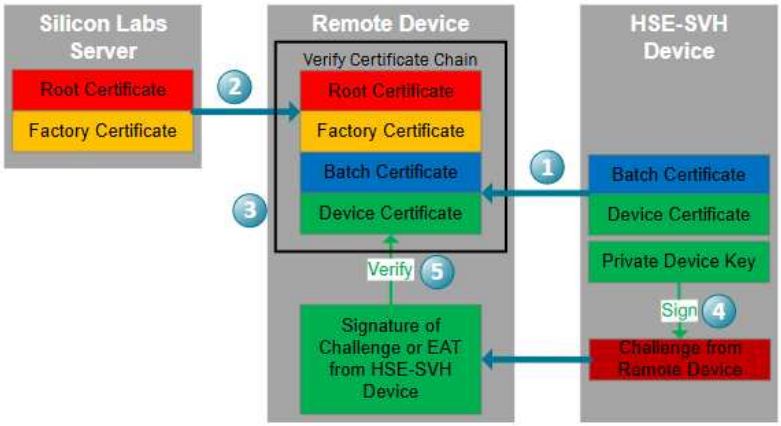
- All custom Silicon Labs claims will have a base of 76000.
- Key ID 76000: Refer to section "Get Status" in AN1303: Programming Series 2 Devices using the Debug Challenge Interface (DCI) and Serial Wire Debug (SWD) for the description (HSE-SVH) of the value.
- Key ID 76001: Refer to section "Read User Configuration" in AN1303: Programming Series 2 Devices using the Debug Challenge Interface (DCI) and Serial Wire Debug (SWD) for the description (HSE-SVH) of the value.
- Key ID 76002 and 76003: Refer to Key Reference for Public Sign Key and Public Command Key.
- Key ID 76004: One nibble per tamper source. Refer to section "Anti-Tamper Configuration" in AN1303: Programming Series 2 Devices using the Debug Challenge Interface (DCI) and Serial Wire Debug (SWD) for the description of the value.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Remote Authentication Process

Remote Authentication Process

Remote authentication is used to manage attestation by requesting that the device sign a challenge or EAT based on its secure identity.



1. The remote device requests the device certificate and batch certificate from the HSE-SVH device.
2. The remote device looks up the factory certificate and root certificate from the Silicon Labs Server.
3. The remote device validates each certificate in the chain using the public key of each Issuer (Verification for Certificates).
4. The remote device then sends an attestation challenge (random number) to the HSE-SVH device. The HSE-SVH device uses the Private Device Key in the Secure Key Storage on the chip to sign the challenge or EAT and sends the signature of challenge or EAT to the remote device.
5. The remote device requires a small library to validate the signature of challenge or EAT using the Public Device Key in the device certificate.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Secure Engine Manager

Secure Engine Manager

The Secure Engine Manager provides thread-safe APIs for the SE's mailbox interface. The following table lists the SE Manager APIs related to secure identity. The SE Manager API document can be found at https://docs.silabs.com/gecko-platform/latest/service/api/group-sl-se-manager.

For the SE's mailbox interface, see section *Secure Engine Subsystem* in Series 2 Secure Debug.

SE Manager API for Security Identity

SE Manager API	Usage
sl_se_read_pubkey	Read stored Public Device Key in the HSE-SVH device.
sl_se_read_cert	Read stored certificates (DER format) in the HSE-SVH device.
sl_se_read_cert_size	Read the size of stored certificates in the HSE-SVH device.
sl_se_attestation_get_psa_iat_token	Get the PSA attestation token from the HSE with the given nonce.
sl_se_attestation_get_psa_iat_token_size	Get the size of a PSA attestation token with the given nonce.
sl_se_attestation_get_config_token	Get the security configuration token from the HSE with the given nonce.
sl_se_attestation_get_config_token_size	Get the size of a security configuration token with the given nonce.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Examples

Examples

Overview

The secure device authentication examples are described in the following table.

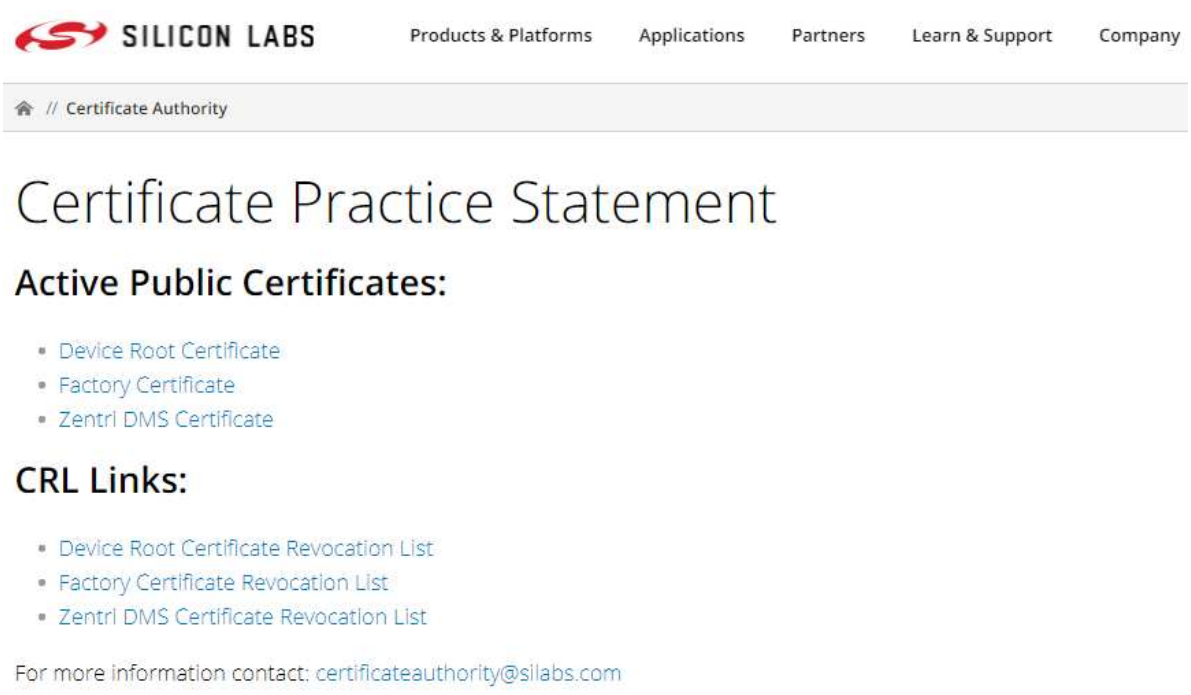
Secure Device Authentication Examples

Example	Device (Radio Board)	HSE Firmware	Tool
Certificate chain verification	EFR32MG21B010F1024IM32 (BRD4181C)	Version 1.2.9	Simplicity Commander and OpenSSL
Certificate chain verification	EFR32MG21B010F1024IM32 (BRD4181C)	Version 1.2.9	Simplicity Commander
Certificate chain verification	EFR32MG21B010F1024IM32 (BRD4181C)	Version 1.2.9	Simplicity Studio 5
Certificate chain verification & Remote authentication	EFR32MG21B010F1024IM32 (BRD4181C)	Version 1.2.9	SE Manager and Mbed TLS
Entity Attestation Token (EAT)	EFR32MG21B010F1024IM32 (BRD4181C)	Version 1.2.9	SE Manager
Entity Attestation Token (EAT)	EFR32MG21B010F1024IM32 (BRD4181C)	Version 1.2.9	Simplicity Commander

> **Note**: Unless specified in the example, these examples can apply to other HSE-SVH devices.

Users can download the device root certificate (`Device-Root-CA-chain.pem`) and factory certificate (`Factory-chain.pem`) from https://www.silabs.com/certificate-authority.

Copyright © 2025 Silicon Laboratories. All rights reserved.



For Simplicity Studio v5.3.0.0 and higher, the device root certificate (`device-root-prod.pem`) and factory certificate (`factory-prod.pem`) can be found in the Window folder below.

C:\SiliconLabs\SimplicityStudio\v5\offline\common\certificates

Using Simplicity Commander

1. This application note uses Simplicity Commander v1.11.2. The procedures and console output may be different on the other versions of Simplicity Commander. The latest version of Simplicity Commander can be downloaded from https://www.silabs.com/developers/mcu-programming-options.

```
commander --version
```

```
Simplicity Commander 1v11p2b998

JLink DLL version: 6.94d
Qt 5.12.1 Copyright (C) 2017 The Qt Company Ltd.
EMDLL Version: 0v17p18b581
mbed TLS version: 2.6.1


DONE
```

2. The Simplicity Commander's Command Line Interface (CLI) is invoked by `commander.exe` in the Simplicity Commander folder. The location for Simplicity Studio 5 in Windows is `C:\SiliconLabs\SimplicityStudio\v5\developer\adapter_packs\commander` . For ease of use, it is highly recommended to add the path of `commander.exe` to the system `PATH` in Windows.
3. If more than one Wireless Starter Kit (WSTK) is connected via USB, the target WSTK must be specified using the `--serialno \<J-Link serial number>` option.
4. If the WSTK is in debug mode OUT, the target device must be specified using the `--device \<device name>` option.

For more information about Simplicity Commander, see UG162: Simplicity Commander Reference Guide.

Using an External Tool

The certificate chain verification example uses the **OpenSSL** to validate the certificate chain. The Windows version of OpenSSL can be downloaded from https://slproweb.com/products/Win32OpenSSL.html. This application note uses OpenSSL

Copyright © 2025 Silicon Laboratories. All rights reserved.

Version 1.1.1h (Win64).

```
openssl version
```

```
OpenSSL 1.1.1h  22 Sep 2020
```

The OpenSSL's Command Line Interface (CLI) is invoked by `openssl.exe` in the OpenSSL folder. The location in Windows (Win64) is `C:\Program Files\OpenSSL-Win64\bin`. For ease of use, it is highly recommended to add the path of `openssl.exe` to the system `PATH` in Windows.

Using Platform Examples

Simplicity Studio 5 includes the SE Manager platform examples for secure identity and attestation. This application note uses platform example of GSDK v3.2.3. The console output may be different on the other versions of GSDK.

Refer to the corresponding `readme.html` file for details about each SE Manager platform example. This file also includes the procedures to create the project and run the example.

Certificate Chain Verification

Certificate chain verification is the process of making sure a given certificate chain is well-formed, valid, properly signed, and trustworthy. The certificate signature is verified using the public key in the issuer certificate (Verification for Certificates).

Simplicity Commander and OpenSSL

1. Run the `security readcert` command to save the batch certificate in PEM format.

   ```
   commander security readcert batch -o batch.pem --serialno 440030580
   ```

   ```
   Writing certificate to batch.pem...
   DONE
   ```

2. Run the `security readcert` command to save the device certificate in PEM format.

   ```
   commander security readcert mcu -o device.pem --serialno 440030580
   ```

   ```
   Writing certificate to device.pem...
   DONE
   ```

3. Get the root certificate (`device-root-prod.pem`) and factory certificate (`factory-prod.pem`) from the certificate folder in Simplicity Studio.
4. Use OpenSSL to display the certificate information (e.g., `device.pem`).

   ```
   openssl x509 -in device.pem -text -noout
   ```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            66:f8:5a:e6:b4:ef:6e:49:d3:36:95:63:c9:c3:99:13:e4:71:93:f6
        Signature Algorithm: ecdsa-with-SHA256
        Issuer: CN = Batch 1001317, O = Silicon Labs Inc., C = US
        Validity
            Not Before: Nov 19 15:10:33 2019 GMT
            Not After : Nov 19 15:10:33 2119 GMT
        Subject: C = US, O = Silicon Labs Inc., CN = EUI:14B457FFFE0F77CE DMS:086AEC3C645836BFB04D312F S:SE0 ID:MCU
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:5c:4b:c9:b0:b3:ff:fa:99:81:c5:99:be:ff:ae:
                    77:74:1a:f4:30:f1:1e:0e:2d:df:96:4b:ff:d2:46:
                    fa:fa:e7:23:4b:79:cb:0a:c7:71:13:fa:7c:39:5f:
                    e2:18:9e:4e:06:43:88:a7:9c:65:53:f3:a3:a1:06:
                    81:e6:06:f2:11
                ASN1 OID: prime256v1
                NIST CURVE: P-256
        X509v3 extensions:
            X509v3 Basic Constraints: critical
                CA:FALSE
            X509v3 Key Usage: critical
                Digital Signature, Non Repudiation, Key Encipherment
            X509v3 Extended Key Usage: critical
                TLS Web Client Authentication
    Signature Algorithm: ecdsa-with-SHA256
        30:44:02:20:57:12:a4:84:d8:37:b8:c0:44:8f:16:ac:c1:a3:
        be:a9:f1:16:38:9f:b9:a2:57:e6:12:49:bf:96:a9:a9:d2:b8:
        02:20:5f:ae:22:f5:00:05:49:b1:da:ee:4a:84:48:70:27:97:
        1c:40:2d:85:5f:f2:12:b3:8b:4a:d7:9a:ee:60:81:7c
```

5. Use OpenSSL to verify the certificate chain from steps 1 to 3.

```
openssl verify -show_chain -CAfile device-root-prod.pem -untrusted factory-prod.pem -untrusted batch.pemdevice.pem
```

```
device.pem: OKChain:
depth=0: C = US, O = Silicon Labs Inc., CN = EUI:14B457FFFE0F7777 DMS:086AEC3CE650543EE73568DA S:SE0 ID:MCU (untrusted)
depth=1: CN = Batch 1001317, O = Silicon Labs Inc., C = US (untrusted)
depth=2: CN = Factory, O = Silicon Labs Inc., C = US (untrusted)
depth=3: CN = Device Root CA, O = Silicon Labs Inc., C = US
```

Simplicity Commander

Run the `security readcert` command to display the key information about the on-chip certificates (e.g., mcu).

```
commander security readcert mcu --serialno 440030580
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
Version          : 3
Subject          : C=US  O=Silicon Labs Inc.  CN=EUI:14B457FFFE0F77CE DMS:086AEC3C645836BFB04D312F S:SE0 ID:MCU
Issuer           : CN=Batch 1001317  O=Silicon Labs Inc.  C=US
Valid From       : November 19 2019
Valid To         : November 19 2119
Signature algorithm: SHA256
Public Key Type    : ECDSA
Public key       : 5c4bc9b0b3fffa9981c599beffae77741af430f11e0e2ddf964bffd246fafae7
                   234b79cb0ac77113fa7c395fe2189e4e064388a79c6553f3a3a10681e606f211
DONE
```

Run the `security attestation` command to verify the on-chip batch and device certificates with root and factory certificates.

```
commander security attestation --serialno 440030580
```

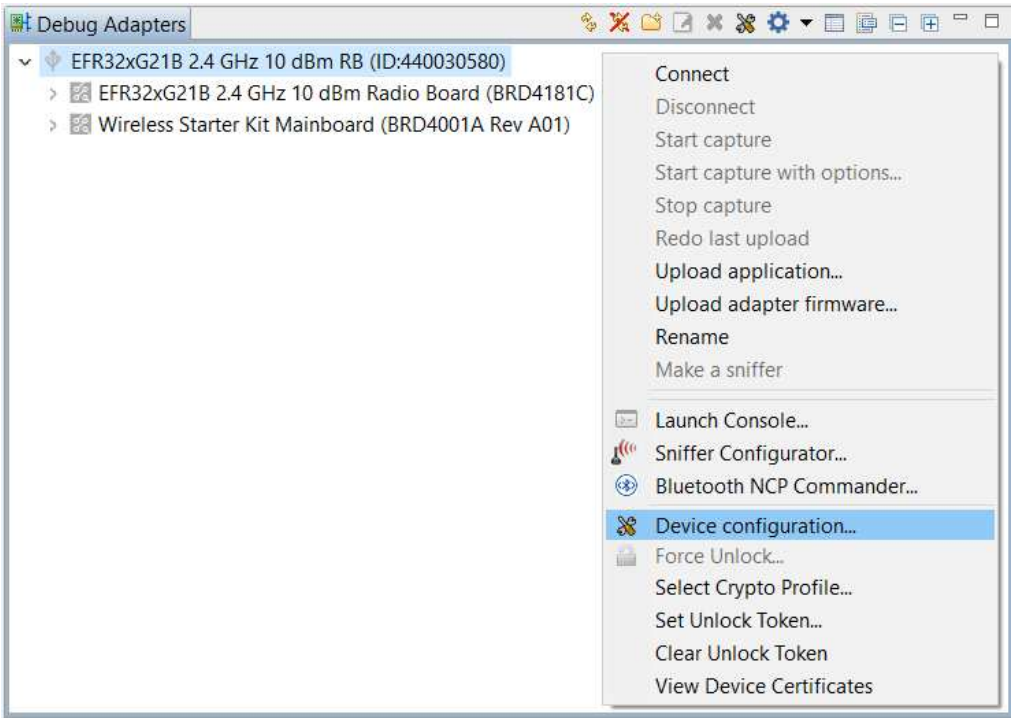
```
Certificate chain successfully validated up to Silicon Labs device root certificate.
-75008 ARM PSA nonce                : 05a88aeef627dd663058e3d758fe9a827942da0793da72af81c79a4f60fa9824
-75000 ARM PSA Profile ID           : SILABS_1
-75009 ARM PSA/IETF EAT UEID        : 0614b457fffe0f77ce
-76000 SE status                    : 00000001000000000000000000000003a90000002000010209ffffffff0000002500000000
-76001 OTP configuration            : 00000000100444400401041411224477242204420a060005
-76002 MCU sign key                 : c4af4ac69aab9512db50f7a26ae5b4801183d85417e729a56da974f4e08a562c
                                      de6019dea9411332dc1a743372d170b436238a34597c410ea177024de20fc819
-76003 MCU command key              : b1bc6f6fa56640ed522b2ee0f5b3cf7e5d48f60be8148f0dc08440f0a4e1dca4
                                      7c04119ed6a1be31b7707e5f9d001a659a051003e95e1b936f05c37ea793ad63
-76004 Current applied tamper settings : 150444400401041411224477142204 42
Successfully validated signature of attestation token.
DONE
```

Simplicity Studio

This application note uses Simplicity Studio v5.2.1.1. The procedures and pictures may be different on the other versions of Simplicity Studio 5.

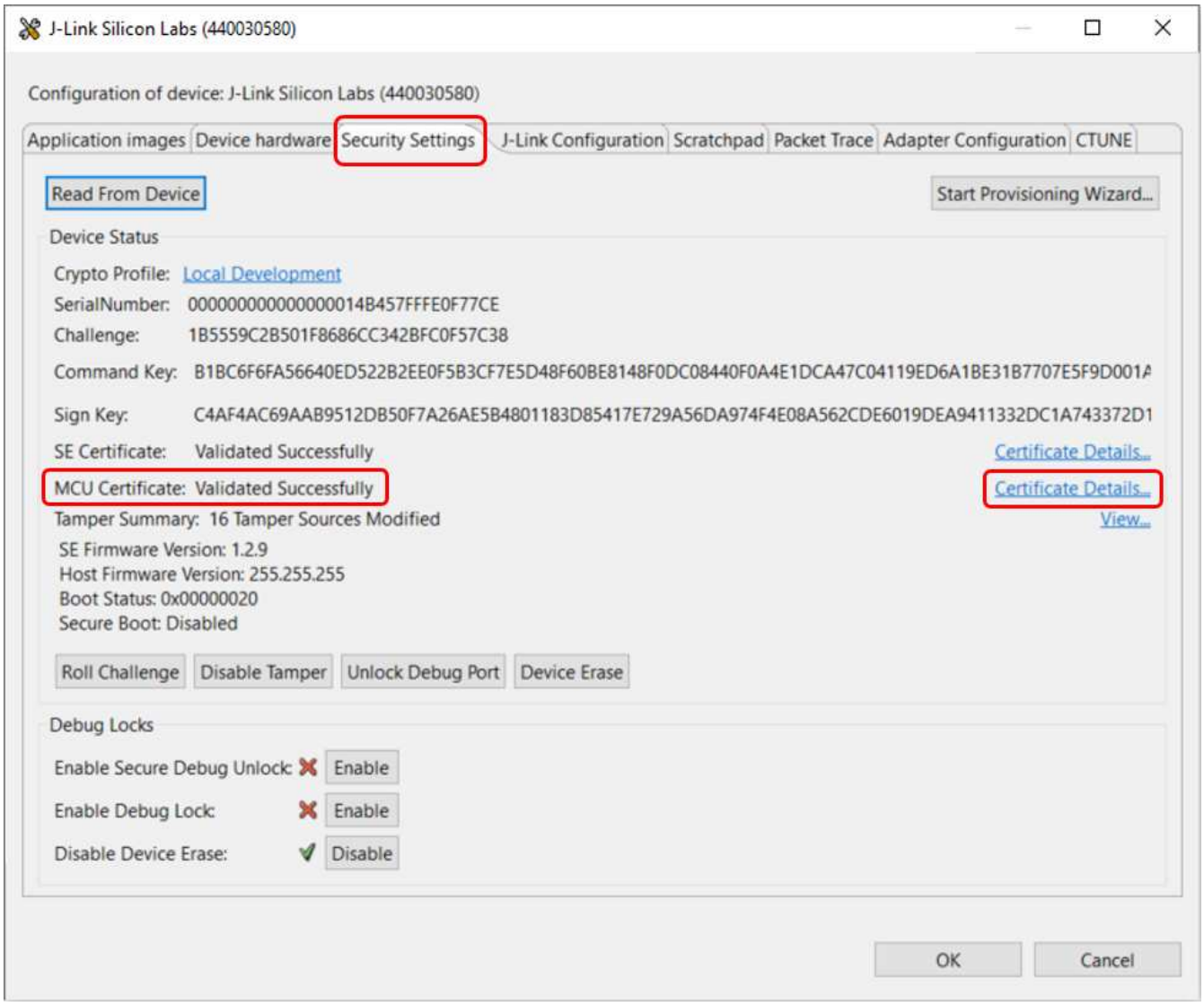
1. Right-click the selected debug adapter **RB (ID:J-Link serial number)** to display the context menu.

Copyright © 2025 Silicon Laboratories. All rights reserved.

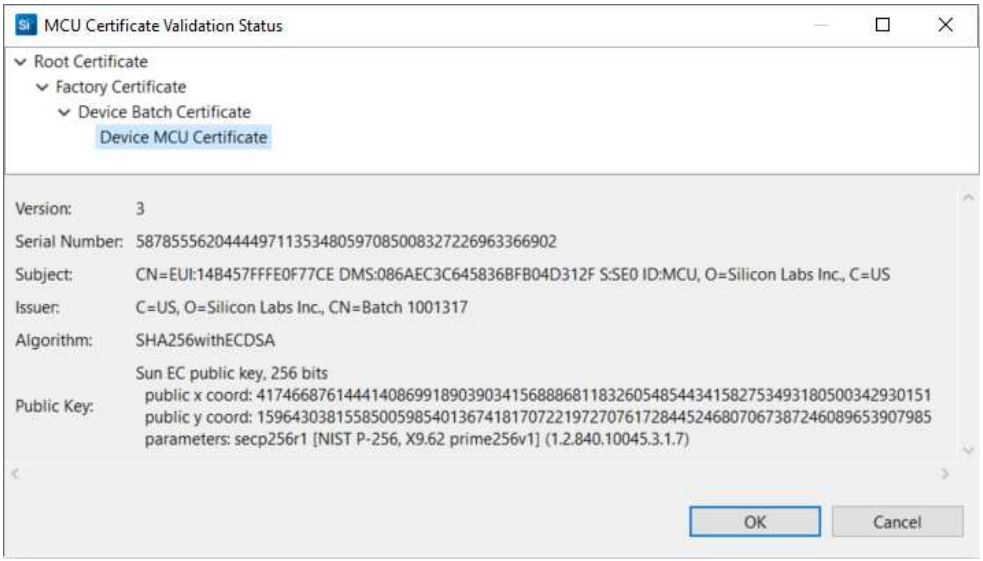


2. Click **Device configuration...** to open the **Configuration of device: J-Link Silicon Labs (serial number)** dialog box. Click the **Security Settings** tab to get the selected device configuration.
3. The **MCU Certificate:** will display **Validated Successfully** if it passed the certificate chain verification process.

Copyright © 2025 Silicon Laboratories. All rights reserved.



4. Click **Certificate Details...** to browse the details of different certificates (e.g., Device MCU Certificate in the figure below).

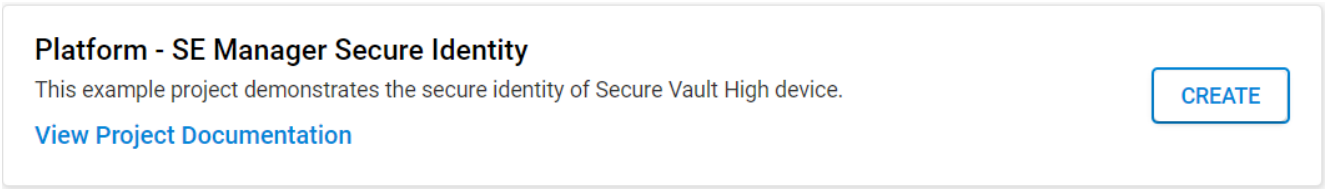


Copyright © 2025 Silicon Laboratories. All rights reserved.

Certificate Chain Verification and Remote Authentication

The SE Manager Secure Identity platform example uses APIs in SE Manager and Mbed TLS to emulate the processes in Remote Authentication Process.

Click the `View Project Documentation` link to open the `readme.html` file.



The HSE-SVH device simulates the operations in the remote device to eliminate the communications between different parties in this example. The factory certificate and root certificate are hard-coded in the `app_mbedtls_x509.c` file.

The Private Device Key in the Secure Key Storage on the chip is used to sign the challenge from the remote device. Therefore this example can only run on a chip with the Standard Device Certificate.

Step 1 in the Remote Authentication Process

```
SE Manager Secure Identity Example - Core running at 38000 kHz.
  . SE manager initialization... SL_STATUS_OK (cycles: 6 time: 0 us)

  . Secure Vault High device:
  + Read size of on-chip certificates... SL_STATUS_OK (cycles: 5296 time: 139 us)
  + Read on-chip device certificate... SL_STATUS_OK (cycles: 5138 time: 135 us)
  + Parse the device certificate (DER format)... SL_STATUS_OK (cycles: 167043 time: 4395 us)
  + Get the public device key in device certificate... OK
  + Read on-chip batch certificate... SL_STATUS_OK (cycles: 5080 time: 133 us)
  + Parse the batch certificate (DER format)... SL_STATUS_OK (cycles: 173151 time: 4556 us)
```

Steps 2 and 3 in the Remote Authentication Process (certificate chain printout is disabled)

```
. Remote device:
  + Parse the factory certificate (PEM format)... SL_STATUS_OK (cycles: 5373122 time: 141 ms)
  + Parse the root certificate (PEM format)... SL_STATUS_OK (cycles: 5448802 time: 143 ms)
  + Verify the certificate chain with root certificate... SL_STATUS_OK (cycles: 958730 time: 25229 us)
```

Steps 2 and 3 in the Remote Authentication Process (certificate chain printout is enabled)

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
. Remote device:
  + Parse the factory certificate (PEM format)... SL_STATUS_OK (cycles: 5373935 time: 141 ms)
  + Parse the root certificate (PEM format)... SL_STATUS_OK (cycles: 5449622 time: 143 ms)
  + Verify requested for (Depth 3) ... OK
      cert. version     : 3
      serial number     : 12:E6:A2:A5:9C:AA:27:F9
      issuer name       : CN=Device Root CA, O=Silicon Labs Inc., C=USsubject name      : CN=Device Root CA, O=Silicon Labs Inc., C=US
      issued  on        : 2018-10-10 17:32:00
      expires on        : 2118-09-16 17:32:00
      signed using      : ECDSA with SHA256
      EC key size       : 256 bits
      basic constraints : CA=true, max_pathlen=2
      key usage         : Digital Signature, Key Cert Sign, CRL Sign
  + Verify requested for (Depth 2) ... OK
      cert. version     : 3
      serial number     : 24:DC:7B:40:0C:32:9C:0A
      issuer name       : CN=Device Root CA, O=Silicon Labs Inc., C=USsubject name      : CN=Factory, O=Silicon Labs Inc., C=US
      issued  on        : 2018-10-10 17:33:00
      expires on        : 2118-09-16 17:32:00
      signed using      : ECDSA with SHA256
      EC key size       : 256 bits
      basic constraints : CA=true, max_pathlen=1
      key usage         : Digital Signature, Key Cert Sign, CRL Sign
  + Verify requested for (Depth 1) ... OK
      cert. version     : 3
      serial number     : 23:09:DA:39:B4:78:05:AA
      issuer name       : CN=Factory, O=Silicon Labs Inc., C=USsubject name      : CN=Batch 1001317, O=Silicon Labs Inc., C=US
      issued  on        : 2019-10-17 21:20:20
      expires on        : 2118-09-16 17:32:00
      signed using      : ECDSA with SHA256
      EC key size       : 256 bits
      basic constraints : CA=true, max_pathlen=0
      key usage         : Digital Signature, Key Cert Sign
  + Verify requested for (Depth 0) ... OK
      cert. version     : 3
      serial number     : 66:F8:5A:E6:B4:EF:6E:49:D3:36:95:63:C9:C3:99:13:E4:71:93:F6
      issuer name       : CN=Batch 1001317, O=Silicon Labs Inc., C=USsubject name      : C=US, O=Silicon Labs Inc., CN=EUI:14B457FFFE0F77CE
DMS:086AEC3C645836BFB04D312F S:SE0 ID:MCU
      issued  on        : 2019-11-19 15:10:33
      expires on        : 2119-11-19 15:10:33
      signed using      : ECDSA with SHA256
      EC key size       : 256 bits
      basic constraints : CA=false
      key usage         : Digital Signature, Non Repudiation, Key Encipherment
      ext key usage     : TLS Web Client Authentication
  + Verify the certificate chain with root certificate... SL_STATUS_OK (cycles: 9703861 time: 255 ms)
```

> **Note**: The longer processing time (255 ms) is due to the certificate chain printout.

Steps 4 and 5 (signature of a challenge) in the Remote Authentication Process

```
. Remote authentication:
  + Create a 16 bytes challenge (random number) in remote device for signing... SL_STATUS_OK (cycles: 3700 time: 97 us)
  + Sign challenge with private device key in Secure Vault High device... SL_STATUS_OK (cycles: 221983 time: 5841 us)
  + Get public device key in Secure Vault High device... SL_STATUS_OK (cycles: 199788 time: 5257 us)
  + Verify signature with public device key in Secure Vault High device... SL_STATUS_OK (cycles: 229054 time: 6027 us)
  + Verify signature with public device key in remote device... SL_STATUS_OK (cycles: 230442 time: 6064 us)

  . SE manager deinitialization... SL_STATUS_OK (cycles: 6 time: 0 us)
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

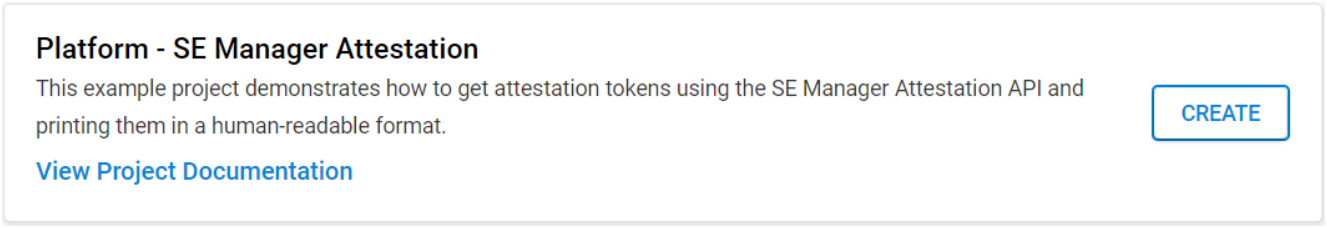
Entity Attestation Token (EAT)

These examples demonstrate how to retrieve the EAT tokens from the HSE-SVH device.

SE Manager - Attestation Platform Example

The SE Manager Attestation platform example uses APIs in SE Manager to retrieve the PSA attestation token and security configuration token from the HSE.

Click the `View Project Documentation` link to open the `readme.html` file.



Press `SPACE` to cycle the challenge size for the PSA attestation token. Press `ENTER` to make a selection and run the program.

```
SE Manager Attestation Example - Core running at 38000 kHz.
Initializing SE Manager...
SL_STATUS_OK (cycles: 10 time: 0 us)

Select nonce size for the IAT token (32, 48 or 64 bytes).
Press SPACE to cycle through the options.
Press ENTER to make a selection.
   Current nonce size:  32
   Selected nonce size: 32
Calling sl_se_attestation_get_psa_iat_token...

SL_STATUS_OK (cycles: 661072 time: 17396 us)
```

PSA Attestation Token (Entity Attestation Token (EAT) and Entity Attestation Token (EAT))

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
PSA IAT token
=============
------------------------------------------------------------------
Raw token:
d28443a10126a058e4a83a000124ff58204ca14d0bc8601cad2e511de1964e93
9338b6fc20f8231aa178ca79519b0ffae73a000124f7715053415f494f545f50
524f46494c455f313a00012500490614b457fffe0f77ce3a000124f8013a0001
24f91920003a000124fa5820011c000106000000001000000f2030f0000000000
0000000000000000000000003a000124fb58204922b7bbd31c0c81c9b0485ccf
b5396ec24ffa877ece441e11c947b791218cf83a000124fd81a3016450526f54
046830303031303230390258206d39caedba129297062b820ba6d85b3e432c44
3c8a8a31d3c6232be6906d38dc584030f9d61523204793965fc9eb2be788db9d
2b02692d877673c86ebffbfb6769984515d2f1a287a92d2c134c1024f20f018d
be952a2ccae7ed2980a9f242d02c9c

COSE_Sign1 structure:
d2    ; tag(18)
  84    ; array(4)
    43    ; byte_str(3)
      a10126
    a0    ; map(0)
    58    ; byte_str(228)
      a83a000124ff58204ca14d0bc8601cad2e511de1964e939338b6fc20f8231aa1
      78ca79519b0ffae73a000124f7715053415f494f545f50524f46494c455f313a
      00012500490614b457fffe0f77ce3a000124f8013a000124f91920003a000124
      fa5820011c000106000000001000000f2030f0000000000000000000000000000
      0000003a000124fb58204922b7bbd31c0c81c9b0485ccfb5396ec24ffa877ece
      441e11c947b791218cf83a000124fd81a3016450526f54046830303031303230
      390258206d39caedba129297062b820ba6d85b3e432c443c8a8a31d3c6232be6
      906d38dc
    58    ; byte_str(64)
      30f9d61523204793965fc9eb2be788db9d2b02692d877673c86ebffbfb676998
      4515d2f1a287a92d2c134c1024f20f018dbe952a2ccae7ed2980a9f242d02c9c
------------------------------------------------------------------
Token claims:
a8    ; map(8)
  3a    ; int(-75008)
  58    ; byte_str(32)
    4ca14d0bc8601cad2e511de1964e939338b6fc20f8231aa178ca79519b0ffae7
  3a    ; int(-75000)
  71    ; text_str(17)
    "PSA_IOT_PROFILE_1"
  3a    ; int(-75009)
  49    ; byte_str(9)
    0614b457fffe0f77ce
  3a    ; int(-75001)
  01    ; int(1)
  3a    ; int(-75002)
  19    ; int(8192)
  3a    ; int(-75003)
  58    ; byte_str(32)
    011c000106000000001000000f2030f000000000000000000000000000000000000
  3a    ; int(-75004)
  58    ; byte_str(32)
    4922b7bbd31c0c81c9b0485ccfb5396ec24ffa877ece441e11c947b791218cf8
  3a    ; int(-75006)
  81    ; array(1)
    a3    ; map(3)
      01    ; int(1)
      64    ; text_str(4)
        "PRoT"
      04    ; int(4)
      68    ; text_str(8)
        "00010209"
      02    ; int(2)
      58    ; byte_str(32)
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
List of claims printed with human-friendly names:
  ARM PSA Nonce, Claim ID: -75008
  58; byte_str(32)
    4ca14d0bc8601cad2e511de1964e939338b6fc20f8231aa178ca79519b0ffae7

  ARM PSA Profile ID, Claim ID: -75000
  71; text_str(17)"PSA_IOT_PROFILE_1"

  ARM PSA / IETF EAT UEID, Claim ID: -75009
  49; byte_str(9)
    0614b457fffe0f77ce

  ARM PSA Partition ID, Claim ID: -75001
  01    ; int(1)

  ARM PSA Lifecycle, Claim ID: -75002
  19; int(8192)

  ARM PSA Implementation ID, Claim ID: -75003
  58; byte_str(32)
    011c000106000000010000000f2030f0000000000000000000000000000000000000

  ARM PSA Boot seed, Claim ID: -75004
  58; byte_str(32)
    4922b7bbd31c0c81c9b0485ccfb5396ec24ffa877ece441e11c947b791218cf8

  ARM PSA Software components, Claim ID: -75006
  81; array(1)
    a3    ; map(3)
      01    ; int(1)64; text_str(4)"PRoT"
      04    ; int(4)68; text_str(8)"00010209"
      02    ; int(2)58; byte_str(32)
        6d39caedba129297062b820ba6d85b3e432c443c8a8a31d3c6232be6906d38dc
```

Security Configuration Token (Entity Attestation Token (EAT))

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
------------------------------------------------------------------

Calling sl_se_attestation_get_config_token...
SL_STATUS_OK (cycles: 541281 time: 14244 us)

Config token
============
------------------------------------------------------------------
Raw token:
d28443a10126a0590133a83a000124ff5820c3e3664dcc47711bf81734bc95f0
87d81dd841d73fc805fc9237c7b3dfa25c503a000124f76853494c4142535f31
3a00012500490614b457fffe0f77ce3a000128df582400000001000000000000
0000000000000000002000010209ffffffff0000002500000003a000128e058
180000000010044440040104141122447742204420a0600053a000128e15840
c4af4ac69aab9512db50f7a26ae5b4801183d85417e729a56da974f4e08a562c
de6019dea9411332dc1a743372d170b436238a34597c410ea177024de20fc819
3a000128e25840b1bc6f6fa56640ed522b2ee0f5b3cf7e5d48f60be8148f0dc0
8440f0a4e1dca47c04119ed6a1be31b7707e5f9d001a659a051003e95e1b936f
05c37ea793ad633a000128e3501504444004010414112244771422044425840b7
47d98be9cef8a91af0292a479a3fa499527018b97ac1188ddefb0fa6fcb9b3d1
d4159240a8663c8803a2ef7cebdf7644fa3394cf1057d612e1b3977d9de92d

COSE_Sign1 structure:
d2    ; tag(18)
  84    ; array(4)
    43    ; byte_str(3)
      a10126
    a0    ; map(0)
    59    ; byte_str(307)
      a83a000124ff5820c3e3664dcc47711bf81734bc95f087d81dd841d73fc805fc
      9237c7b3dfa25c503a000124f76853494c4142535f313a00012500490614b457
      fffe0f77ce3a000128df582400000001000000000000000000000000000000020
      00010209ffffffff0000002500000003a000128e0581800000001004444004
      0104141122447742204420a0600053a000128e15840c4af4ac69aab9512db50
      f7a26ae5b4801183d85417e729a56da974f4e08a562cde6019dea9411332dc1a
      743372d170b436238a34597c410ea177024de20fc8193a000128e25840b1bc6f
      6fa56640ed522b2ee0f5b3cf7e5d48f60be8148f0dc08440f0a4e1dca47c0411
      9ed6a1be31b7707e5f9d001a659a051003e95e1b936f05c37ea793ad633a0001
      28e3501504444004010414112244771422044 2
    58    ; byte_str(64)
      b747d98be9cef8a91af0292a479a3fa499527018b97ac1188ddefb0fa6fcb9b3
      d1d4159240a8663c8803a2ef7cebdf7644fa3394cf1057d612e1b3977d9de92d
------------------------------------------------------------------
Token claims:
a8    ; map(8)
  3a    ; int(-75008)
  58    ; byte_str(32)
    c3e3664dcc47711bf81734bc95f087d81dd841d73fc805fc9237c7b3dfa25c50
  3a    ; int(-75000)
  68    ; text_str(8)
    "SILABS_1"
  3a    ; int(-75009)
  49    ; byte_str(9)
    0614b457fffe0f77ce
  3a    ; int(-76000)
  58    ; byte_str(36)
    000000010000000000000000000000000000000020000102 9ffffffff0000002500000000
  3a    ; int(-76001)
  58    ; byte_str(24)
    000000001004444004010414112244772422044 0a060005
  3a    ; int(-76002)
  58    ; byte_str(64)
    c4af4ac69aab9512db50f7a26ae5b4801183d85417e729a56da974f4e08a562c
    de6019dea9411332dc1a743372d170b436238a34597c410ea177024de20fc819
  3a    ; int(-76003)
  58    ; byte_str(64)
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
7c04119ed6a1be31b7707e5f9d001a659a051003e95e1b936f05c37ea793ad63
  3a    ; int(-76004) 50; byte_str(16) 150444400401041411224477142204 42

List of claims printed with human-friendly names:
  ARM PSA Nonce, Claim ID: -75008
  58; byte_str(32)
    c3e3664dcc47711bf81734bc95f087d81dd841d73fc805fc9237c7b3dfa25c50

  ARM PSA Profile ID, Claim ID: -75000
  68; text_str(8) "SILABS_1"

  ARM PSA / IETF EAT UEID, Claim ID: -75009
  49; byte_str(9)
    0614b457fffe0f77ce

  SE Status, Claim ID: -76000
  58; byte_str(36)
    000000010000000000000000000000000000000020000102 09ffffffff0000025000000000

  OTP Configuration, Claim ID: -76001
  58; byte_str(24)
    000000001004444004010414112244772422044 20a060005

  OTP MCU Boot key, Claim ID: -76002
  58; byte_str(64)
    c4af4ac69aab9512db50f7a26ae5b4801183d85417e729a56da974f4e08a562c
    de6019dea9411332dc1a743372d170b436238a34597c410ea177024de20fc819

  OTP MCU Auth key, Claim ID: -76003
  58; byte_str(64)
    b1bc6f6fa56640ed522b2ee0f5b3cf7e5d48f60be8148f0dc08440f0a4e1dca4
    7c04119ed6a1be31b7707e5f9d001a659a051003e95e1b936f05c37ea793ad63

  Current applied tamper settings, Claim ID: -76004
  50; byte_str(16) 150444400401041411224477142204 42
-------------------------------------------------------------------


Exiting...
SL_STATUS_OK (cycles: 8 time: 0 us)
```

> **Note**: The reserved tamper source in ID 76004 returns a value of 0 or 5.

Simplicity Commander

Run the `security attestation` command to retrieve and validate the security configuration token (Entity Attestation Token (EAT)) from the HSE.

```
commander security attestation --serialno 440030580
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
Certificate chain successfully validated up to Silicon Labs device root certificate.

-75008 ARM PSA nonce                : 05a88aeef627dd663058e3d758fe9a827942da0793da72af81c79a4f60fa9824
-75000 ARM PSA Profile ID           : SILABS_1
-75009 ARM PSA/IETF EAT UEID        : 0614b457fffe0f77ce
-76000 SE status                    : 00000001000000000000000000000003a90000002000010209ffffffff0000002500000000
-76001 OTP configuration            : 00000000100444400401041411224477242204420a060005
-76002 MCU sign key                 : c4af4ac69aab9512db50f7a26ae5b4801183d85417e729a56da974f4e08a562c
                                      de6019dea9411332dc1a743372d170b436238a34597c410ea177024de20fc819
-76003 MCU command key              : b1bc6f6fa56640ed522b2ee0f5b3cf7e5d48f60be8148f0dc08440f0a4e1dca4
                                      7c04119ed6a1be31b7707e5f9d001a659a051003e95e1b936f05c37ea793ad63
-76004 Current applied tamper settings : 150444400401041411224477142204 42

Successfully validated signature of attestation token.
DONE
```

> **Note**: The reserved tamper source in ID 76004 returns a value of 0 or 5.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Secure Key Storage

Secure Key Storage

Note: This section replaces *AN1271: Secure Key Storage*. Further updates to this application note will be provided here.

Secure Key Storage is a feature in High devices that allows for the protection of cryptographic keys by key wrapping. User keys are encrypted by the device's root key for non-volatile storage for later usage. This prevents the need for a key to be stored in plaintext format on the device, preventing attackers from gaining access to the keys through traditional flash-extraction or application attacks, and allowing for a potentially unlimited number of keys to be securely stored in any available storage.

Series 2 devices can use TrustZone to implement Secure Key Storage, so this feature is now also available on Mid devices.

This document describes the operation and usage of this feature, and provides comparisons with other key storage methods.

Key Points

- Keys are encrypted or 'wrapped' with a root key
- root key is not stored on the device, instead it is generated on each reset
- Wrapped keys are confidential to the , and can be stored in non-volatile memory safely
- Wrapped keys can be cached into for usage at a later time
- TrustZone Secure Key Storage

Copyright © 2025 Silicon Laboratories. All rights reserved.

Series 2 Device Security Features

Series 2 Device Security Features

Protecting IoT devices against security threats is central to a quality product. Silicon Labs offers several security options to help developers build secure devices, secure application software, and secure paths of communication to manage those devices. Silicon Labs' security offerings were significantly enhanced by the introduction of the Series 2 products that included a Secure Engine. The Secure Engine is a tamper-resistant component used to securely store sensitive data and keys and to execute cryptographic functions and secure services.

On Series 1 devices, the security features are implemented by the TRNG (if available) and CRYPTO peripherals.

On Series 2 devices, the security features are implemented by the Secure Engine and CRYPTOACC (if available). The Secure Engine may be hardware-based, or virtual (software-based). Throughout this document, the following abbreviations are used:

- HSE - Hardware Secure Engine
- VSE - Virtual Secure Engine
- SE - Secure Engine (either HSE or VSE)

Additional security features are provided by Secure Vault. Three levels of Secure Vault feature support are available, depending on the part and SE implementation, as reflected in the following table:

Level (1)	SE Support	Part (2)
Secure Vault High (SVH)	HSE only (HSE-SVH)	Refer to IoT Endpoint Security Fundamentals for details on supporting devices.
Secure Vault Mid (SVM)	HSE (HSE-SVM)	"
"	VSE (VSE-SVM)	"
Secure Vault Base (SVB)	N/A	"

Notes:

1. The features of different Secure Vault levels can be found in https://www.silabs.com/security.
2. IoT Endpoint Security Fundamentals.

Secure Vault Mid consists of two core security functions:

- Secure Boot: Process where the initial boot phase is executed from an immutable memory (such as ROM) and where code is authenticated before being authorized for execution.
- Secure Debug access control: The ability to lock access to the debug ports for operational security, and to securely unlock them when access is required by an authorized entity.

Secure Vault High offers additional security options:

- Secure Key Storage: Protects cryptographic keys by "wrapping" or encrypting the keys using a root key known only to the HSE-SVH.
- Anti-Tamper protection: A configurable module to protect the device against tamper attacks.
- Device authentication: Functionality that uses a secure device identity certificate along with digital signatures to verify the source or target of device communications.

A Secure Engine Manager and other tools allow users to configure and control their devices both in-house during testing and manufacturing, and after the device is in the field.

User Assistance

Copyright © 2025 Silicon Laboratories. All rights reserved.

In support of these products Silicon Labs offers whitepapers, webinars, and documentation. The following table summarizes the key security documents:

Document	Summary	Applicability
Series 2 Secure Debug	How to lock and unlock Series 2 debug access, including background information about the SE	Secure Vault Mid and High
Series 2 Secure Boot with RTSL	Describes the secure boot process on Series 2 devices using SE	Secure Vault Mid and High
AN1222: Production Programming of Series 2 Devices	How to program, provision, and configure security information using SE during device production	Secure Vault Mid and High
Anti-Tamper Protection Configuration and Use	How to program, provision, and configure the anti-tamper module	Secure Vault High
Authenticating Silicon Labs Devices using Device Certificates	How to authenticate a device using secure device certificates and signatures, at any time during the life of the product	Secure Vault High
Secure Key Storage (this document)	How to securely 'wrap' keys so they can be stored in non-volatile storage.	Secure Vault High

Key Reference

Public/Private keypairs along with other keys are used throughout Silicon Labs security implementations. Because terminology can sometimes be confusing, the following table lists the key names, their applicability, and the documentation where they are used.

Key Name	Customer Programmed	Purpose
Public Sign key (Sign Key Public)	Yes	Secure Boot binary authentication and/or OTA upgrade payload authentication
Public Command key (Command Key Public)	Yes	Secure Debug Unlock or Disable Tamper command authentication
OTA Decryption key (GBL Decryption key) aka AES-128 Key	Yes	Decrypting GBL payloads used for firmware upgrades
Attestation key aka Private Device Key	No	Device authentication for secure identity

SE Firmware

Silicon Labs strongly recommends installing the latest SE firmware on Series 2 devices to support the required security features. Refer to AN1222 for the procedure to upgrade the SE firmware and IoT Endpoint Security Fundamentals for the latest SE Firmware shipped with Series 2 devices and modules.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Introduction

Introduction

The HSE isolates cryptographic functions and data from the host Cortex-M33 core. It is used to accelerate cryptographic operations as well as to provide a method to securely store keys. This application note will cover the Secure Key Storage feature of the HSE-SVH devices.

The HSE contains one-time programmable memory (OTP) key storage slots for three specific keys:

1. The Public Sign Key, used for Secure Boot and Secure Upgrades
2. The Public Command Key, used for Secure Debug unlock and tamper disable
3. The Symmetric OTA Decryption Key, used for Over-The-Air updates

These keys are one-time programmable, and, after programming, are persistent for the lifetime of the device.

HSE-SVH devices also contain four volatile storage slots for any other user keys. These slots are not persistent through a reset. In the case where a key needs persistent storage, the key must be stored outside of the HSE in non-volatile storage. After a device reset, the key can be loaded into the HSE volatile key storage for usage by index, or used in-place (passed to the HSE on every requested operation). Without any secure key storage mechanism, the user key stored in non-volatile storage is opened to storage-extraction attacks (such as gaining access to and downloading device flash), as well as application-level attacks (i.e., taking control of the user application or privileges in a manner that allows access to the keys).

With Secure Key Storage, a user can only access a key from the HSE in a 'wrapped' format. In this format, the key is encrypted by a device-unique root key, only available to the HSE. This allows a user to store a key confidentially in non-volatile storage to provide key persistence. Using Secure Key Storage, the plaintext key is never stored in non-volatile memory, preventing storage-extraction attacks from obtaining the key. After a device reset, the wrapped key can be loaded into the HSE for usage without ever exposing the plaintext key to the application, which also prevents application-level attacks from exposing the key.

SVM devices can only support Secure Key Storage through the use of TrustZone. GSDK v4.2.2 is the first version to support TrustZone software development on Series 2 devices.

Silicon Labs provides Custom Part Manufacturing Service (CPMS) to inject custom secret keys on the chips during manufacturing. For more information about CPMS, see the Custom Part Manufacturing Service User's Guide.

Copyright © 2025 Silicon Laboratories. All rights reserved.

HSE Secure Key Storage

HSE Secure Key Storage

The following sections demonstrate three methods for key storage: ARM® TrustZone®, plaintext, and Secure Key Storage.

> **Note**: In the following examples, AES key usage is demonstrated. However, any other key types supported by the device can also be used for key storage.

Key Generation and Usage

In HSE-SVH devices, cryptographic functions are performed by the HSE. In order to perform these functions, the HSE must have access to any user keys needed. Keys can be generated and used by the HSE in multiple ways:

1. External storage, in-place usage:
    1. A user generates a plaintext key and stores it in device memory.
    2. The user provides a key descriptor to the HSE that points to this key for a specific cryptographic operation.
    3. The HSE performs the cryptographic operation using this key, but does not store it in any HSE volatile storage slot.
2. External storage with HSE import:
    1. A user generates a plaintext key and stores it in device memory.
    2. The user provides a key descriptor to the HSE that points to this key, as well as a slot number to store the key.
    3. The HSE imports this key into a volatile key storage slot or can optionally save it in wrapped form in device memory.
    4. The user requests that the HSE performs a cryptographic function by providing the index of the storage slot or a pointer to the wrapped key in device memory.
3. Internal HSE key generation:
    1. The user commands the HSE to generate a new key within one of the HSE's volatile key slots or can optionally save it in wrapped form in device memory.
    2. The user requests that the HSE performs a cryptographic function by providing the index of the storage slot or a pointer to the wrapped key in device memory.

Notes:

- In each case, to provide persistent storage for the key, the key must be stored in non-volatile memory.
- Plain Key Storage and Secure Key Storage provide details on key generation and usage with HSE-SVH device.

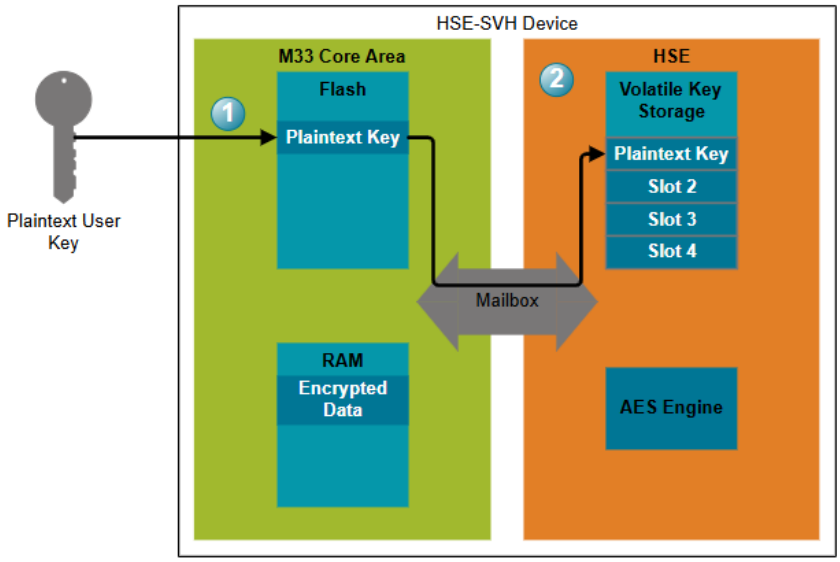
Plaintext Key Storage

Plaintext Key Import

The simplest manner to store a key is to save it in plaintext form. The steps to store and use a key stored in plaintext form are as follows:

1. A user key is generated and imported into device memory. For persistent storage, this must be non-volatile storage, such as device flash.
2. After a device reset, the HSE volatile key storage will be empty. The plaintext key is imported (method 2) into a slot for usage. Alternatively, the key could be used in place (method 1) from non-volatile storage on a per-operation basis.

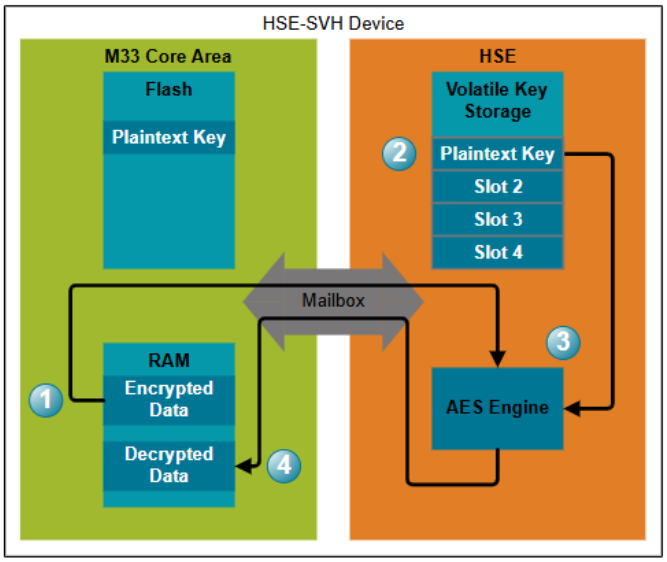
Copyright © 2025 Silicon Laboratories. All rights reserved.



Plaintext Key Usage

In order to use the key for a cryptographic operation, the following procedure is used.

1. The user passes data to be processed (in this specific example, AES encrypted data) to the HSE.
2. The user requests that a cryptographic operation be performed on this data using one of the keys stored in the HSE volatile key storage slots (method 2). Alternatively, the key can be passed to the HSE directly for a singular cryptographic operation (method 1).
3. The HSE performs the cryptographic operation.
4. The output of the cryptographic operation is passed back to the user for processing.



This method exposes the keys to two major vulnerabilities:

1. Access to device storage gives access to the keys. In this case, an attack that gains access to the flash contents will expose the user key.
2. Since the application has access to the keys, compromising the application or device privileges can compromise the keys. Such an attack might not directly access device memory, but take control of the application in a way that causes the application to expose the key to an attacker.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Secure Key Storage

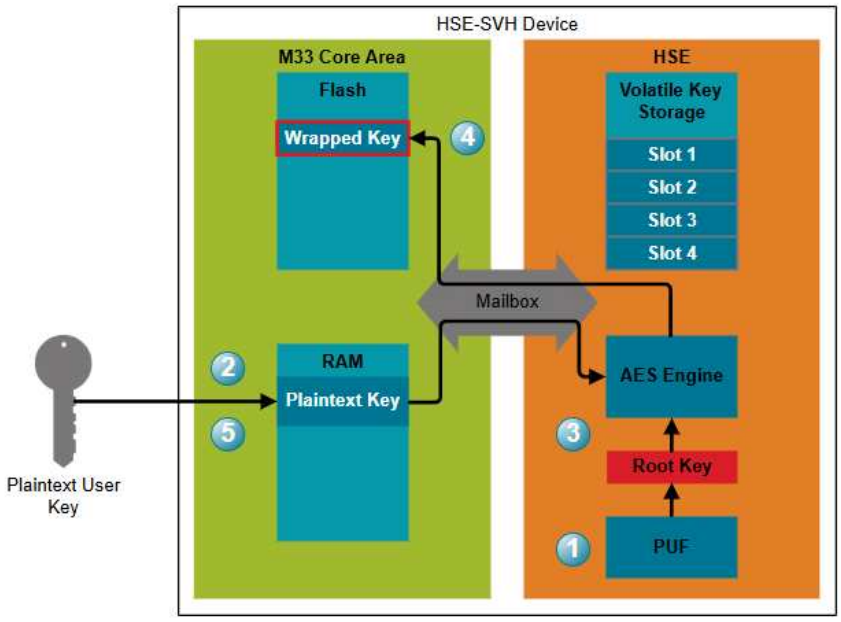
With Secure Key Storage, the user key, using the HSE, can be accessed in an encrypted, or 'wrapped' form. Only the HSE has access to the HSE root key used to decrypt, or 'unwrap', the wrapped key. This HSE root key is not stored on the device during power-down, but rather reconstructed after each reset. Key wrapping allows a user to securely store a key in non-volatile memory, limiting the number of keys that can be stored only by the amount of storage the user has available.

> **Note**: The reconstructed root key after each reset is IDENTICAL and UNIQUE on each HSE-SVH device.

Wrap an External Key

To wrap an externally-generated key:

1. After power-on, the device's unique root key is reconstructed with output from the Physically Unclonable Function (PUF).
2. A user key is generated and imported into device memory. In this example, the key is imported into RAM for easy deletion, and the added security that, if device power is removed, the key will be lost.
3. The user key is passed to the HSE, where it is encrypted with the HSE's root key.
4. The wrapped key is passed back to the user application for storage in non-volatile memory (in this case, device flash).
5. The plaintext key can now be deleted from the device. From this point forward, only the HSE will have access to the plaintext key.

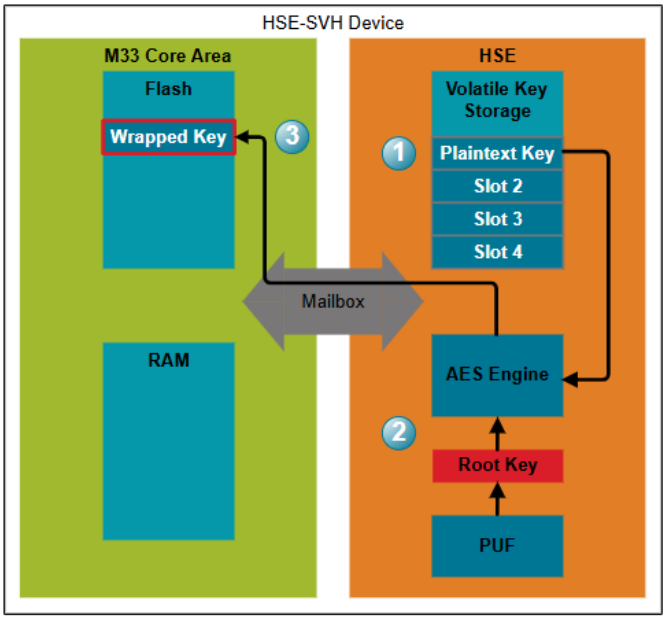


Generate an Internal Wrapped Key

Instead of importing an external key, the HSE can generate a new key directly into one of its volatile key storage slots. This key can then be exported in wrapped form for secure persistent storage.

1. The user requests that the HSE generates a new key into one of its storage slots using the True Random Number Generator (TRNG).
2. The key is encrypted with the HSE's root key.
3. The wrapped key is passed back to the user application for non-volatile storage (flash, in this case).

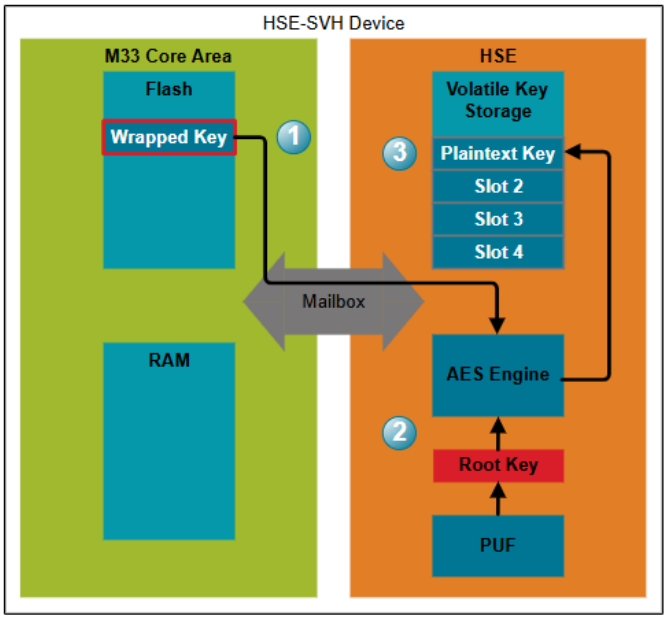
Copyright © 2025 Silicon Laboratories. All rights reserved.



Wrapped Key Import

In order to import a wrapped key into the HSE for usage:

1. The wrapped key is passed to the HSE.
2. The wrapped key is decrypted ("unwrapped") with the HSE's root key.
3. The plaintext key is stored in a volatile key storage slot.



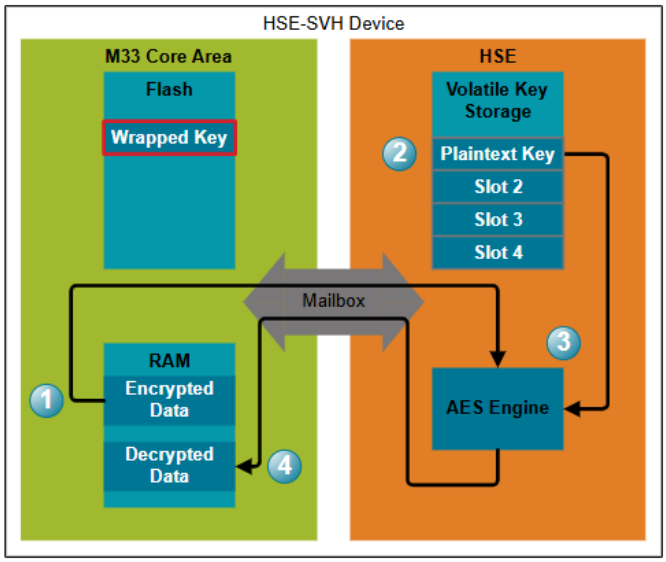
Wrapped Key Usage

In order to use the key for a cryptographic operation, the same steps are followed as when using a plaintext key that has been imported into the HSE:

1. The user passes data to be processed (in this specific example, AES encrypted data) to the HSE.

Copyright © 2025 Silicon Laboratories. All rights reserved.

2. The user requests that a cryptographic operation be performed on this data using one of the keys stored in the HSE volatile key storage slots. Alternatively, the wrapped key can be passed to the HSE directly for a singular cryptographic operation. In this case, the key will be unwrapped before being used, but will not be stored for future operations.
3. The HSE performs the cryptographic operation.
4. The output of the cryptographic operation is passed back to the user for processing.



Secure Key Storage Advantages

Secure Key Storage confers the following benefits over other key storage methods:

1. Access to device memory does not expose user keys.
2. Compromising the user application does not expose user keys, since the user application itself does not have access to the plaintext keys.
3. The number of user keys that can be securely stored is only limited by the amount of storage available to the user, including external storage.

Operation Details

Root Key Generation

Secure Key Storage depends on the HSE to encrypt / decrypt (wrap / unwrap) user keys with its own symmetric root key. The symmetric key used for this wrapping and unwrapping must be highly secure as it can expose all other key material in the system. The HSE key Management system uses a Physically Unclonable Function (PUF) to generate a persistent device-unique seed on power up to dynamically reconstruct this critical root key. The key is only visible to the AES encryption engine, and it is not retained when the device loses power.

Access a Wrapped Key

By default, a key in an HSE storage slot can be exported to the application as a plaintext key. To prevent this, the user can use the key descriptor to set a user key to non-exportable. This option prevents any request to export the wrapped key in plaintext from HSE, so the user application can only access the key encrypted by the HSE's root key. The HSE also tags the key with information to identify the wrapped key. Since only the HSE can access the root key to unwrap the user key, the plaintext key is non-accessible to the user application.

> **Note**: Wrapped keys are slightly larger than the equivalent plaintext key, as some additional metadata is required to identify the wrapped key to the HSE.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Wrapped Key Storage and Usage

Once a key has been wrapped, it can be safely stored anywhere - device flash, RAM, external storage, etc. The number of keys that can be securely stored is only limited by the available storage space. A wrapped key can later be imported into a HSE volatile storage slot for usage, or used in-place. Once the key is wrapped and stored, the plaintext key available to the application can be deleted. From here, only the HSE will have the ability to unwrap and use the key.

With access to the wrapped key, the HSE can use this key in one of two ways:

1. A user can request that a cryptographic operation be performed using the key stored in memory. In this case, the HSE will import the key, unwrap it, and then perform the cryptographic operation. The key will not be stored within the HSE.
2. A user can import the wrapped key into a HSE volatile storage slot. In this case, the key is unwrapped by the HSE and stored in plaintext in a volatile slot. The user can then later request that a cryptographic function be performed by the HSE by referencing the volatile slot index. This provides a performance increase over using wrapped keys in place, as the HSE does not need to import and unwrap the key on each requested operation.

Password Protection

When defining a key descriptor for a new key, or when importing an existing key into HSE, the user can choose to require a password to allow use of the key. The password field in the key descriptor structure is eight bytes in length. If unspecified, the key will use the default password of all zeros.

After importing a key with a password, failing to provide the correct password when performing a cryptographic operation will result in HSE returning an invalid credentials error, and no operation will be performed.

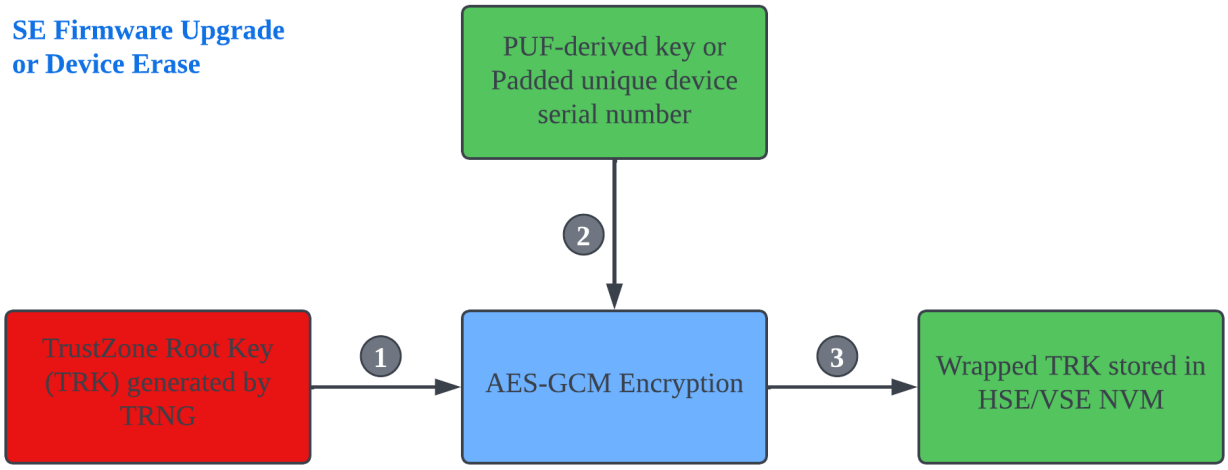
Copyright © 2025 Silicon Laboratories. All rights reserved.

TrustZone Secure Key Storage

In Series 2 devices, key management can be handled by a feature called TrustZone. TrustZone divides the device memory map into a Secure Processing Environment (SPE) and a Non-secure Processing Environment (NSPE). User code is executed from the NSPE, which cannot access any part of the SPE. The SPE is used to store cryptographic keys securely and to control other Secure operations.

The following sections describe using TrustZone on Series 2 devices for Secure Key Storage. Refer to Series 2 TrustZone for details about TrustZone implementation on Series 2 devices.

TrustZone Root Key Generation (HSE and VSE)

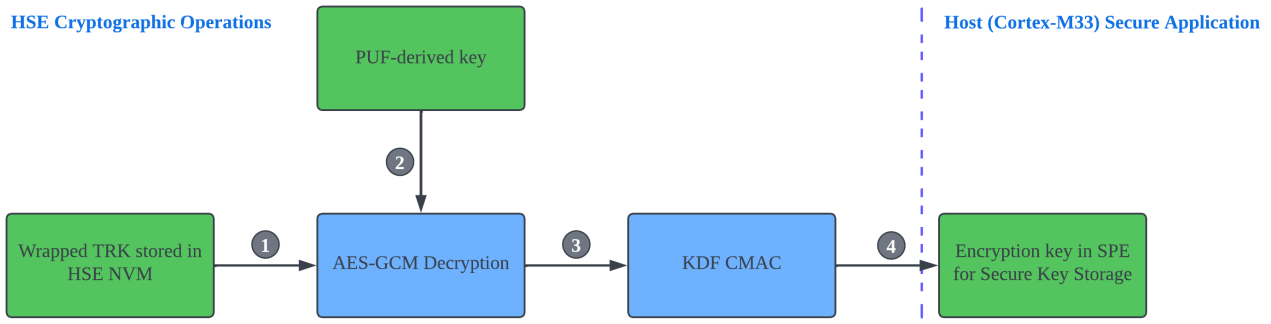


1. The TrustZone Root Key (TRK) is generated by the True Random Number Generator (TRNG) in Series 2 devices.
2. The PUF-derived key (HSE and xG27 VSE devices) or padded unique device serial number (xG22 VSE devices) is used to wrap (AES-GCM) the TRK.
3. The wrapped TRK is stored in the SE Non-volatile memory (NVM), and the TRK in RAM is deleted.
    - The wrapped TRK already existed if the shipped Series 2 device with SE firmware version supports this key.
    - The wrapped TRK will be generated when upgrading from a SE firmware version that did not support this key to the one that does.
    - The wrapped TRK will be renewed after performing a Device Erase.

> **Note**: The Physically Unclonable Function (PUF) is not retained when the device loses power, so the TRK wrapped by the PUF-derived key is not vulnerable to a storage-extraction attack.

TrustZone Root Key Usage (HSE)

Copyright © 2025 Silicon Laboratories. All rights reserved.

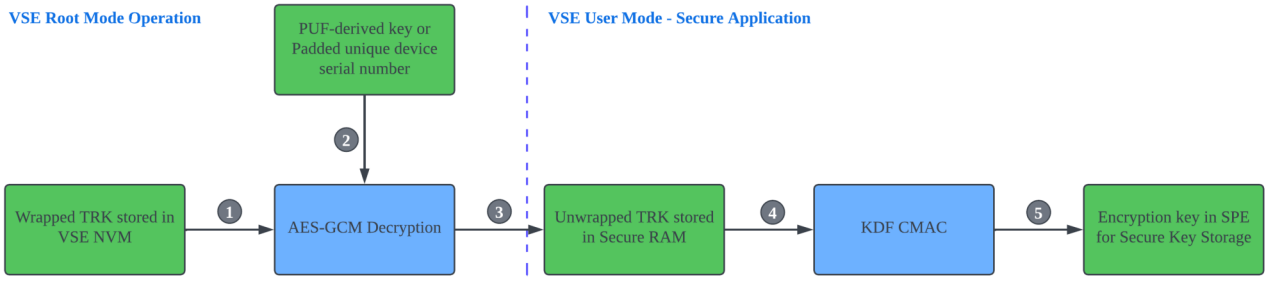


1. The Secure application in the host uses a non-exportable built-in key to access the wrapped TRK in HSE NVM for cryptographic operations.
2. The PUF-derived key is used to decrypt (AES-GCM) the wrapped TRK in HSE NVM.
3. The unwrapped TRK in the HSE is the master key of a Key Derivation Function (KDF).
4. The encryption key in SPE for Secure Key Storage is derived from the KDF CMAC.

Notes:

- All cryptographic operations are performed by the HSE (security co-processor).
- Only the HSE can access the unwrapped TRK for KDF, so this key will not expose the Secure application in the host.

TrustZone Root Key Usage (VSE)



1. The wrapped TRK in VSE NVM is accessed by the VSE Root mode firmware.
2. The PUF-derived key (xG27) or padded unique device serial number (xG22) is used to decrypt (AES-GCM) the wrapped TRK in VSE NVM.
3. Unwrapped TRK is transferred to the shared RAM when switching from VSE Root mode to User mode. The VSE user mode Secure application stores this key to the Secure RAM in SPE and deletes this key in the shared RAM.
4. The unwrapped TRK in the Secure RAM is the master key of a Key Derivation Function (KDF).
5. The encryption key in SPE for Secure Key Storage is derived from the KDF CMAC.

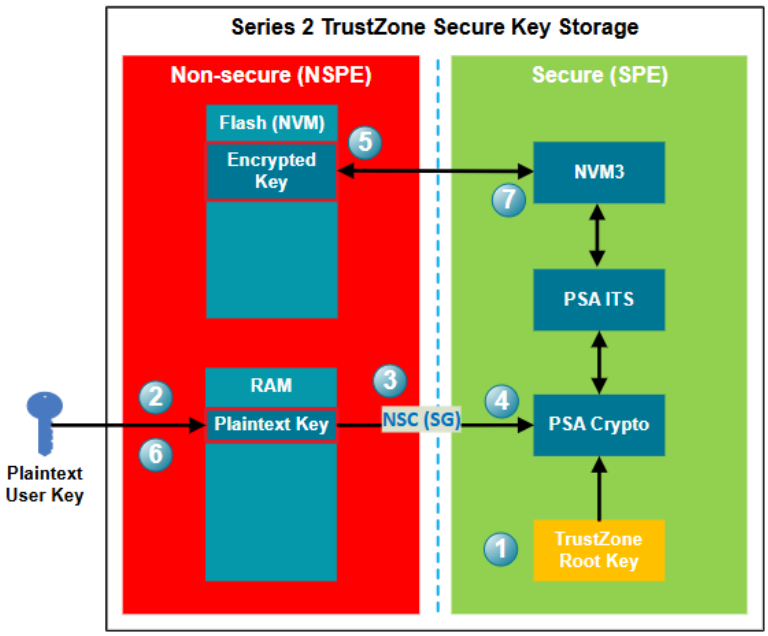
> **Note**: On VSE devices, all cryptographic operations are performed by the Cryptographic Accelerator (CRYPTOACC) peripheral.

For more information about the HSE and VSE, refer to the section *Secure Engine Subsystem* in Series 2 Secure Debug.

TrustZone Secure Key Storage (HSE and VSE)

The TRK allows a user to securely store a key in the Non-secure flash, limiting the number of keys that can be saved only by the amount of Non-secure storage. The following figure describes using the TRK to encrypt a plaintext key and store it

Copyright © 2025 Silicon Laboratories. All rights reserved.



in Non-secure NVM.

1. After power-on, the device's TRK (wrapped in HSE NVM and unwrapped in VSE Secure RAM) is available for the SPE.
2. A user key is generated and imported into the device's Non-secure memory. In this example, the key is imported into Non-secure RAM for easy deletion, and the key is lost if device power is removed.
3. Call the PSA Crypto API (`psa_import_key()` or `psa_generate_key()`) through the Secure Gateway (SG) in Non-secure Callable (NSC) memory to generate a key for crypto operations.
4. The plaintext key is passed in the PSA Crypto API to the SPE, where it is encrypted (AES-GCM) with the encryption key derived (KDF CMAC) from the TRK.
5. The encrypted key is stored to the NVM region in the NSPE through the PSA Internal Trusted Storage (ITS) and NVM3 drivers.
6. The plaintext key can now be deleted from the Non-secure RAM.
7. Only the PSA Crypto API in the SPE can retrieve the encrypted key from NVM in the NSPE and decrypt it for crypto operations in the SPE.

> **Note**: Ignore steps 2 and 6 if the plaintext key is randomly generated by the PSA Crypto.

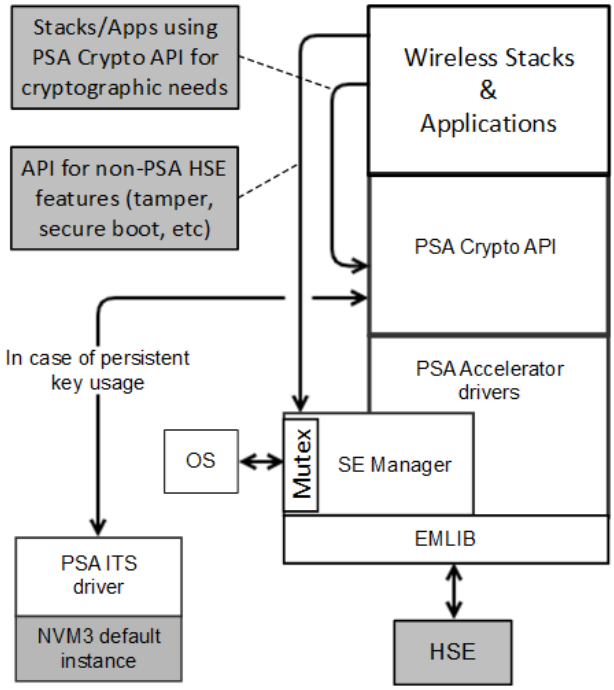
Copyright © 2025 Silicon Laboratories. All rights reserved.

Secure Key Storage Implementations

Secure Key Storage Implementations

Users can use Secure Engine Manager (SE Manager) or PSA Crypto in the following figure to access the secure key storage on HSE-SVH devices. SE Manager APIs for secure key storage and crypto are usually not considered external APIs. PSA Crypto API abstracts the entropy sources, crypto primitives, and even advanced security features like secure key storage from the calling functions.

Silicon Labs recommends using PSA Crypto API for secure key storage and cryptography whenever possible. It makes the solution more portable and hardware agnostic. In some cases, however, setting up tamper and initializing the secure boot can only be implemented by the SE Manager APIs.



Component	Functionality
EMLIB (em_se.c)	Abstracts the mailbox interface: how to construct, send and receive low-level HSE mailbox commands.
SE Manager	On top of EMLIB, it abstracts the HSE command set: translates function calls into mailbox messages. The SE Manager also provides thread synchronization.
PSA Accelerator Drivers	A translation layer to map the PSA Crypto HSE interface and crypto acceleration calls to SE Manager calls.
PSA Crypto API	Platform independent cryptographic hardware acceleration support by implementing standardized APIs.
PSA ITS Driver	The key management functionality in PSA Crypto needs access to non-volatile memory for persistent storage of plaintext or wrapped keys. NVM3 gets wrapped by this translation layer, mapping the PSA ITS (Internal Trusted Storage) interface to NVM3 calls.

For the SE's mailbox interface, see section *Secure Engine Subsystem* in Series 2 Secure Debug.

For more information about NVM3, see https://docs.silabs.com/gecko-platform/latest/driver/api/group-nvm3.

Copyright © 2025 Silicon Laboratories. All rights reserved.

For more information about PSA Crypto, see AN1311: Integrating Crypto Functionality Using PSA Crypto Compared to Mbed TLS.

SE Manager API

The following table lists the SE Manager APIs related to Secure Key Storage operations. The SE Manager API document can be found at https://docs.silabs.com/gecko-platform/latest/service/api/group-sl-se-manager.

SE Manager API	Usage
sl_se_generate_key	Generate a new key and store it either in a volatile HSE storage slot or as a wrapped key.
sl_se_import_key	Import a plaintext key and store it either in a volatile HSE storage slot or as a wrapped key.
sl_se_export_key	Export a volatile or wrapped key back to plaintext if allowed. It will fail for a key that has been flagged as SL_SE_KEY_FLAG_NON_EXPORTABLE.
sl_se_transfer_key	Transfer a volatile or wrapped key to another storage option (volatile HSE storage slot or a wrapped key) if allowed.
sl_se_delete_key	Delete a key from a volatile HSE storage slot.

PSA Crypto API

The following table lists the PSA Crypto APIs related to Secure Key Storage operations. The PSA Crypto API document can be found at https://docs.silabs.com/mbed-tls/latest/.

For more information about PSA Crypto APIs on Secure Key Storage, see AN1311: Integrating Crypto Functionality Using PSA Crypto Compared to Mbed TLS.

PSA Crypto API	Usage
psa_generate_key	Generate a new plaintext or wrapped key and store it either in volatile or non-volatile memory.
psa_import_key	Import a plaintext key and save it in plaintext or wrapped form. It can store either in volatile or non-volatile memory.
psa_export_key	Export a key back to plaintext if allowed. The policy on the key must have the usage flag `PSA_KEY_USAGE_EXPORT` set.
psa_copy_key	Copy key material from one location to another, which may have a different lifetime (e.g., volatile to non-volatile).
psa_destroy_key	Destroy a key from both volatile memory and, if applicable, non-volatile storage.

SE Manager API Versus PSA Crypto API

The following table compares the SE Manager APIs with PSA Crypto APIs on Secure Key Storage.

Item	SE Manager API	PSA Crypto API
Availability	Only on HSE devices	Platform independent
API	Silicon Labs proprietary	Standardized by ARM®
Key Storage	Volatile (RAM) memory only	Volatile (RAM) or non-volatile (flash) memory
Wrapped Key Cache	Can use a volatile HSE storage slot	Not yet implemented
Password Protection	Can define in a key descriptor	Not yet defined in PSA Crypto
Custom ECC Curve	Can define in a key descriptor	Not yet defined in PSA Crypto

PSA Crypto Key Types with TrustZone Secure Key Storage

The following tables describes the storage differences between key storage with and without TrustZone on SVM and SVH devices.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Table: TrustZone Secure Key Storage (SKS) on SVM Devices

Key Type	Storage without TrustZone SKS	Storage with TrustZone SKS
Volatile Plaintext	RAM	Secure RAM (2)
Persistent Plaintext	NVM	Encrypted in NS NVM (2)
Volatile Wrapped	Not supported	Not supported
Persistent Wrapped	Not supported	Not supported

Table: TrustZone Secure Key Storage (SKS) on SVH Devices

Key Type	Storage without TrustZone SKS	Storage with TrustZone SKS
Volatile Plaintext	Plaintext key in RAM	Plaintext key in Secure RAM
Persistent Plaintext	Plaintext key in NVM	Encrypted plaintext key in NS NVM
Volatile Wrapped	Wrapped key in RAM (1)	Wrapped key in Secure RAM
Persistent Wrapped	Wrapped key in NVM (1)	Encrypted wrapped key in NS NVM

Notes:

- The NVM or NS NVM is at the last part of the main flash.
- It is possible to replace the wrapped key solution on the SVH device (1) with TrustZone Secure Key Storage on the SVM device (2), but this is a less secure approach.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Examples

Examples

Simplicity Studio 5 includes the SE Manager and PSA Crypto platform examples for Secure Key Storage. Refer to the corresponding `readme` file for details about each SE Manager and PSA Crypto platform example. This file also includes the procedures to create the project and run the example.

Table: Platform Examples for Secure Key Storage

Category	SE Manager Platform Example	PSA Crypto Platform Example
Key Handling	SE Manager Symmetric Key Handling	PSA Crypto Symmetric Key
	SE Manager Asymmetric Key Handling	PSA Crypto Asymmetric Key
Symmetric Key Usage	SE Manager Block Cipher	PSA Crypto AEAD
		PSA Crypto Cipher
		PSA Crypto KDF
		PSA Crypto MAC
Asymmetric Key Usage	SE Manager Digital Signature (ECDSA and EdDSA)	PSA Crypto DSA
	SE Manager Key Agreement (ECDH)	PSA Crypto ECDH
X.509 Certificate	-	PSA Crypto X.509
TrustZone Secure Key Storage	-	tz_psa_crypto_ecdh_ws

Copyright © 2025 Silicon Laboratories. All rights reserved.

Connect API Reference Guide

Connect

Silicon Labs Connect Stack API

Connect stack API is the primary Application Programming Interface (API) for applications running on Silicon Labs EFR32 Wireless Gecko SoCs to interact with the Silicon Labs Connect wireless stack.

Silicon Labs is developing products designed to meet the demands of customers moving to a connected world of devices in the home, often referred to as the IoT (Internet of Things). At a high level, the IoT goals for Silicon Labs are as follows:

- Connect all devices in the home with best-in-class mesh networking, either with Ember ZigBee PRO or other emerging standards.
- Leverage the company expertise in low-power, constrained devices.
- Enhance established low-power, mixed-signal chips.
- Provide low-cost bridging to existing Ethernet and Wi-Fi devices.
- Enable cloud services and connectivity to smartphones and tablets that promote ease of use and a common user experience for customers.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Connect Stack API Reference

Connect Stack API Reference

The primary API towards the Connect radio stack.

Modules

Connect Stack Version

Connect Data Types

Stack Information

Network Management

Radio Stream

Configuration

Status Codes

Stack Tokens

Event Scheduling

Memory Buffer

Copyright © 2025 Silicon Laboratories. All rights reserved.

Connect Stack Version

Connect Stack Version

Macros to determine the stack version.

Note that the Connect Stack version might not match the version of Flex SDK.

See config.h for source code.

Macros

#define EMBER_MAJOR_VERSION 4
The major version of the release. First digit of A.B.C.D.

#define EMBER_MINOR_VERSION 1
The minor version of the release. Second digit of A.B.C.D.

#define EMBER_PATCH_VERSION 1
The patch version of the release. Third digit of A.B.C.D.

#define EMBER_SPECIAL_VERSION 0
Special version of the release. Fourth digit of A.B.C.D.

#define EMBER_BUILD_NUMBER 0
Build number of the release. Should be stored on 2 bytes.

#define EMBER_FULL_VERSION undefined
Full version number stored on 2 bytes, with each of the four digits stored on 4 bits.

#define EMBER_VERSION_TYPE EMBER_VERSION_TYPE_GA
Version type of the release. EMBER_VERSION_TYPE_GA means generally available.

#define SOFTWARE_VERSION EMBER_FULL_VERSION
Full version number stored on 2 bytes, with each of the four digits stored on 4 bits.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Connect Data Types

Connect Data Types

Definitions of Connect data types used by various Connect API functions.

See ember-types.h for source code.

Modules

EmberNetworkParameters

EmberIncomingMessage

EmberOutgoingMessage

EmberMacAddress

EmberMacFrame

EmberIncomingMacMessage

EmberOutgoingMacMessage

EmberKeyData

sl_connect_radio_priority

EventActions

Event_s

EventQueue_s

EmberEventControl

EmberEventData_S

EmberTaskControl

Enumerations

enum EmberNodeType {

EMBER_UNKNOWN_DEVICE = 0
EMBER_STAR_COORDINATOR = 1
EMBER_STAR_RANGE_EXTENDER = 2
EMBER_STAR_END_DEVICE = 3
EMBER_STAR_SLEEPY_END_DEVICE = 4
EMBER_DIRECT_DEVICE = 5
EMBER_MAC_MODE_DEVICE = 6
EMBER_MAC_MODE_SLEEPY_DEVICE = 7

}

Define the possible types of nodes and the roles that a node might play in a network.

enum EmberNetworkStatus {

EMBER_NO_NETWORK
EMBER_JOINING_NETWORK
EMBER_JOINED_NETWORK
EMBER_RADIO_TEST

Copyright © 2025 Silicon Laboratories. All rights reserved.

}

Defines the possible join states for a node.

enum EmberChildFlags {

EMBER_CHILD_FLAGS_DEVICE_IS_RANGE_EXTENDER_BIT = 0×02
EMBER_CHILD_FLAGS_DEVICE_IS_SLEEPY_BIT = 0×04
EMBER_CHILD_FLAGS_HAVE_PENDING_DATA_BIT = 0×08
EMBER_CHILD_FLAGS_AES_SECURITY_CAPABLE_BIT = 0×10
EMBER_CHILD_FLAG_DEVICE_IS_EXTENDED_BIT = 0×20

}

Child flags.

enum EmberMessageOptions {

EMBER_OPTIONS_NONE = 0×00
EMBER_OPTIONS_SECURITY_ENABLED = 0×01
EMBER_OPTIONS_ACK_REQUESTED = 0×02
EMBER_OPTIONS_HIGH_PRIORITY = 0×04
EMBER_OPTIONS_INDIRECT = 0×08

}

Message options.

enum EmberMacAddressMode {

EMBER_MAC_ADDRESS_MODE_NONE = 0×00
EMBER_MAC_ADDRESS_MODE_SHORT = 0×02
EMBER_MAC_ADDRESS_MODE_LONG = 0×03

}

802.15.4 addressing mode.

enum EmberEventUnits {

EMBER_EVENT_INACTIVE = 0
EMBER_EVENT_MS_TIME
EMBER_EVENT_QS_TIME
EMBER_EVENT_MINUTE_TIME
EMBER_EVENT_ZERO_DELAY

}

Either marks an event as inactive or specifies the units for the event execution time.

enum @2 {

EMBER_OUTGOING_MESSAGES = 0×0001
EMBER_INCOMING_MESSAGES = 0×0002
EMBER_RADIO_IS_ON = 0×0004
EMBER_ASSOCIATING = 0×0008
EMBER_SCANNING = 0×0010

}

Define tasks that prevent the stack from sleeping.

Copyright © 2025 Silicon Laboratories. All rights reserved.

enum EmberCounterType {

EMBER_COUNTER_PHY_IN_PACKETS
EMBER_COUNTER_PHY_OUT_PACKETS
EMBER_COUNTER_MAC_IN_UNICAST
EMBER_COUNTER_MAC_IN_BROADCAST
EMBER_COUNTER_MAC_OUT_UNICAST_NO_ACK
EMBER_COUNTER_MAC_OUT_UNICAST_ACK_SUCCESS
EMBER_COUNTER_MAC_OUT_UNICAST_ACK_FAIL
EMBER_COUNTER_MAC_OUT_UNICAST_CCA_FAIL
EMBER_COUNTER_MAC_OUT_UNICAST_RETRY
EMBER_COUNTER_MAC_OUT_BROADCAST
EMBER_COUNTER_MAC_OUT_BROADCAST_CCA_FAIL
EMBER_COUNTER_MAC_OUT_ENCRYPT_FAIL
EMBER_COUNTER_MAC_DROP_IN_MEMORY
EMBER_COUNTER_MAC_DROP_IN_FRAME_COUNTER
EMBER_COUNTER_MAC_DROP_IN_DECRYPT
EMBER_COUNTER_NWK_OUT_FORWARDING
EMBER_COUNTER_NWK_IN_SUCCESS
EMBER_COUNTER_NWK_DROP_IN_WRONG_SOURCE
EMBER_COUNTER_NWK_DROP_IN_FORWARDING
EMBER_COUNTER_UART_IN_DATA
EMBER_COUNTER_UART_IN_MANAGEMENT
EMBER_COUNTER_UART_IN_FAIL
EMBER_COUNTER_UART_OUT_DATA
EMBER_COUNTER_UART_OUT_MANAGEMENT
EMBER_COUNTER_UART_OUT_FAIL
EMBER_COUNTER_ROUTE_2_HOP_LOOP
EMBER_COUNTER_BUFFER_ALLOCATION_FAIL
EMBER_ASH_V3_ACK_SENT
EMBER_ASH_V3_ACK_RECEIVED
EMBER_ASH_V3_NACK_SENT
EMBER_ASH_V3_NACK_RECEIVED
EMBER_ASH_V3_RESEND
EMBER_ASH_V3_BYTES_SENT
EMBER_ASH_V3_TOTAL_BYTES_RECEIVED
EMBER_ASH_V3_VALID_BYTES_RECEIVED
EMBER_ASH_V3_PAYLOAD_BYTES_SENT
EMBER_COUNTER_TYPE_COUNT

}

Define the event counters that can be requested from the application using emberGetCounter()

enum EmberPhyType {

EMBER_RADIO_CONFIGURATOR
EMBER_STANDARD_PHY_2_4GHZ
EMBER_STANDARD_PHY_915MHZ
EMBER_STANDARD_PHY_863MHZ

}

Define the PHY configuration of connect stack.

enum EmberCalType {

EMBER_CAL_TEMP_VCO = 0×00000001
EMBER_CAL_IRCAL = 0×00010000
EMBER_CAL_ALL = 0×00010001

}

Define the type of calibration requested.

enum EmberTxStreamParameters {

TX_STREAM_PN9
TX_STREAM_CW

}

Radio Stream mode.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Typedefs

typedef uint8_t **EmberEUI64**[EUI64_SIZE]
EUI 64-bit ID (IEEE 802.15.4 long address).

typedef uint16_t **EmberNodeId**
IEEE 802.15.4 node ID. Also known as short address.

typedef uint16_t **EmberPanId**
IEEE 802.15.4 PAN ID.

typedef uint16_t **EmberMessageLength**
Message length in bytes.

typedef uint8_t **EmberTaskId**
An identifier for a task.

typedef struct sl_connect_radio_priority **sl_connect_radio_priority_t**
Struct used to specify priorities for Connect's radio operations in multiprotocol.

typedef struct Event_s **Event**

typedef struct EventQueue_s **EventQueue**
An event queue is currently just a list of events ordered by execution time.

typedef const struct EmberEventData_S **EmberEventData**
Complete events with a control and a handler procedure.

typedef uint16_t **EmberBuffer**
Buffers used by the memory buffer system.

Functions

uint8_t * **emberKeyContents**(EmberKeyData *key)
This macro allows the programmer to gain access to the key data bytes of the EmberKeyData structure.

Macros

#define **EXTENDED_PAN_ID_SIZE** 8
Size of an extended PAN identifier in bytes (8).

#define **EUI64_SIZE** 8
Size of EUI64 (an IEEE address) in bytes (8).

#define **EMBER_ENCRYPTION_KEY_SIZE** 16
Size of an encryption key in bytes (16).

#define **EMBER_NULL_NODE_ID** 0xFFFFu
A distinguished network ID that will never be assigned to any node. Used to indicate the absence of a node ID.

#define **EMBER_BROADCAST_ADDRESS** 0xFFFF
Broadcast address.

#define **EMBER_USE_LONG_ADDRESS** 0xFFFE
Special short address indicating the node should use long addressing as source address.

Copyright © 2025 Silicon Laboratories. All rights reserved.

#define EMBER_COORDINATOR_ADDRESS 0x0000
The coordinator short address.

#define EMBER_CAL_INVALID_VALUE (0xFFFFFFFF)

Enumeration Documentation

EmberNodeType

> EmberNodeType

Define the possible types of nodes and the roles that a node might play in a network.

Enumerator	
EMBER_UNKNOWN_DEVICE	Device is not in network.
EMBER_STAR_COORDINATOR	Extended star mode device: Will relay messages and can act as a parent to range extender and end device nodes. A node can be started as a coordinator using the emberFormNetwork() API.
EMBER_STAR_RANGE_EXTENDER	Extended star mode device: Will relay messages and can act as a parent to end device nodes. Joins to a coordinator. A node can be started as a range extender using the emberJoinNetwork() or emberJoinNetworkExtended() API.
EMBER_STAR_END_DEVICE	Extended star mode device: Communicates only with its parent and will not relay messages. A node can be started as an end device using the emberJoinNetwork() or emberJoinNetworkExtended() API.
EMBER_STAR_SLEEPY_END_DEVICE	Extended star mode device: An end device whose radio is turned off when not communicating to save power. The application must call emberPollForData() to receive messages. A node can be started as a sleepy end device using the emberJoinNetwork() or emberJoinNetworkExtended() API.
EMBER_DIRECT_DEVICE	Direct mode device: A device able to send and receive messages from other devices in range on the same PAN ID, with no star topology restrictions. Such device does not relay messages. A node can be started as a direct device by using the emberJoinCommissioned() API.
EMBER_MAC_MODE_DEVICE	MAC mode device: A device able to send and receive MAC-level messages. A node can be started as a MAC mode device by using the emberJoinCommissioned(), the emberJoinNetworkExtended() or the emberMacFormNetwork() API.
EMBER_MAC_MODE_SLEEPY_DEVICE	MAC mode device: A sleepy device able to send and receive MAC-level messages. The radio on the device is turned off when not communicating. The application must call emberPollForData() to receive messages. A node can be started as a sleepy MAC mode device by using the emberJoinCommissioned() API or the emberJoinNetworkExtended() API.

EmberNetworkStatus

> EmberNetworkStatus

Defines the possible join states for a node.

Enumerator	
EMBER_NO_NETWORK	The node is not associated with a network in any way.
EMBER_JOINING_NETWORK	The node is currently attempting to join a network.

Copyright © 2025 Silicon Laboratories. All rights reserved.

EMBER_JOINED_NETWORK	The node is joined to a network.
EMBER_RADIO_TEST	The node is currently using its radio for test

EmberChildFlags

EmberChildFlags

Child flags.

Enumerator	
EMBER_CHILD_FLAGS_DEVICE_IS_RANGE_EXTENDER_BIT	The child is a range extender.
EMBER_CHILD_FLAGS_DEVICE_IS_SLEEPY_BIT	The child is a sleepy end device.
EMBER_CHILD_FLAGS_HAVE_PENDING_DATA_BIT	The node has data pending for the child.
EMBER_CHILD_FLAGS_AES_SECURITY_CAPABLE_BIT	The child supports AES-based security.
EMBER_CHILD_FLAG_DEVICE_IS_EXTENDED_BIT	The child is connected through range extender.

EmberMessageOptions

EmberMessageOptions

Message options.

Enumerator	
EMBER_OPTIONS_NONE	No options.
EMBER_OPTIONS_SECURITY_ENABLED	The packet should be sent out encrypted.
EMBER_OPTIONS_ACK_REQUESTED	An acknowledgment should be requested for the outgoing packet.
EMBER_OPTIONS_HIGH_PRIORITY	The packet should be sent with high priority. High priority messages are added to the head of transmit queue, other messages are added to the tail.
EMBER_OPTIONS_INDIRECT	The packet should be sent via the indirect queue. This option only applies to nodes with EMBER_MAC_MODE_DEVICE or EMBER_MAC_MODE_SLEEPY_DEVICE node type.

EmberMacAddressMode

EmberMacAddressMode

802.15.4 addressing mode.

Enumerator	
EMBER_MAC_ADDRESS_MODE_NONE	No address on this field.
EMBER_MAC_ADDRESS_MODE_SHORT	Short addressing mode.
EMBER_MAC_ADDRESS_MODE_LONG	Long addressing mode.

EmberEventUnits

EmberEventUnits

Copyright © 2025 Silicon Laboratories. All rights reserved.

Either marks an event as inactive or specifies the units for the event execution time.

Enumerator	
EMBER_EVENT_INACTIVE	The event is not scheduled to run.
EMBER_EVENT_MS_TIME	The execution time is in approximate milliseconds.
EMBER_EVENT_QS_TIME	The execution time is in 'binary' quarter seconds (256 timer ticks).
EMBER_EVENT_MINUTE_TIME	The execution time is in 'binary' minutes (65536 timer ticks).
EMBER_EVENT_ZERO_DELAY	The event is scheduled to run at the earliest opportunity.

@2

> @2

Define tasks that prevent the stack from sleeping.

Enumerator	
EMBER_OUTGOING_MESSAGES	Messages are waiting for transmission.
EMBER_INCOMING_MESSAGES	One or more incoming messages are processed.
EMBER_RADIO_IS_ON	The radio is currently powered on. On sleepy devices, the radio is turned off when not in use. On non-sleepy devices, i.e., EMBER_STAR_COORDINATOR, EMBER_STAR_RANGE_EXTENDER, EMBER_STAR_END_DEVICE, or EMBER_MAC_MODE_DEVICE, the radio is always on.
EMBER_ASSOCIATING	The node is currently trying to associate to a Connect network.
EMBER_SCANNING	The node is currently performing a MAC-level scanning procedure.

EmberCounterType

> EmberCounterType

Define the event counters that can be requested from the application using emberGetCounter()

Enumerator	
EMBER_COUNTER_PHY_IN_PACKETS	Every packet that comes in over the radio (except MAC ACKs).
EMBER_COUNTER_PHY_OUT_PACKETS	Every packet that goes out over the radio (except MAC ACKs).
EMBER_COUNTER_MAC_IN_UNICAST	Incoming MAC layer unicasts.
EMBER_COUNTER_MAC_IN_BROADCAST	Incoming MAC layer broadcasts.
EMBER_COUNTER_MAC_OUT_UNICAST_NO_ACK	Outgoing MAC layer unicasts that do not require an ACK.
EMBER_COUNTER_MAC_OUT_UNICAST_ACK_SUCCESS	Outgoing MAC layer unicasts that required an ACK, for which an ACK was received (possibly after retrying).
EMBER_COUNTER_MAC_OUT_UNICAST_ACK_FAIL	Outgoing MAC layer unicasts that required an ACK, for which an ACK was never received (even after retrying).
EMBER_COUNTER_MAC_OUT_UNICAST_CCA_FAIL	Outgoing MAC layer packets which were never transmitted because clear channel assessment always returned busy.
EMBER_COUNTER_MAC_OUT_UNICAST_RETRY	Outgoing MAC layer unicast retries. This does not count the initial transmission. Note a single MAC transmission can result in multiple retries.
EMBER_COUNTER_MAC_OUT_BROADCAST	Outgoing MAC layer broadcasts

Copyright © 2025 Silicon Laboratories. All rights reserved.

EMBER_COUNTER_MAC_OUT_BROADCAST_CCA_FAIL	Outgoing MAC layer broadcasts which were never transmitted because clear channel assessment always returned busy.
EMBER_COUNTER_MAC_OUT_ENCRYPT_FAIL	Outgoing MAC layer messages which were not transmitted due to security error e.g. because security key was not set.
EMBER_COUNTER_MAC_DROP_IN_MEMORY	Dropped incoming MAC layer packets (out of memory)
EMBER_COUNTER_MAC_DROP_IN_FRAME_COUNTER	Dropped incoming MAC layer packets (invalid frame counter)
EMBER_COUNTER_MAC_DROP_IN_DECRYPT	Dropped incoming MAC layer packets (can't decrypt)
EMBER_COUNTER_NWK_OUT_FORWARDING	Outgoing Network layer forwarded packets
EMBER_COUNTER_NWK_IN_SUCCESS	Incoming Network layer data frames correctly processed
EMBER_COUNTER_NWK_DROP_IN_WRONG_SOURCE	Dropped incoming Network layer packets (wrong source node)
EMBER_COUNTER_NWK_DROP_IN_FORWARDING	Dropped incoming Network layer packets (can't forward)
EMBER_COUNTER_UART_IN_DATA	NCP UART in data packet
EMBER_COUNTER_UART_IN_MANAGEMENT	NCP UART in management packet
EMBER_COUNTER_UART_IN_FAIL	NCP UART in frame errors
EMBER_COUNTER_UART_OUT_DATA	NCP UART out data packet
EMBER_COUNTER_UART_OUT_MANAGEMENT	NCP UART out management packet
EMBER_COUNTER_UART_OUT_FAIL	NCP UART out frame errors
EMBER_COUNTER_ROUTE_2_HOP_LOOP	
EMBER_COUNTER_BUFFER_ALLOCATION_FAIL	
EMBER_ASH_V3_ACK_SENT	ASHv3 ACK frame sent
EMBER_ASH_V3_ACK_RECEIVED	ASHv3 ACK frame received
EMBER_ASH_V3_NACK_SENT	ASHv3 NACK frame sent
EMBER_ASH_V3_NACK_RECEIVED	ASHv3 NACK frame received
EMBER_ASH_V3_RESEND	ASHv3 resends of frames that were not acked
EMBER_ASH_V3_BYTES_SENT	ASHv3 number of bytes sent
EMBER_ASH_V3_TOTAL_BYTES_RECEIVED	ASHv3 number of bytes received
EMBER_ASH_V3_VALID_BYTES_RECEIVED	ASHv3 number of valid bytes received
EMBER_ASH_V3_PAYLOAD_BYTES_SENT	ASHv3 number of payload bytes sent
EMBER_COUNTER_TYPE_COUNT	A placeholder giving the number of Ember counter types.

EmberPhyType

> EmberPhyType

Define the PHY configuration of connect stack.

Enumerator	
EMBER_RADIO_CONFIGURATOR	Connect uses radio configurator PHY
EMBER_STANDARD_PHY_2_4GHZ	Connect is configured to use only 2.4GHz standard PHY
EMBER_STANDARD_PHY_915MHZ	Connect is configured to use only 915MHz standard PHY
EMBER_STANDARD_PHY_863MHZ	Connect is configured to use only 863MHz standard PHY

EmberCalType

Copyright © 2025 Silicon Laboratories. All rights reserved.

EmberCalType

Define the type of calibration requested.

Enumerator	
EMBER_CAL_TEMP_VCO	Temperature calibration
EMBER_CAL_IRCAL	Image Rejection calibration
EMBER_CAL_ALL	Both calibrations

EmberTxStreamParameters

EmberTxStreamParameters

Radio Stream mode.

Enumerator	
TX_STREAM_PN9	Pseudo random PN9 byte sequence
TX_STREAM_CW	An unmodulated carrier wave

Typedef Documentation

EmberEUI64

typedef uint8_t EmberEUI64[EUI64_SIZE] [EUI64_SIZE]

EUI 64-bit ID (IEEE 802.15.4 long address).

EmberNodeId

typedef uint16_t EmberNodeId

IEEE 802.15.4 node ID. Also known as short address.

EmberPanId

typedef uint16_t EmberPanId

IEEE 802.15.4 PAN ID.

EmberMessageLength

typedef uint16_t EmberMessageLength

Message length in bytes.

Copyright © 2025 Silicon Laboratories. All rights reserved.

EmberTaskId

```
typedef uint8_t EmberTaskId
```

An identifier for a task.

sl_connect_radio_priority_t

```
typedef struct sl_connect_radio_priority sl_connect_radio_priority_t
```

Struct used to specify priorities for Connect's radio operations in multiprotocol.

Event

```
typedef struct Event_s Event
```

EventQueue

```
typedef struct EventQueue_s EventQueue
```

An event queue is currently just a list of events ordered by execution time.

EmberEventData

```
typedef const struct EmberEventData_S EmberEventData
```

Complete events with a control and a handler procedure.

An application typically creates an array of events along with their handlers. The main loop passes the array to emberRunEvents() to call the handlers of any events whose time has arrived.

EmberBuffer

```
typedef uint16_t EmberBuffer
```

Buffers used by the memory buffer system.

Function Documentation

emberKeyContents

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
uint8_t * emberKeyContents (EmberKeyData * key)
```

This macro allows the programmer to gain access to the key data bytes of the EmberKeyData structure.

Parameters

Type	Direction	Argument Name	Description
EmberKeyData *	[in]	key	A Pointer to an EmberKeyData structure.

Returns

- uint8_t* Returns a pointer to the first byte of the key data.

Copyright © 2025 Silicon Laboratories. All rights reserved.

EmberNetworkParameters

Hold network parameters.

For information about power settings and radio channels, see the technical specification for the RF communication module in your Developer Kit and the Radio Configurator Guide (AN971).

Public Attributes

uint16_t	panId
int16_t	radioTxPower
uint16_t	radioChannel

Public Attribute Documentation

panId

```
uint16_t EmberNetworkParameters::panId
```

The network's PAN identifier.

radioTxPower

```
int16_t EmberNetworkParameters::radioTxPower
```

The transmit power setting, in deci-dBm.

radioChannel

```
uint16_t EmberNetworkParameters::radioChannel
```

The radio channel. Be sure to specify a channel supported by the radio.

Copyright © 2025 Silicon Laboratories. All rights reserved.

EmberIncomingMessage

An instance of this structure is passed to emberIncomingMessageHandler(). It describes the incoming message.

Public Attributes

EmberMessageOptions	options
EmberNodeId	source
uint8_t	endpoint
int8_t	rssi
EmberMessageLength	length
uint8_t *	payload
uint32_t	timestamp
uint8_t	lqi

Public Attribute Documentation

options

```
EmberMessageOptions EmberIncomingMessage::options
```

An EmberMessageOptions value indicating the options used for the incoming packet.

source

```
EmberNodeId EmberIncomingMessage::source
```

An EmberNodeId value indicating source node ID.

endpoint

```
uint8_t EmberIncomingMessage::endpoint
```

The endpoint the message is destined to.

rssi

```
int8_t EmberIncomingMessage::rssi
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

The RSSI in dBm the packet was received with.

length

```
EmberMessageLength EmberIncomingMessage::length
```

An EmberMessageLength value indicating the length in bytes of the incoming message.

payload

```
uint8_t* EmberIncomingMessage::payload
```

A pointer to the message payload.

timestamp

```
uint32_t EmberIncomingMessage::timestamp
```

The millisecond system time returned by emberGetInt32uMillisecondTick() at the time the sync word was detected.

lqi

```
uint8_t EmberIncomingMessage::lqi
```

The LQI the packet was received with.

Copyright © 2025 Silicon Laboratories. All rights reserved.

EmberOutgoingMessage

An instance of this structure is passed to emberMessageSentHandler(). It describes the outgoing packet.

Public Attributes

EmberMessageOptions	options
EmberNodeId	destination
uint8_t	endpoint
uint8_t	tag
EmberMessageLength	length
uint8_t *	payload
int8_t	ackRssi
uint32_t	timestamp

Public Attribute Documentation

options

```
EmberMessageOptions EmberOutgoingMessage::options
```

An EmberMessageOptions value indicating the options used for transmitting the outgoing message.

destination

```
EmberNodeId EmberOutgoingMessage::destination
```

An EmberNodeId value indicating the destination short ID.

endpoint

```
uint8_t EmberOutgoingMessage::endpoint
```

The endpoint the message is destined to.

tag

```
uint8_t EmberOutgoingMessage::tag
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

A tag value the application can use to match emberMessageSend() calls to the corresponding emberMessageSentHandler() calls.

length

```
EmberMessageLength EmberOutgoingMessage::length
```

An EmberMessageLength value indicating the length in bytes of the incoming message.

payload

```
uint8_t* EmberOutgoingMessage::payload
```

A pointer to the message payload.

ackRssi

```
int8_t EmberOutgoingMessage::ackRssi
```

The RSSI in dBm of the ACK corresponding to this message. This field is meaningful only if EMBER_OPTIONS_ACK_REQUESTED flag is set in the options field.

timestamp

```
uint32_t EmberOutgoingMessage::timestamp
```

The millisecond system time returned by ::sl_sleeptimer at the time the sync word was transmitted.

Copyright © 2025 Silicon Laboratories. All rights reserved.

EmberMacAddress

A structure that stores an 802.15.4 address.

Public Attributes

uint8_t	longAddress Long (EUI-64) address. Valid if `mode` is EMBER_MAC_ADDRESS_MODE_LONG.
uint16_t	shortAddress Short address (node ID). Valid if `mode` is EMBER_MAC_ADDRESS_MODE_SHORT.
union EmberMacAddress::@3	addr
EmberMacAddressMode	mode

Public Attribute Documentation

longAddress

```
uint8_t EmberMacAddress::longAddress[EUI64_SIZE]
```

Long (EUI-64) address. Valid if `mode` is EMBER_MAC_ADDRESS_MODE_LONG.

shortAddress

```
uint16_t EmberMacAddress::shortAddress
```

Short address (node ID). Valid if `mode` is EMBER_MAC_ADDRESS_MODE_SHORT.

addr

```
union EmberMacAddress::@3 EmberMacAddress::addr
```

mode

```
EmberMacAddressMode EmberMacAddress::mode
```

Addressing mode

Copyright © 2025 Silicon Laboratories. All rights reserved.

EmberMacFrame

A structure that describes the addressing fields of a 802.15.4 frame.

Public Attributes

EmberMacAddress	srcAddress
EmberMacAddress	dstAddress
EmberPanId	srcPanId
EmberPanId	dstPanId
bool	srcPanIdSpecified
bool	dstPanIdSpecified

Public Attribute Documentation

srcAddress

```
EmberMacAddress EmberMacFrame::srcAddress
```

An EmberMacAddress structure indicating the source address of a MAC frame.

dstAddress

```
EmberMacAddress EmberMacFrame::dstAddress
```

An EmberMacAddress structure indicating the destination address of a MAC frame.

srcPanId

```
EmberPanId EmberMacFrame::srcPanId
```

An EmberPanId struct indicating the source PAN ID of a MAC frame. This field is meaningful only if srcPanIdSpecified is set to true.

dstPanId

```
EmberPanId EmberMacFrame::dstPanId
```

An EmberPanId struct indicating the destination PAN ID of a MAC frame. This field is meaningful only if dstPanIdSpecified is set to true.

Copyright © 2025 Silicon Laboratories. All rights reserved.

srcPanIdSpecified

```
bool EmberMacFrame::srcPanIdSpecified
```

True if the srcPanId field is set, false otherwise.

dstPanIdSpecified

```
bool EmberMacFrame::dstPanIdSpecified
```

True if the dstPanId field is set, false otherwise.

Copyright © 2025 Silicon Laboratories. All rights reserved.

EmberIncomingMacMessage

An instance of this structure is passed to emberIncomingMacMessageHandler(). It describes the incoming MAC frame.

Public Attributes

EmberMessageOptions	options
EmberMacFrame	macFrame
int8_t	rssi
uint8_t	lqi
uint32_t	frameCounter
EmberMessageLength	length
uint8_t *	payload
uint32_t	timestamp

Public Attribute Documentation

options

```
EmberMessageOptions EmberIncomingMacMessage::options
```

An EmberMessageOptions value indicating the options used for the incoming packet.

macFrame

```
EmberMacFrame EmberIncomingMacMessage::macFrame
```

An EmberMacFrame structure indicating the source and destination addresses and source and destination PAN IDs.

rssi

```
int8_t EmberIncomingMacMessage::rssi
```

The RSSI in dBm the packet was received with.

lqi

```
uint8_t EmberIncomingMacMessage::lqi
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

The LQI the packet was received with.

frameCounter

```
uint32_t EmberIncomingMacMessage::frameCounter
```

The security MAC frame counter (if any).

length

```
EmberMessageLength EmberIncomingMacMessage::length
```

An EmberMessageLength value indicating the length in bytes of the MAC payload of the incoming message.

payload

```
uint8_t* EmberIncomingMacMessage::payload
```

A pointer to the message MAC payload.

timestamp

```
uint32_t EmberIncomingMacMessage::timestamp
```

The millisecond system time returned by ::sl_sleeptimer at the time the sync word was detected.

Copyright © 2025 Silicon Laboratories. All rights reserved.

EmberOutgoingMacMessage

An instance of this structure is passed to emberMacMessageSentHandler(). It describes the outgoing MAC frame.

Public Attributes

EmberMessageOptions	options
EmberMacFrame	macFrame
uint8_t	tag
uint32_t	frameCounter
EmberMessageLength	length
uint8_t *	payload
int8_t	ackRssi
uint32_t	timestamp

Public Attribute Documentation

options

```
EmberMessageOptions EmberOutgoingMacMessage::options
```

An EmberMessageOptions value indicating the options used for transmitting the outgoing message.

macFrame

```
EmberMacFrame EmberOutgoingMacMessage::macFrame
```

An EmberMacFrame struct indicating the source and destination addresses and source and destination PAN IDs of the outgoing MAC frame.

tag

```
uint8_t EmberOutgoingMacMessage::tag
```

A tag value the application can use to match emberMacMessageSend() calls to the corresponding emberMacMessageSentHandler() calls.

frameCounter

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
uint32_t EmberOutgoingMacMessage::frameCounter
```

The security frame counter of the outgoing MAC frame (if any).

length

```
EmberMessageLength EmberOutgoingMacMessage::length
```

An EmberMessageLength value indicating the length in bytes of the incoming message.

payload

```
uint8_t* EmberOutgoingMacMessage::payload
```

A pointer to the message payload.

ackRssi

```
int8_t EmberOutgoingMacMessage::ackRssi
```

The RSSI in dBm of the ACK corresponding to this message. This field is meaningful only if EMBER_OPTIONS_ACK_REQUESTED flag is set in the options field.

timestamp

```
uint32_t EmberOutgoingMacMessage::timestamp
```

The millisecond system time returned by ::sl_sleeptimer at the time the sync word was transmitted.

Copyright © 2025 Silicon Laboratories. All rights reserved.

EmberKeyData

This data structure contains the security key, most prominently used by emberSetSecurityKey.

Public Attributes

uint8_t contents

Public Attribute Documentation

contents

```
uint8_t EmberKeyData::contents[EMBER_ENCRYPTION_KEY_SIZE]
```

This is the key byte data.

Copyright © 2025 Silicon Laboratories. All rights reserved.

sl_connect_radio_priority

Struct used to specify priorities for Connect's radio operations in multiprotocol.

Public Attributes

uint8_t	rx
uint8_t	starting_tx
uint8_t	tx_step

Public Attribute Documentation

rx

```
uint8_t sl_connect_radio_priority::rx
```

RX priority.

starting_tx

```
uint8_t sl_connect_radio_priority::starting_tx
```

Starting TX priority - First try of Tx will use this priority

tx_step

```
uint8_t sl_connect_radio_priority::tx_step
```

TX priority will decrement by this value at every retry

Copyright © 2025 Silicon Laboratories. All rights reserved.

EventActions

The static part of an event. Each event can be used with only one event queue.

Public Attributes

struct EventQueue_s *	queue
void(*	handler
void(*	marker
const char *	name

Public Attribute Documentation

queue

```
struct EventQueue_s* EventActions::queue
```

handler

```
void(* EventActions::handler) (struct Event_s *)
```

marker

```
void(* EventActions::marker) (struct Event_s *)
```

name

```
const char* EventActions::name
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Event_s

Public Attributes

EventActions *	actions
struct Event_s *	next
uint32_t	timeToExecute

Public Attribute Documentation

actions

```
EventActions* Event_s::actions
```

next

```
struct Event_s* Event_s::next
```

timeToExecute

```
uint32_t Event_s::timeToExecute
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

EventQueue_s

An event queue is currently just a list of events ordered by execution time.

Public Attributes

Event *	isrEvents
Event *	events

Public Attribute Documentation

isrEvents

```
Event* EventQueue_s::isrEvents
```

events

```
Event* EventQueue_s::events
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

EmberEventControl

Control structure for events.

This structure should not be accessed directly. It holds the event status (one of the **EMBER_EVENT_** values) and the time left before the event fires.

Public Attributes

EmberEventUnits	status
EmberTaskId	taskid
uint32_t	timeToExecute

Public Attribute Documentation

status

```
EmberEventUnits EmberEventControl::status
```

The event's status, either inactive or the units for timeToExecute.

taskid

```
EmberTaskId EmberEventControl::taskid
```

The task ID this event belongs to.

timeToExecute

```
uint32_t EmberEventControl::timeToExecute
```

How long before the event fires. Units are always in milliseconds.

Copyright © 2025 Silicon Laboratories. All rights reserved.

EmberEventData_S

Complete events with a control and a handler procedure.

An application typically creates an array of events along with their handlers. The main loop passes the array to emberRunEvents() to call the handlers of any events whose time has arrived.

Public Attributes

EmberEventControl *	control
void(*	handler

Public Attribute Documentation

control

```
EmberEventControl* EmberEventData_S::control
```

The control structure for the event.

handler

```
void(* EmberEventData_S::handler) (void)
```

The procedure to call when the event fires.

Copyright © 2025 Silicon Laboratories. All rights reserved.

EmberTaskControl

Control structure for tasks.

This structure should not be accessed directly.

Public Attributes

uint32_t	nextEventTime
EmberEventData *	events
bool	busy

Public Attribute Documentation

nextEventTime

```
uint32_t EmberTaskControl::nextEventTime
```

The time when the next event associated with this task will fire

events

```
EmberEventData* EmberTaskControl::events
```

The list of events associated with this task

busy

```
bool EmberTaskControl::busy
```

A flag that indicates the task has something to do other than events

Copyright © 2025 Silicon Laboratories. All rights reserved.

Stack Information

Stack Information

Connect API for accessing and modifying stack states and behaviors.

See stack-info.h for source code.

Modules

Stack Counters

Handlers

void	emberStackStatusHandler(EmberStatus status) A callback invoked when the status of the stack changes.
void	emberStackIsrHandler(void) This handler is invoked in ISR context when certain stack-related ISR routines fire.
bool	emberStackIdleHandler(uint32_t *idleTimeMs) A callback to allow the application to manage idling the MCU.
void	emberRadioNeedsCalibratingHandler(void) The radio calibration callback function.
void	emberChildJoinHandler(EmberNodeType nodeType, EmberNodeId nodeId) Invoked at coordinator, range extender, or mac mode nodes when a new child has joined the device.

APIs

void	emberStackPowerDown(void) Immediately turns the radio power completely off.
void	emberStackPowerUp(void) Power up the radio. Typically called coming out of sleep.
EmberNetworkStatus	emberNetworkState(void) Return the current join status.
bool	emberStackIsUp(void) Indicate whether the stack is currently up.
EmberStatus	emberSetSecurityKey(EmberKeyData *key) Write a key at the address of the formerly used security key. The key set by this function will not be used by the stack. The API is meant to be used to erase a key as it is now managed by PSA Crypto API.
EmberStatus	emberGetSecurityKey(EmberKeyData *key) Get the legacy security key. This function does not return the value set with the PSA Crypto API.
EmberStatus	emberSetRadioRxFifo(uint16_t rxFifoSize) Set the radio receive FIFO size.

Copyright © 2025 Silicon Laboratories. All rights reserved.

EmberStatus emberSetRadioChannelExtended(uint16_t channel, bool persistent)
Set the channel for sending and receiving messages on the current network. The available channels depend on the radio config you use. Channels can differ more than the frequency if it's a multi-PHY config.

EmberStatus emberSetRadioChannel(uint16_t channel)
Set the channel for sending and receiving messages on the current network. The available channels depend on the radio config you use. Channels can differ more than the frequency if it's a multi-PHY config.

uint16_t emberGetRadioChannel(void)
Get the radio channel, to which a node is set, on the current network. The available channels depend on the radio config you use. Channels can differ more than the frequency if it's a multi-PHY config.

uint16_t emberGetDefaultChannel(void)
Get the first available channel in the current radio configuration.

EmberStatus emberPhyConfigInit(EmberPhyType phyType)
Indicate if the PHY configuration of the stack. Currently only supporting EMBER_RADIO_CONFIGURATOR and EMBER_STANDARD_PHY_2_4GHZ. It must be called before initializing the stack.

EmberStatus emberCalibrateCurrentChannelExtended(uint32_t calValueIn, uint32_t *calValueOut)
Perform image rejection calibration on the current channel. The stack will notify the application that it needs channel calibration via the emberRadioNeedsCalibratingHandler() callback function during emberTick(). This function should only be called from within the context of the emberRadioNeedsCalibratingHandler() callback function. Note if this function is called when the radio is off, it will turn the radio on and leave it on.

EmberStatus emberCalibrateCurrentChannel(void)
Perform image rejection calibration on the current channel. The stack will notify the application that it needs channel calibration via the emberRadioNeedsCalibratingHandler() callback function during emberTick(). This function should only be called from within the context of the emberRadioNeedsCalibratingHandler() callback function. Note if this function is called when the radio is off, it will turn the radio on and leave it on.

EmberStatus emberApplyIrCalibration(uint32_t calValue)
Apply Image Rejection calibration on the current channel. The stack will notify the application that it needs channel calibration via the emberRadioNeedsCalibratingHandler() callback function during emberTick(). This function should only be called from within the context of the emberRadioNeedsCalibratingHandler() callback function. Note if this function is called when the radio is off, it will turn the radio on and leave it on.

EmberStatus emberTempCalibration(void)
Perform Temperature VCO calibration calibration on the current channel. The stack will notify the application that it needs channel calibration via the emberRadioNeedsCalibratingHandler() callback function during emberTick(). This function should only be called from within the context of the emberRadioNeedsCalibratingHandler() callback function. Note if this function is called when the radio is off, it will turn the radio on and leave it on.

EmberCalType emberGetCalType(void)
Fetch calibration type associated to the latest emberRadioNeedsCalibratingHandler() callback.

EmberStatus emberSetRadioPower(int16_t power, bool persistent)
Set the radio output power at which a node is to operate for the current network. The radio has a finite power resolution, so it will approximate the requested power with the closest possible value at or below the requested value.

int16_t emberGetRadioPower(void)
Get the radio output power of the current network at which a node is operating. This might be different to what you set using emberSetRadioPower because the radio has a finite power resolution, and emberSetRadioPower must approximate to the closest possible value at or below the requested value. This API however returns with the actual setting.

EmberStatus emberSetRadioPowerMode(bool radioOn)
Allow the application to turn the radio on/off. This API is intended for use with direct devices only.

EmberStatus emberSetUnencryptedPacketsAcceptance(bool accept)
Set if the stack should forward unencrypted messages to the application.

Copyright © 2025 Silicon Laboratories. All rights reserved.

EmberStatus emberSetMacParams(int8_t ccaThreshold, uint8_t maxCcaAttempts, uint8_t minBackoffExp, uint8_t maxBackoffExp, uint16_t ccaBackoff, uint16_t ccaDuration, uint8_t maxRetries, uint32_t csmaTimeout, uint16_t ackTimeout)
Set the MAC layer transmission parameters.

EmberStatus emberMacGetParentAddress(EmberMacAddress *parentAddress)
Retrieve the parent address. This API can be invoked only for nodes of EMBER_MAC_MODE_DEVICE or EMBER_MAC_MODE_SLEEPY_DEVICE type.

uint32_t emberStackIdleTimeMs(uint16_t *currentStackTasks)
Return the time in milliseconds the stack could idle for.

uint32_t emberGetInt32uMillisecondTick(void)
Return the current time in milliseconds.

uint16_t emberCurrentStackTasks(void)
Return a bitmask indicating the stack's current tasks.

bool emberOkToNap(void)
Indicate whether the stack is currently in a state with no high-priority tasks and may sleep.

bool emberOkToHibernate(void)
Indicate whether the stack currently has any pending tasks.

uint8_t * emberGetEui64(void)
Return the EUI64 ID of the local node.

bool emberIsLocalEui64(EmberEUI64 eui64)
Determine whether `eui64` is the local node's EUI64 ID. EUI64 is easily accessible in SoC mode, but in Host-NCP, the address is stored on the NCP. This API can be used on the Host to compare a value with the locally stored one.

EmberNodeId emberGetNodeId(void)
Return the 16-bit node ID of local node on the current network.

EmberPanId emberGetPanId(void)
Return the local node's PAN ID of the current network.

EmberNodeType emberGetNodeType(void)
Return an EmberNodeType value indicating the type of the node.

EmberNodeId emberGetParentId(void)
Return the parent's node ID.

EmberStatus emberGetVersionInfo(uint16_t *platform_version, uint16_t *connect_stack_version, uint32_t *bootloader_version)
Get the Platform, Stack and bootloader versions all at once. The version format are not all the same. Please refer to the corresponding documentation to handle the information correctly. The Platform version corresponds to the deprecated GSDK version.

Macros

#define EMBER_HIGH_PRIORITY_TASKS (EMBER_OUTGOING_MESSAGES | EMBER_INCOMING_MESSAGES | EMBER_RADIO_IS_ON)
A mask of the tasks that prevent a device from sleeping.

#define EMBER_INVALID_CHANNEL 65535
Invalid channel number.

Handlers Documentation

Copyright © 2025 Silicon Laboratories. All rights reserved.

emberStackStatusHandler

```
void emberStackStatusHandler (EmberStatus status)
```

A callback invoked when the status of the stack changes.

Parameters

Type	Direction	Argument Name	Description
EmberStatus	[in]	status	Stack status. One of the following: • EMBER_NETWORK_UP • EMBER_NETWORK_DOWN • EMBER_NO_VALID_BEACONS • EMBER_JOIN_SCAN_FAILED • EMBER_JOIN_FAILED • EMBER_JOIN_DENIED • EMBER_JOIN_TIMEOUT • EMBER_MAC_SYNC_TIMEOUT

The application is free to begin messaging once it receives the EMBER_NETWORK_UP status.

emberStackIsrHandler

```
void emberStackIsrHandler (void )
```

This handler is invoked in ISR context when certain stack-related ISR routines fire.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

emberStackIdleHandler

```
bool emberStackIdleHandler (uint32_t * idleTimeMs)
```

A callback to allow the application to manage idling the MCU.

Parameters

Type	Direction	Argument Name	Description
uint32_t *	[inout]	idleTimeMs	A pointer to the time in millisecond the stack is allowed to idle. If the application decides to manage idling the MCU, it should update the passed value with the actual time the MCU was idled.

Returns

- true if the application is managing idling the MCU, false otherwise. If this function returns false, the stack will manage idling the MCU.

Copyright © 2025 Silicon Laboratories. All rights reserved.

emberRadioNeedsCalibratingHandler

```
void emberRadioNeedsCalibratingHandler (void )
```

The radio calibration callback function.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

This handler is invoked by the stack upon receiving a "calibration needed" event from the radio to inform the application that it should perform calibration of the current channel as soon as possible using the emberCalibrateCurrentChannel() API. While calibration only takes tens of microseconds, the application can failsafe any critical processes or peripherals before calling emberCalibrateCurrentChannel(). The application must call emberCalibrateCurrentChannel() in response to this callback to maintain expected radio performance.

emberChildJoinHandler

```
void emberChildJoinHandler (EmberNodeType nodeType, EmberNodeId nodeId)
```

Invoked at coordinator, range extender, or mac mode nodes when a new child has joined the device.

Parameters

Type	Direction	Argument Name	Description
EmberNodeType	[in]	nodeType	The role of the joining device (EMBER_STAR_RANGE_EXTENDER, EMBER_STAR_END_DEVICE, EMBER_STAR_SLEEPY_END_DEVICE, EMBER_MAC_MODE_DEVICE or EMBER_MAC_MODE_SLEEPY_DEVICE).
EmberNodeId	[in]	nodeId	The node ID of the joining device.

APIs Documentation

emberStackPowerDown

```
void emberStackPowerDown (void )
```

Immediately turns the radio power completely off.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

After calling this function, do not call any other stack function except emberStackPowerUp() because all other stack functions require that the radio is powered to operate properly.

emberStackPowerUp

```
void emberStackPowerUp (void )
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Power up the radio. Typically called coming out of sleep.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

For non-sleepy devices, also turns the radio on and leaves it in RX mode.

emberNetworkState

```
EmberNetworkStatus emberNetworkState (void )
```

Return the current join status.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns a value indicating whether the node is joining, joined to, or leaving a network.

Returns

- An EmberNetworkStatus value indicating the current join status.

emberStackIsUp

```
bool emberStackIsUp (void )
```

Indicate whether the stack is currently up.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns true if the stack is joined to a network and ready to send and receive messages. This reflects only the state of the local node and does not indicate whether or not other nodes are able to communicate with this node.

Returns

- **true** if the stack is up, **false** otherwise.

emberSetSecurityKey

```
EmberStatus emberSetSecurityKey (EmberKeyData * key)
```

Write a key at the address of the formerly used security key. The key set by this function will not be used by the stack. The API is meant to be used to erase a key as it is now managed by PSA Crypto API.

Parameters

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
EmberKeyData *	[in]	key	An EmberKeyData value containing the security key to be set.

Returns

- An EmberStatus value of EMBER_SUCCESS if the key was successfully set. Otherwise, it returns an EmberStatus value of EMBER_INVALID_CALL.

emberGetSecurityKey

> EmberStatus emberGetSecurityKey (EmberKeyData * key)

Get the legacy security key. This function does not return the value set with the PSA Crypto API.

Parameters

Type	Direction	Argument Name	Description
EmberKeyData *	[in]	key	An EmberKeyData where the legcay security key will be stored

Returns

- An EmberStatus value of EMBER_SUCCESS if the key was successfully read.

emberSetRadioRxFifo

> EmberStatus emberSetRadioRxFifo (uint16_t rxFifoSize)

Set the radio receive FIFO size.

Parameters

Type	Direction	Argument Name	Description
uint16_t	[in]	rxFifoSize	The size (in bytes) of the RX FIFO buffer.

This function sets the RAM memory area for the receive FIFO, a circular buffer used to hold received data from one or more packets.

The actual FIFO size can be determined by the output p_size_bytes argument. The chosen size is determined based on the available FIFO sizes supported by the hardware and will be the closest allowed size less than or equal to the requested size, unless the requested size is smaller than the minimum FIFO size, in which case an assert.

Returns

- An EmberStatus value indicating:
  - EMBER_SUCCESS if the RX FIFO is set successfully.
  - EMBER_NO_BUFFERS if buffer allocation fails.

emberSetRadioChannelExtended

> EmberStatus emberSetRadioChannelExtended (uint16_t channel, bool persistent)

Set the channel for sending and receiving messages on the current network. The available channels depend on the radio config you use. Channels can differ more than the frequency if it's a multi-PHY config.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Parameters

Type	Direction	Argument Name	Description
uint16_t	[in]	channel	A desired radio channel.
bool	[in]	persistent	A flag to instruct the stack to save the channel setting in persistent or not. Each persistent call triggers a token write. Excessive usage might cause flash to wear-out.

Note

- Care should be taken when using this API. All devices on a network must use the same channel.

Returns

- An EmberStatus value of:
  - EMBER_SUCCESS if the stack accepted the channel change.
  - EMBER_INVALID_CALL if the node is currently performing frequency hopping.
  - EMBER_PHY_INVALID_CHANNEL if the passed channel is invalid.
  - EMBER_MAC_BUSY if the MAC is currently performing a high priority task.

emberSetRadioChannel

> EmberStatus emberSetRadioChannel (uint16_t channel)

Set the channel for sending and receiving messages on the current network. The available channels depend on the radio config you use. Channels can differ more than the frequency if it's a multi-PHY config.

Parameters

Type	Direction	Argument Name	Description
uint16_t	[in]	channel	A desired radio channel.

Note

- Care should be taken when using this API. All devices on a network must use the same channel. Each call triggers a token write. Excessive usage might cause flash to wear-out.

Returns

- An EmberStatus value of:
  - EMBER_SUCCESS if the stack accepted the channel change.
  - EMBER_INVALID_CALL if the node is currently performing frequency hopping.
  - EMBER_PHY_INVALID_CHANNEL if the passed channel is invalid.
  - EMBER_MAC_BUSY if the MAC is currently performing a high priority task.

emberGetRadioChannel

> uint16_t emberGetRadioChannel (void)

Get the radio channel, to which a node is set, on the current network. The available channels depend on the radio config you use. Channels can differ more than the frequency if it's a multi-PHY config.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Copyright © 2025 Silicon Laboratories. All rights reserved.

Returns

- The current radio channel.

emberGetDefaultChannel

```
uint16_t emberGetDefaultChannel (void )
```

Get the first available channel in the current radio configuration.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- The first available channel in the radio configuration. 0xffff if error

emberPhyConfigInit

```
EmberStatus emberPhyConfigInit (EmberPhyType phyType)
```

Indicate if the PHY configuration of the stack. Currently only supporting EMBER_RADIO_CONFIGURATOR and EMBER_STANDARD_PHY_2_4GHZ. It must be called before initializing the stack.

Parameters

Type	Direction	Argument Name	Description
EmberPhyType	N/A	phyType	

Returns

- EMBER_BAD_ARGUMENT if phyType is incorrect EMBER_INVALID_CALL if API is called after stack initialization EMBER_SUCCESS otherwise

emberCalibrateCurrentChannelExtended

```
EmberStatus emberCalibrateCurrentChannelExtended (uint32_t calValueIn, uint32_t * calValueOut)
```

Perform image rejection calibration on the current channel. The stack will notify the application that it needs channel calibration via the emberRadioNeedsCalibratingHandler() callback function during emberTick(). This function should only be called from within the context of the emberRadioNeedsCalibratingHandler() callback function. Note if this function is called when the radio is off, it will turn the radio on and leave it on.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	calValueIn	the calibration value to use. Set to EMBER_CAL_INVALID_VALUE to perform automatic calibration.
uint32_t *	[out]	calValueOut	a pointer to the calibration value that was used. This parameter is ignored when set to NULL.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Returns

- An EmberStatus value indicating the success or failure of the command.

emberCalibrateCurrentChannel

```
EmberStatus emberCalibrateCurrentChannel (void )
```

Perform image rejection calibration on the current channel. The stack will notify the application that it needs channel calibration via the emberRadioNeedsCalibratingHandler() callback function during emberTick(). This function should only be called from within the context of the emberRadioNeedsCalibratingHandler() callback function. Note if this function is called when the radio is off, it will turn the radio on and leave it on.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

emberApplyIrCalibration

```
EmberStatus emberApplyIrCalibration (uint32_t calValue)
```

Apply Image Rejection calibration on the current channel. The stack will notify the application that it needs channel calibration via the emberRadioNeedsCalibratingHandler() callback function during emberTick(). This function should only be called from within the context of the emberRadioNeedsCalibratingHandler() callback function. Note if this function is called when the radio is off, it will turn the radio on and leave it on.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	calValue	the calibration value to apply. Should not be set to EMBER_CAL_INVALID_VALUE.

Returns

- An EmberStatus value indicating the success or failure of the command.

emberTempCalibration

```
EmberStatus emberTempCalibration (void )
```

Perform Temperature VCO calibration calibration on the current channel. The stack will notify the application that it needs channel calibration via the emberRadioNeedsCalibratingHandler() callback function during emberTick(). This function should only be called from within the context of the emberRadioNeedsCalibratingHandler() callback function. Note if this function is called when the radio is off, it will turn the radio on and leave it on.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- An EmberStatus value indicating the success or failure of the command.

Copyright © 2025 Silicon Laboratories. All rights reserved.

emberGetCalType

```
EmberCalType emberGetCalType (void )
```

Fetch calibration type associated to the latest emberRadioNeedsCalibratingHandler() callback.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- An EmberCalType value indicating which type of calibration should be performed.

emberSetRadioPower

```
EmberStatus emberSetRadioPower (int16_t power, bool persistent)
```

Set the radio output power at which a node is to operate for the current network. The radio has a finite power resolution, so it will approximate the requested power with the closest possible value at or below the requested value.

Parameters

Type	Direction	Argument Name	Description
int16_t	[in]	power	Desired radio output power, in deci-dBm.
bool	[in]	persistent	A flag to instruct the stack to save the power setting in persistent or not.

Note

- Care should be taken when using this API on a running network, because it directly impacts the established link qualities neighboring nodes have with the node on which it is called. This can lead to disruption of existing routes and erratic network behavior.

Returns

- An EmberStatus value indicating the success or failure of the command. Failure indicates that the requested power level is out of range.

emberGetRadioPower

```
int16_t emberGetRadioPower (void )
```

Get the radio output power of the current network at which a node is operating. This might be different to what you set using emberSetRadioPower because the radio has a finite power resolution, and emberSetRadioPower must approximate to the closest possible value at or below the requested value. This API however returns with the actual setting.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Current radio output power, in deci-dBm.

Copyright © 2025 Silicon Laboratories. All rights reserved.

emberSetRadioPowerMode

```
EmberStatus emberSetRadioPowerMode (bool radioOn)
```

Allow the application to turn the radio on/off. This API is intended for use with direct devices only.

Parameters

Type	Direction	Argument Name	Description
bool	[in]	radioOn	If this parameter is **true**, the radio is turned on, otherwise it's turned off.

Returns

- An EmberStatus value indicating the success or failure of the command. Failure indicates that the node type is a type other than EMBER_DIRECT_DEVICE.

emberSetUnencryptedPacketsAcceptance

```
EmberStatus emberSetUnencryptedPacketsAcceptance (bool accept)
```

Set if the stack should forward unencrypted messages to the application.

Parameters

Type	Direction	Argument Name	Description
bool	[in]	accept	If this parameter is **true**, unencrypted messages are forwarded to the application. They are not if this parameter is **false**.

Returns

- An EmberStatus value of EMBER_INVALID_CALL if trying to filter out unencrypted message and if the stack cannot receive encrypted messages (eg. no security key is set). An EmberStatus value of EMBER_SUCCESS otherwise.

emberSetMacParams

```
EmberStatus emberSetMacParams (int8_t ccaThreshold, uint8_t maxCcaAttempts, uint8_t minBackoffExp, uint8_t maxBackoffExp, uint16_t ccaBackoff, uint16_t ccaDuration, uint8_t maxRetries, uint32_t csmaTimeout, uint16_t ackTimeout)
```

Set the MAC layer transmission parameters.

Parameters

Type	Direction	Argument Name	Description
int8_t	[in]	ccaThreshold	The CCA RSSI threshold, in dBm, above which the channel is considered 'busy'. This parameter is by default set to EMBER_RADIO_CCA_THRESHOLD.
uint8_t	[in]	maxCcaAttempts	The maximum number of clear channel assessment attempts that are performed prior to fail to transmit a packet with EMBER_PHY_TX_CCA_FAIL status. This parameter is set by default to 4. If this parameter is set to 0, the CCA assessment shall not be performed.
uint8_t	[in]	minBackoffExp	The backoff exponent used if the initial channel clear assessment fails. This parameter is set by default to 3. Note: this is meaningful only if the checkCca parameter is set to **true**.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
uint8_t	[in]	maxBackoffExp	The backoff exponent used if the final channel clear assessment fails. This parameter is set by default to 5. Note: this is meaningful only if the checkCca parameter is set to **true**.
uint16_t	[in]	ccaBackoff	The backoff unit period in microsecond. It is multiplied by the random backoff exponential controlled by minBackoffExp and maxBackoffExp to determine the overall backoff period. This parameter is set by default to the PHY symbol time in microseconds multiplied by 20.
uint16_t	[in]	ccaDuration	The minimum desired CCA check duration in microseconds. This parameter is set by default to the PHY symbol time in microseconds multiplied by 8.
uint8_t	[in]	maxRetries	The number of transmission retries that is performed if no acknowledgment was received. This parameter is set by default to 3 (which means that a total of 4 transmission attempts will be performed).
uint32_t	[in]	csmaTimeout	An overall timeout in microsecond time base for the the CSMA operations. This value is set by default to **0** which means that no timeout is imposed.
uint16_t	[in]	ackTimeout	The ack timeout in microseconds after which the transmitting gives up waiting for an acknowledgment. This parameter is set by default to (EMBER_MAC_ACK_TIMEOUT_MS * 1000).

Note

- The CSMA/CA (CCA) values are directly used in RAIL's `sl_rail_csma_config_t` and further information can be found in the RAIL API documentation.

Returns

- An EmberStatus value indicating whether the MAC parameters were successfully set or the reason of failure.

emberMacGetParentAddress

> EmberStatus emberMacGetParentAddress (EmberMacAddress * parentAddress)

Retrieve the parent address. This API can be invoked only for nodes of EMBER_MAC_MODE_DEVICE or EMBER_MAC_MODE_SLEEPY_DEVICE type.

Parameters

Type	Direction	Argument Name	Description
EmberMacAddress *	N/A	parentAddress	

Returns

- An EmberStatus value of EMBER_SUCCESS if the parent address was successfully retrieved, otherwise an EmberStatus value indicating the reason of failure.

emberStackIdleTimeMs

> uint32_t emberStackIdleTimeMs (uint16_t * currentStackTasks)

Return the time in milliseconds the stack could idle for.

Parameters

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
uint16_t *	[in]	currentStackTasks	A pointer to an integer that is written with the active stack tasks at the time of the API call.

Returns

- Allowed idle time in milliseconds.

emberGetInt32uMillisecondTick

```
uint32_t emberGetInt32uMillisecondTick (void )
```

Return the current time in milliseconds.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Current time in milliseconds.

emberCurrentStackTasks

```
uint16_t emberCurrentStackTasks (void )
```

Return a bitmask indicating the stack's current tasks.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

The mask EMBER_HIGH_PRIORITY_TASKS defines which tasks are high-priority. Devices should not sleep if any high-priority tasks are active. Active tasks that are not high-priority are waiting for messages to arrive from other devices. If there are active tasks, but no high-priority ones, the device may sleep but should periodically wake up and call emberPollForData() to receive messages. Parents will hold messages for EMBER_INDIRECT_TRANSMISSION_TIMEOUT_MS milliseconds before discarding them.

Returns

- A bitmask of the stack's active tasks.

emberOkToNap

```
bool emberOkToNap (void )
```

Indicate whether the stack is currently in a state with no high-priority tasks and may sleep.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Copyright © 2025 Silicon Laboratories. All rights reserved.

Tasks may be expecting incoming messages, in which case the device should periodically wake up and call emberPollForData() to receive messages. This function can only be called when the node type is EMBER_STAR_SLEEPY_END_DEVICE.

Returns

- **true** if the application may sleep but the stack may be expecting incoming messages.

emberOkToHibernate

```
bool emberOkToHibernate (void )
```

Indicate whether the stack currently has any pending tasks.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

If no tasks are pending , emberTick() does not need to be called until next stack API function is called. This function can only be called when the node type is EMBER_STAR_SLEEPY_END_DEVICE.

Returns

- **true** if the application may sleep for as long as it wishes.

emberGetEui64

```
uint8_t * emberGetEui64 (void )
```

Return the EUI64 ID of the local node.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- The 64-bit ID.

emberIsLocalEui64

```
bool emberIsLocalEui64 (EmberEUI64 eui64)
```

Determine whether `eui64` is the local node's EUI64 ID. EUI64 is easily accessible in SoC mode, but in Host-NCP, the address is stored on the NCP. This API can be used on the Host to compare a value with the locally stored one.

Parameters

Type	Direction	Argument Name	Description
EmberEUI64	[in]	eui64	An EUI64 ID.

Returns

Copyright © 2025 Silicon Laboratories. All rights reserved.

- **true** if `eui64` is the local node's ID, otherwise **false**.

emberGetNodeId

```
EmberNodeId emberGetNodeId (void )
```

Return the 16-bit node ID of local node on the current network.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- The 16-bit ID. Byte order is little endian.

emberGetPanId

```
EmberPanId emberGetPanId (void )
```

Return the local node's PAN ID of the current network.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- The PAN ID.

emberGetNodeType

```
EmberNodeType emberGetNodeType (void )
```

Return an EmberNodeType value indicating the type of the node.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- The node type.

emberGetParentId

```
EmberNodeId emberGetParentId (void )
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Return the parent's node ID.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- The parent's node ID.

emberGetVersionInfo

> EmberStatus emberGetVersionInfo (uint16_t * platform_version, uint16_t * connect_stack_version, uint32_t * bootloader_version)

Get the Platform, Stack and bootloader versions all at once. The version format are not all the same. Please refer to the corresponding documentation to handle the information correctly. The Platform version corresponds to the deprecated GSDK version.

Parameters

Type	Direction	Argument Name	Description
uint16_t *	N/A	platform_version	
uint16_t *	N/A	connect_stack_version	
uint32_t *	N/A	bootloader_version	

Returns

- EMBER_SUCCESS if successful

Copyright © 2025 Silicon Laboratories. All rights reserved.

Stack Counters

Stack Counters

Stack counters API.

Functions

EmberStatus emberGetCounter(EmberCounterType counterType, uint32_t *count)
Retrieve the stack counter corresponding to the passed counter type.

Function Documentation

emberGetCounter

> EmberStatus emberGetCounter (EmberCounterType counterType, uint32_t * count)

Retrieve the stack counter corresponding to the passed counter type.

Parameters

Type	Direction	Argument Name	Description
EmberCounterType	[in]	counterType	An EmberCounterType value indicating the stack counter to be retrieved.
uint32_t *	[out]	count	The counter of the requested `counterType` is returned here

Returns

- An EmberStatus value of EMBER_SUCCESS if the stack counter was successfully retrieved. An EmberStatus value of EMBER_INVALID_CALL if the passed counterType is invalid. An EmberStatus value of EMBER_LIBRARY_NOT_PRESENT if the stack counter library is not present.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Network Management

Network Management

Connect API for finding, forming, joining, and leaving Connect networks.

See network-management.h for source code.

Modules

Frequency Hopping

Parent Support

Handlers

void emberIncomingBeaconHandler(EmberPanId panId, EmberMacAddress *source, int8_t rssi, bool permitJoining, uint8_t beaconFieldsLength, uint8_t *beaconFields, uint8_t beaconPayloadLength, uint8_t *beaconPayload)
Invoked if a beacon is received during the scanning procedure if the handler was initiated by the application with the emberStartActiveScan() stack APIs.

void emberActiveScanCompleteHandler(void)
Invoked after the application calls the emberStartActiveScan() stack API to inform the application that the scanning procedure is complete.

void emberEnergyScanCompleteHandler(int8_t mean, int8_t min, int8_t max, uint16_t variance)
Invoked after the application calls the emberStartEnergyScan() stack API to inform the application that the energy scan procedure is complete and to provide statistics.

Functions

EmberStatus emberInit(void)
Initialize the radio and the Ember stack.

void emberTick(void)
A periodic tick routine that should be called in the main loop in the application.

EmberStatus emberNetworkInit(void)
Resume the network operation after a reboot.

EmberStatus emberStartActiveScan(uint16_t channel)
Start an active scan. EMBER_SUCCESS signals that the scan successfully started. Upon receiving a beacon, the emberIncomingBeaconHandler() stack handler is called. At the end of the scanning procedure, the emberActiveScanCompleteHandler() stack handler is called. Note that, while a scan can be initiated when the node is currently joined to a network, the node will generally be unable to communicate with its PAN during the scan period. In particular, time-sensitive network operations might be affected because a scan operation will prevent any network operation for the duration of the scan.

EmberStatus emberSetActiveScanDuration(uint16_t durationMs)
Set the time in milliseconds the node will spend listening for incoming beacons during an active scan. The default value is set based on the symbol time of the current PHY configuration according to the 802.15.4 specs.

uint16_t emberGetActiveScanDuration(void)
Get the current active scan duration in milliseconds.

Copyright © 2025 Silicon Laboratories. All rights reserved.

EmberStatus emberStartEnergyScan(uint16_t channel, uint8_t samples)
Start an energy scan. EMBER_SUCCESS signals that the scan successfully started. At the end of the scanning procedure, the emberEnergyScanCompleteHandler() stack handler is called. Note that, while a scan can be initiated when the node is currently joined to a network, the node is generally unable to communicate with its PAN during the scan period. In particular, time-sensitive network operations might be affected because a scan operation will prevent any network operation for the duration of the scan.

EmberStatus emberSetApplicationBeaconPayload(uint8_t payloadLength, uint8_t *payload)
Allow the application to set the application portion of the beacon payload. It's by default set to the empty string.

EmberStatus emberJoinNetworkExtended(EmberNodeType nodeType, EmberNodeId nodeId, EmberNetworkParameters *parameters)
Cause the stack to associate with the network using the specified network parameters. It can take several seconds for the stack to associate with the local network. Do not send messages until a call to the emberStackStatusHandler() callback informs you that the stack is up. Notice that forming a network causes the node's security frame counter to be reset.

EmberStatus emberJoinNetwork(EmberNodeType nodeType, EmberNetworkParameters *parameters)
Cause the stack to associate with the network using the specified network parameters. The network ID is assigned by the network coordinator. It can take several seconds for the stack to associate with the local network. Do not send messages until a call to the emberStackStatusHandler() callback informs you that the stack is up. Notice that joining a network causes the node's security frame counter to be reset.

EmberStatus emberPermitJoining(uint8_t duration)
Tell the stack to allow other nodes to join the network with this node as their parent. Joining is initially disabled by default. This function may only be called after the node is part of a network and the stack is up.

EmberStatus emberJoinCommissioned(EmberNodeType nodeType, EmberNodeId nodeId, EmberNetworkParameters *parameters)
Cause the stack to go up with the passed network parameters without performing any over-the-air message exchange. Notice that commissioning a network causes the node's security frame counter to be reset.

EmberStatus emberSetSelectiveJoinPayload(uint8_t payloadLength, uint8_t *payload)
When invoked at a EMBER_STAR_COORDINATOR or a EMBER_STAR_RANGE_EXTENDER, it causes the stack to only accept subsequent joining nodes with matching joining payload. When invoked at a node that has not yet joined a network, it sets the joining payload that will be included in the joining process. Notice, the join payload is included in a non-standard 802.15.4 command, therefore this feature is not available for nodes operating as EMBER_MAC_MODE_DEVICE or EMBER_MAC_MODE_SLEEPY_DEVICE.

EmberStatus emberClearSelectiveJoinPayload(void)
Clear the join payload previously set with the emberSetSelectiveJoinPayload() API. When invoked at an EMBER_STAR_COORDINATOR or an EMBER_STAR_RANGE_EXTENDER it causes the stack to accept joining nodes with any join payload pattern. When invoked at a node that has not yet joined a network, it clears the join payload. Subsequent joining attempts will not include any join payload in the over-the-air joining handshake.

EmberStatus emberSetAuxiliaryAddressFilteringEntry(EmberNodeId nodeId, uint8_t entryIndex)
Set an entry in the auxiliary address filtering table at a given address. Nodes of EMBER_DIRECT_DEVICE device type can receive incoming messages destined to any of the node IDs in the auxiliary address filtering table (while also receiving messages destined to actual node ID). If the passed node ID is EMBER_NULL_NODE_ID, the entry is cleared.

EmberNodeId emberGetAuxiliaryAddressFilteringEntry(uint8_t entryIndex)
Retrieve the content of the auxiliary address filtering table at a given address. See emberSetAuxiliaryAddressFilteringEntry() for details.

void emberResetNetworkState(void)
Forget the current network and reverts to a network status of EMBER_NO_NETWORK.

EmberStatus emberMacFormNetwork(EmberNetworkParameters *parameters)
Form a new network as an EMBER_MAC_MODE_DEVICE by becoming the coordinator. This API should be used to form a compliant 802.15.4 PAN and to inter-operate with other 802.15.4 devices. Notice that forming a network causes the node's security frame counter to be reset.

Copyright © 2025 Silicon Laboratories. All rights reserved.

EmberStatus emberMacSetPanCoordinator(bool isCoordinator)
Configure a EMBER_MAC_MODE_DEVICE node to be a PAN coordinator. Note, this only applies to nodes that have been commissioned as EMBER_MAC_MODE_DEVICE.

EmberStatus emberNetworkLeave(void)
Allow a star topology node that previously joined a network to leave the network. The node will notify the parent node and eventually leave the network. The application is notified that the leave procedure completed via the emberStackStatusHandler() handler.

EmberStatus emberMacAddShortToLongAddressMapping(EmberNodeId shortId, EmberEUI64 longId)
Populate the short-to-long address mapping table at the MAC layer. The table is meaningful only when running as EMBER_MAC_MODE_DEVICE or EMBER_MAC_MODE_SLEEPY_DEVICE. The standard 802.15.4 encryption and authentication process requires the security nonce to be populated with the source node long ID. A receiver must do the same to decrypt a secured incoming message. This short-to-long mapping table is used to decrypt a secured incoming packet from a node using short source addressing. If no entry is found in this table, the incoming message will be dropped. This table is also used to encrypt secured outgoing messages with short source addressing in case the node is sending out a secured message with a short source address other than its own.

EmberStatus emberMacClearShortToLongAddressMappings(void)
Clear the short-to-long address mapping table at the MAC layer.

EmberStatus emberOfdmSetMcs(uint8_t mcs)
Set the MCS in case of an OFDM PHY. MCS can range from 0 to 7.

EmberStatus emberOfdmGetMcs(uint8_t *mcs)
Get the MCS in case of an OFDM PHY.

Macros

#define EMBER_MAC_MAX_APP_BEACON_PAYLOAD_LENGTH 16
The maximum length in bytes of the application beacon payload.

#define EMBER_MAC_STACK_BEACON_PAYLOAD "silabs-connect"

#define EMBER_MAC_STACK_BEACON_PAYLOAD_LENGTH 14
The length in bytes of the stack beacon payload.

#define EMBER_MAC_MAX_BEACON_FIELDS_LENGTH 84
The maximum length in bytes of the beacon fields (superframe, GTS, pending address) as per 802.15.4 specs.

#define EMBER_CHILD_TABLE_AGING_DISABLED 0x20C400
A special timeout value that disables aging of the child table.

#define EMBER_CHILD_TABLE_MAX_TIMEOUT_S (EMBER_CHILD_TABLE_AGING_DISABLED - 1)
The maximum timeout in seconds after which a stale entry may be removed from the child table.

#define EMBER_MAX_SELECTIVE_JOIN_PAYLOAD_LENGTH 50
The maximum length in bytes of the join payload.

#define EMBER_MAX_AUXILIARY_ADDRESS_FILTERING_TABLE_LENGTH 2
The maximum number of entries the auxiliary address filtering table can hold.

Handlers Documentation

emberIncomingBeaconHandler

> void emberIncomingBeaconHandler (EmberPanId panId, EmberMacAddress * source, int8_t rssi, bool permitJoining, uint8_t beaconFieldsLength, uint8_t * beaconFields, uint8_t beaconPayloadLength, uint8_t * beaconPayload)

Copyright © 2025 Silicon Laboratories. All rights reserved.

Invoked if a beacon is received during the scanning procedure if the handler was initiated by the application with the emberStartActiveScan() stack APIs.

Parameters

Type	Direction	Argument Name	Description
EmberPanId	[in]	panId	The source pan ID of the received beacon.
EmberMacAddress *	[in]	source	The source node address of the received beacon.
int8_t	[in]	rssi	The RSSI the beacon was received with.
bool	[in]	permitJoining	The permit joining flag in the received beacon.
uint8_t	[in]	beaconFieldsLength	The length in bytes of the beacon fields defined as per 802.15.4 specs (superframe, GTS fields and pending address fields) of the received beacon.
uint8_t *	[in]	beaconFields	A pointer to the beacon fields defined as per 802.15.4 specs (superframe, GTS fields and pending address fields) of the received beacon.
uint8_t	[in]	beaconPayloadLength	The length in bytes of the application beacon payload of the received beacon.
uint8_t *	[in]	beaconPayload	A pointer to the application beacon payload of the received beacon.

emberActiveScanCompleteHandler

```
void emberActiveScanCompleteHandler (void )
```

Invoked after the application calls the emberStartActiveScan() stack API to inform the application that the scanning procedure is complete.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

emberEnergyScanCompleteHandler

```
void emberEnergyScanCompleteHandler (int8_t mean, int8_t min, int8_t max, uint16_t variance)
```

Invoked after the application calls the emberStartEnergyScan() stack API to inform the application that the energy scan procedure is complete and to provide statistics.

Parameters

Type	Direction	Argument Name	Description
int8_t	[in]	mean	The average energy detected in dBm.
int8_t	[in]	min	The minimum energy detected in dBm.
int8_t	[in]	max	The maximum energy detected in dBm.
uint16_t	[in]	variance	The variance of the energy detected in dBm.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Function Documentation

emberInit

```
EmberStatus emberInit (void )
```

Initialize the radio and the Ember stack.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Device configuration functions must be called before emberInit() is called.

Note

- The application must check the return value of this function. If the initialization fails, normal messaging functions are not available. Some failure modes are not fatal, but the application must follow certain procedures to permit recovery. Ignoring the return code results in unpredictable radio and API behavior. (In particular, problems with association will occur.)

Returns

- An EmberStatus value indicating successful initialization or the reason for failure.

emberTick

```
void emberTick (void )
```

A periodic tick routine that should be called in the main loop in the application.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

emberNetworkInit

```
EmberStatus emberNetworkInit (void )
```

Resume the network operation after a reboot.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

This API must be called on boot prior to ANY network operations. It initializes the networking system and attempts to resume the previous network identity and configuration. If the node was not previously joined, this routine should still be called.

If the node was previously joined to a network, it will retain its original type (e.g., coordinator, router, end device, and so on.)

EMBER_NOT_JOINED is returned if the node is not part of a network.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Returns

- An EmberStatus value that indicates one of the following:
  - successful initialization,
  - EMBER_NOT_JOINED if the node is not part of a network, or
  - the reason for failure.

emberStartActiveScan

```
EmberStatus emberStartActiveScan (uint16_t channel)
```

Start an active scan. EMBER_SUCCESS signals that the scan successfully started. Upon receiving a beacon, the emberIncomingBeaconHandler() stack handler is called. At the end of the scanning procedure, the emberActiveScanCompleteHandler() stack handler is called. Note that, while a scan can be initiated when the node is currently joined to a network, the node will generally be unable to communicate with its PAN during the scan period. In particular, time-sensitive network operations might be affected because a scan operation will prevent any network operation for the duration of the scan.

Parameters

Type	Direction	Argument Name	Description
uint16_t	[in]	channel	The channel to scan.

Possible error responses and their meanings:

- EMBER_INVALID_CALL, the node is currently frequency hopping.
- EMBER_MAC_SCANNING, indicates an ongoing scan.
- EMBER_PHY_INVALID_CHANNEL, the specified channel is not a valid channel on the current platform.

emberSetActiveScanDuration

```
EmberStatus emberSetActiveScanDuration (uint16_t durationMs)
```

Set the time in milliseconds the node will spend listening for incoming beacons during an active scan. The default value is set based on the symbol time of the current PHY configuration according to the 802.15.4 specs.

Parameters

Type	Direction	Argument Name	Description
uint16_t	[in]	durationMs	The active scan duration in milliseconds. A value of 0xFFFF restores the default value.

Returns

- an EmberStatus value of EMBER_SUCCESS if the active scan duration was successfully set, otherwise an EmberStatus value indicating the reason of failure.

emberGetActiveScanDuration

```
uint16_t emberGetActiveScanDuration (void )
```

Get the current active scan duration in milliseconds.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- a 16-bit integer indicating the current duration in millisecond of the active scan.

emberStartEnergyScan

> EmberStatus emberStartEnergyScan (uint16_t channel, uint8_t samples)

Start an energy scan. EMBER_SUCCESS signals that the scan successfully started. At the end of the scanning procedure, the emberEnergyScanCompleteHandler() stack handler is called. Note that, while a scan can be initiated when the node is currently joined to a network, the node is generally unable to communicate with its PAN during the scan period. In particular, time-sensitive network operations might be affected because a scan operation will prevent any network operation for the duration of the scan.

Parameters

Type	Direction	Argument Name	Description
uint16_t	[in]	channel	The channel to scan.
uint8_t	[in]	samples	The number of energy samples to be produced. Each sample is performed averaging the detected energy over X symbols time, whereas X depends on the selected PHY configuration and set by default to 8. The symbol time duration also depends on the selected PHY configuration.

Possible error responses and their meanings:

- EMBER_INVALID_CALL, the node is currently frequency hopping.
- EMBER_BAD_ARGUMENT, the samples parameter is invalid.
- EMBER_MAC_SCANNING, indicates an ongoing scan.
- EMBER_PHY_INVALID_CHANNEL, the specified channel is not a valid channel on the current platform.
- EMBER_NO_BUFFERS, the stack doesn't have enough memory at the moment to perform the requested scan.

emberSetApplicationBeaconPayload

> EmberStatus emberSetApplicationBeaconPayload (uint8_t payloadLength, uint8_t * payload)

Allow the application to set the application portion of the beacon payload. It's by default set to the empty string.

Parameters

Type	Direction	Argument Name	Description
uint8_t	[in]	payloadLength	The length in bytes of the application beacon payload to be set. This value can not exceed EMBER_MAC_MAX_APP_BEACON_PAYLOAD_LENGTH.
uint8_t *	[out]	payload	A pointer to the application beacon payload to be set.

Returns

- an EmberStatus value of EMBER_SUCCESS if the application beacon payload was successfully set, otherwise an EmberStatus value indicating the reason of failure.

Copyright © 2025 Silicon Laboratories. All rights reserved.

emberJoinNetworkExtended

> EmberStatus emberJoinNetworkExtended (EmberNodeType nodeType, EmberNodeId nodeId, EmberNetworkParameters * parameters)

Cause the stack to associate with the network using the specified network parameters. It can take several seconds for the stack to associate with the local network. Do not send messages until a call to the emberStackStatusHandler() callback informs you that the stack is up. Notice that forming a network causes the node's security frame counter to be reset.

Parameters

Type	Direction	Argument Name	Description
EmberNodeType	[in]	nodeType	Specification of the role that this node will have in the network. This role can be EMBER_STAR_RANGE_EXTENDER, EMBER_STAR_END_DEVICE, EMBER_STAR_SLEEPY_END_DEVICE, EMBER_MAC_MODE_DEVICE or EMBER_MAC_MODE_SLEEPY_DEVICE. If the node is frequency hopping, the role can not be EMBER_STAR_RANGE_EXTENDER.
EmberNodeId	[in]	nodeId	An EmberNodeId value indicating the short ID the node intends to use for addressing purposes. If this value is EMBER_NULL_NODE_ID, the network coordinator will allocate a new short address. Addresses should be allocated by the coordinator unless there is a specific need to join a network with a specific ID. If a specific ID is used, uniqueness should be guaranteed across the entire network by the application, via some out of band means. Notice that nodes of EMBER_MAC_MODE_DEVICE or EMBER_MAC_MODE_SLEEPY_DEVICE require this parameter to be set to EMBER_NULL_NODE_ID.
EmberNetworkParameters *	[in]	parameters	An EmberNetworkParameters value that specifies the network parameters of the network with which the node should associate.

Returns

- An EmberStatus value that indicates either:
  - that the association process began successfully or
  - the reason for failure.

emberJoinNetwork

> EmberStatus emberJoinNetwork (EmberNodeType nodeType, EmberNetworkParameters * parameters)

Cause the stack to associate with the network using the specified network parameters. The network ID is assigned by the network coordinator. It can take several seconds for the stack to associate with the local network. Do not send messages until a call to the emberStackStatusHandler() callback informs you that the stack is up. Notice that joining a network causes the node's security frame counter to be reset.

Parameters

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
EmberNodeType	[in]	nodeType	Specification of the role that this node will have in the network. This role can be EMBER_STAR_RANGE_EXTENDER, EMBER_STAR_END_DEVICE, EMBER_STAR_SLEEPY_END_DEVICE, EMBER_MAC_MODE_DEVICE or EMBER_MAC_MODE_SLEEPY_DEVICE. If the node is frequency hopping, the role can not be EMBER_STAR_RANGE_EXTENDER.
EmberNetworkParameters *	[in]	parameters	An EmberNetworkParameters value that specifies the network parameters of the network with which the node should associate.

Returns

- An EmberStatus value that indicates either:
  - that the association process began successfully or
  - the reason for failure.

emberPermitJoining

> EmberStatus emberPermitJoining (uint8_t duration)

Tell the stack to allow other nodes to join the network with this node as their parent. Joining is initially disabled by default. This function may only be called after the node is part of a network and the stack is up.

Parameters

Type	Direction	Argument Name	Description
uint8_t	[in]	duration	A value of 0x00 disables joining. A value of 0xFF enables joining indefinitely. Any other value enables joining for that number of seconds.

Returns

- an EmberStatus value of EMBER_SUCCESS if the permit joining was successfully set, otherwise an EmberStatus value indicating the reason of failure.

emberJoinCommissioned

> EmberStatus emberJoinCommissioned (EmberNodeType nodeType, EmberNodeId nodeId, EmberNetworkParameters * parameters)

Cause the stack to go up with the passed network parameters without performing any over-the-air message exchange. Notice that commissioning a network causes the node's security frame counter to be reset.

Parameters

Type	Direction	Argument Name	Description
EmberNodeType	[in]	nodeType	Specifies the role that this node will have in the network. The only device types allowed in the commissioning API are EMBER_DIRECT_DEVICE, EMBER_MAC_MODE_DEVICE and EMBER_MAC_MODE_SLEEPY_DEVICE.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
EmberNodeId	[in]	nodeId	An EmberNodeId value that specifies the short ID the node will have. The passed node ID must be a valid short address (any value other than EMBER_NULL_NODE_ID or EMBER_BROADCAST_ADDRESS). A value of EMBER_USE_LONG_ADDRESS is allowed only when commissioning the node as EMBER_MAC_MODE_DEVICE or EMBER_MAC_MODE_SLEEPY_DEVICE and will cause the node to send MAC level control messages such as data polls or beacons using long source addressing.
EmberNetworkParameters *	[in]	parameters	An EmberNetworkParameters value that specifies the network parameters of the network the node should participate in.

Returns

- An EmberStatus value that indicates either:
  - that the node successfully commissioned the passed network parameters
  - the reason for failure.

emberSetSelectiveJoinPayload

```
EmberStatus emberSetSelectiveJoinPayload (uint8_t payloadLength, uint8_t * payload)
```

When invoked at a EMBER_STAR_COORDINATOR or a EMBER_STAR_RANGE_EXTENDER, it causes the stack to only accept subsequent joining nodes with matching joining payload. When invoked at a node that has not yet joined a network, it sets the joining payload that will be included in the joining process. Notice, the join payload is included in a non-standard 802.15.4 command, therefore this feature is not available for nodes operating as EMBER_MAC_MODE_DEVICE or EMBER_MAC_MODE_SLEEPY_DEVICE.

Parameters

Type	Direction	Argument Name	Description
uint8_t	[in]	payloadLength	The length in bytes of the passed joining payload. This can not exceed EMBER_MAX_SELECTIVE_JOIN_PAYLOAD_LENGTH.
uint8_t *	[in]	payload	A pointer to the payload to be set.

Returns

- An EmberStatus value that indicates either:
  - that the node successfully set the join payload.
  - the reason for failure.

emberClearSelectiveJoinPayload

```
EmberStatus emberClearSelectiveJoinPayload (void )
```

Clear the join payload previously set with the emberSetSelectiveJoinPayload() API. When invoked at an EMBER_STAR_COORDINATOR or an EMBER_STAR_RANGE_EXTENDER it causes the stack to accept joining nodes with any join payload pattern. When invoked at a node that has not yet joined a network, it clears the join payload. Subsequent joining attempts will not include any join payload in the over-the-air joining handshake.

Parameters

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
void	N/A		

Returns

- An EmberStatus value that indicates either:
  - that the node successfully cleared the join payload.
  - the reason for failure.

emberSetAuxiliaryAddressFilteringEntry

```
EmberStatus emberSetAuxiliaryAddressFilteringEntry (EmberNodeId nodeId, uint8_t entryIndex)
```

Set an entry in the auxiliary address filtering table at a given address. Nodes of EMBER_DIRECT_DEVICE device type can receive incoming messages destined to any of the node IDs in the auxiliary address filtering table (while also receiving messages destined to actual node ID). If the passed node ID is EMBER_NULL_NODE_ID, the entry is cleared.

Parameters

Type	Direction	Argument Name	Description
EmberNodeId	[in]	nodeId	An EmberNodeId value to be added to the auxiliary address filtering table at the passed entry index.
uint8_t	[in]	entryIndex	The index of the auxiliary address filtering table entry to be set.

Returns

- An EmberStatus value of EMBER_SUCCESS if auxiliary address filtering table entry was successfully set. An EmberStatus value of EMBER_INVALID_CALL if the passed entry index is invalid.

emberGetAuxiliaryAddressFilteringEntry

```
EmberNodeId emberGetAuxiliaryAddressFilteringEntry (uint8_t entryIndex)
```

Retrieve the content of the auxiliary address filtering table at a given address. See emberSetAuxiliaryAddressFilteringEntry() for details.

Parameters

Type	Direction	Argument Name	Description
uint8_t	[in]	entryIndex	The index in the auxiliary address filtering table entry to be retrieved.

Returns

- An EmberNodeId value of EMBER_NULL_NODE_ID if the passed entry index is invalid or if the passed entry index refers to an unused entry. Otherwise, it returns the content of the auxiliary address filtering table entry corresponding to the passed entry index.

emberResetNetworkState

```
void emberResetNetworkState (void )
```

Forget the current network and reverts to a network status of EMBER_NO_NETWORK.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

emberMacFormNetwork

```
EmberStatus emberMacFormNetwork (EmberNetworkParameters * parameters)
```

Form a new network as an EMBER_MAC_MODE_DEVICE by becoming the coordinator. This API should be used to form a compliant 802.15.4 PAN and to inter-operate with other 802.15.4 devices. Notice that forming a network causes the node's security frame counter to be reset.

Parameters

Type	Direction	Argument Name	Description
EmberNetworkParameters *	[in]	parameters	An EmberNetworkParameters value that specifies the network parameters of the network to be formed.

Returns

- An EmberStatus value that indicates either the successful formation of the new network or the reason that the network formation failed.

emberMacSetPanCoordinator

```
EmberStatus emberMacSetPanCoordinator (bool isCoordinator)
```

Configure a EMBER_MAC_MODE_DEVICE node to be a PAN coordinator. Note, this only applies to nodes that have been commissioned as EMBER_MAC_MODE_DEVICE.

Parameters

Type	Direction	Argument Name	Description
bool	[in]	isCoordinator	If set to true, the node will identify itself as the PAN coordinator.

Returns

- An EmberStatus value of EMBER_SUCCESS if the coordinator flag was successfully set, or another EmberStatus value indicating the reason of failure.

emberNetworkLeave

```
EmberStatus emberNetworkLeave (void )
```

Allow a star topology node that previously joined a network to leave the network. The node will notify the parent node and eventually leave the network. The application is notified that the leave procedure completed via the emberStackStatusHandler() handler.

Parameters

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
void	N/A		

Returns

- An EmberStatus value of EMBER_SUCCESS if the node successfully initiated the network leave procedure, or another EmberStatus value indicating the reason of failure.

emberMacAddShortToLongAddressMapping

> EmberStatus emberMacAddShortToLongAddressMapping (EmberNodeId shortId, EmberEUI64 longId)

Populate the short-to-long address mapping table at the MAC layer. The table is meaningful only when running as EMBER_MAC_MODE_DEVICE or EMBER_MAC_MODE_SLEEPY_DEVICE. The standard 802.15.4 encryption and authentication process requires the security nonce to be populated with the source node long ID. A receiver must do the same to decrypt a secured incoming message. This short-to-long mapping table is used to decrypt a secured incoming packet from a node using short source addressing. If no entry is found in this table, the incoming message will be dropped. This table is also used to encrypt secured outgoing messages with short source addressing in case the node is sending out a secured message with a short source address other than its own.

Parameters

Type	Direction	Argument Name	Description
EmberNodeId	[in]	shortId	The short address of the [short, long] entry to be added to the table.
EmberEUI64	[in]	longId	The long address of the [short, long] entry to be added to the table.

Note

- Because the table is stored in RAM, the application should ensure it gets correctly re-populated upon reboot.
- Adding a new entry will cause the removal of existing entries matching the passed short ID or long ID.

Returns

- an EmberStatus value of:
  - EMBER_SUCCESS if the mapping was successfully added to the table.
  - EMBER_INVALID_CALL if the node is not running as EMBER_MAC_MODE_DEVICE or as EMBER_MAC_MODE_SLEEPY_DEVICE.
  - EMBER_TABLE_FULL if the table is currently full.
  - EMBER_NO_BUFFERS if the heap does not currently have enough space for the new entry. The size of the table is controlled by the EMBER_SECURITY_SHORT_TO_LONG_MAPPING_TABLE_SIZE.

emberMacClearShortToLongAddressMappings

> EmberStatus emberMacClearShortToLongAddressMappings (void)

Clear the short-to-long address mapping table at the MAC layer.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- an EmberStatus value of EMBER_SUCCESS if table was cleared, or another EmberStatus value indicating the reason of failure.

Copyright © 2025 Silicon Laboratories. All rights reserved.

emberOfdmSetMcs

```
EmberStatus emberOfdmSetMcs (uint8_t mcs)
```

Set the MCS in case of an OFDM PHY. MCS can range from 0 to 7.

Parameters

Type	Direction	Argument Name	Description
uint8_t	[in]	mcs	The MCS value to set.

Returns

- an EmberStatus value of EMBER_SUCCESS if the MCS is valid, an EmberStatus value of EMBER_INVALID_CALL if the current PHY is not OFDM or an EmberStatus value of EMBER_BAD_ARGUMENT if the MCS value is not valid.

emberOfdmGetMcs

```
EmberStatus emberOfdmGetMcs (uint8_t * mcs)
```

Get the MCS in case of an OFDM PHY.

Parameters

Type	Direction	Argument Name	Description
uint8_t *	[out]	mcs	A pointer to the uint8_t that should hold the current MCS value.

Returns

- an EmberStatus value of EMBER_INVALID_CALL if the current PHY is not OFDM or an EmberStatus value of EMBER_SUCCESS if the current PHY is OFDM.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Frequency Hopping

Frequency Hopping

API and callbacks for frequency hopping configuration.

See network-management.h for source code.

Callbacks

void	emberFrequencyHoppingStartClientCompleteHandler(EmberStatus status) This stack handler is invoked after the application calls the emberFrequencyHoppingStartClient() stack API to inform the application that the synchronization process with the server is complete. See emberFrequencyHoppingStartClient() for details.

Functions

EmberStatus	emberFrequencyHoppingSetChannelMask(uint8_t channelMaskLength, uint8_t *channelMask) Set the channel mask for frequency hopping. This API can only be invoked when the node is not frequency hopping.
EmberStatus	emberFrequencyHoppingStartServer(void) Start the device operating as a frequency hopping server. This API can only be invoked when the node is joined to a network. Notice that the server upon starting hopping shall perform an initial advertisement across the entire channel hopping sequence. This is done to resynchronize clients in case the server was started as result of a reboot.
EmberStatus	emberFrequencyHoppingStartClient(EmberNodeId serverNodeId, EmberPanId serverPanId) Start operating as a frequency hopping client and synchronize with the specified server. This API can be invoked on nodes that are already joined to a network (with the exception of nodes started as EMBER_MAC_MODE_DEVICE or EMBER_MAC_MODE_SLEEPY_DEVICE) and nodes that are not joined to a network yet. If the node is already performing frequency hopping, this API returns EMBER_INVALID_CALL. If this API returns EMBER_SUCCESS, the emberFrequencyHoppingStartClientCompleteHandler() is invoked asynchronously to inform the application whether the node successfully synchronized with the specified server or to inform the application of the reason of failure. After the client is synced to a server, it may seamlessly perform the resynchronization process if needed. Sleepy devices in particular periodically perform the resynchronization process. If the client fails a resynchronization process, it informs the application by invoking the emberStackStatusHandler() handler with EMBER_MAC_SYNC_TIMEOUT status. When this occurs, the client will no longer be synced to the server. The application may elect to attempt a new synchronization process by invoking this API again.
EmberStatus	emberFrequencyHoppingStop(void) Stop frequency hopping. This API can only be invoked when the node is frequency hopping. Applicable for both server and client.
EmberStatus	emberSetRadioPriority(sl_connect_radio_priority_t *priorities) Sets the radio priority levels.
EmberStatus	emberGetRadioPriority(sl_connect_radio_priority_t *priorities) Gets the current radio priority levels.

Callbacks Documentation

emberFrequencyHoppingStartClientCompleteHandler

> void emberFrequencyHoppingStartClientCompleteHandler (EmberStatus status)

Copyright © 2025 Silicon Laboratories. All rights reserved.

This stack handler is invoked after the application calls the emberFrequencyHoppingStartClient() stack API to inform the application that the synchronization process with the server is complete. See emberFrequencyHoppingStartClient() for details.

Parameters

Type	Direction	Argument Name	Description
EmberStatus	[in]	status	An EmberStatus value indicating whether the synchronization process with the server was completed successfully or the reason for failure.

Function Documentation

emberFrequencyHoppingSetChannelMask

> EmberStatus emberFrequencyHoppingSetChannelMask (uint8_t channelMaskLength, uint8_t * channelMask)

Set the channel mask for frequency hopping. This API can only be invoked when the node is not frequency hopping.

Parameters

Type	Direction	Argument Name	Description
uint8_t	[in]	channelMaskLength	Length of the bitmap in bytes
uint8_t *	[in]	channelMask	A pointer to a bitmap representing allowed channels for frequency hopping.

Note

- The application is responsible for applying this setting to both the server and clients.

Note

- The bitmap size needs to be at least (EMBER_FREQUENCY_HOPPING_END_CHANNEL + 8) >> 3 or an error is thrown.
- The bitmap needs to be set again after stopping frequency hopping.

Returns

- An EmberStatus value of EMBER_SUCCESS if the node successfully set the bitmask. An EmberStatus value of EMBER_INVALID_CALL if the node is currently performing frequency hopping. An EmberStatus value of EMBER_BAD_ARGUMENT if the resulting channel list is empty, or if channelMaskLength is shorter than expected.

emberFrequencyHoppingStartServer

> EmberStatus emberFrequencyHoppingStartServer (void)

Start the device operating as a frequency hopping server. This API can only be invoked when the node is joined to a network. Notice that the server upon starting hopping shall perform an initial advertisement across the entire channel hopping sequence. This is done to resynchronize clients in case the server was started as result of a reboot.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

-

Copyright © 2025 Silicon Laboratories. All rights reserved.

An EmberStatus value of EMBER_SUCCESS if the node successfully initiated frequency hopping server operations. An EmberStatus value of EMBER_INVALID_CALL if the node is not currently joined to a network or if the node is already performing frequency hopping.

emberFrequencyHoppingStartClient

```
EmberStatus emberFrequencyHoppingStartClient (EmberNodeId serverNodeId, EmberPanId serverPanId)
```

Start operating as a frequency hopping client and synchronize with the specified server. This API can be invoked on nodes that are already joined to a network (with the exception of nodes started as EMBER_MAC_MODE_DEVICE or EMBER_MAC_MODE_SLEEPY_DEVICE) and nodes that are not joined to a network yet. If the node is already performing frequency hopping, this API returns EMBER_INVALID_CALL. If this API returns EMBER_SUCCESS, the emberFrequencyHoppingStartClientCompleteHandler() is invoked asynchronously to inform the application whether the node successfully synchronized with the specified server or to inform the application of the reason of failure. After the client is synced to a server, it may seamlessly perform the resynchronization process if needed. Sleepy devices in particular periodically perform the resynchronization process. If the client fails a resynchronization process, it informs the application by invoking the emberStackStatusHandler() handler with EMBER_MAC_SYNC_TIMEOUT status. When this occurs, the client will no longer be synced to the server. The application may elect to attempt a new synchronization process by invoking this API again.

Parameters

Type	Direction	Argument Name	Description
EmberNodeId	[in]	serverNodeId	An EmberNodeId value indicating the node ID of the server to synchronize with.
EmberPanId	[in]	serverPanId	An EmberPanId value indicating the PAN ID of the server to synchronize with. Note that this parameter is meaningful only if the node is not currently joined to any network.

Returns

- An EmberStatus value of EMBER_SUCCESS indicating that the node successfully initiated the synchronization process with the server, otherwise an EmberStatus value indicating the reason of failure.

emberFrequencyHoppingStop

```
EmberStatus emberFrequencyHoppingStop (void )
```

Stop frequency hopping. This API can only be invoked when the node is frequency hopping. Applicable for both server and client.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- An EmberStatus value of EMBER_SUCCESS indicating that the node successfully stopped frequency hopping. An EmberStatus value of EMBER_INVALID_CALL if the node is not currently frequency hopping.

emberSetRadioPriority

Copyright © 2025 Silicon Laboratories. All rights reserved.

> EmberStatus emberSetRadioPriority (sl_connect_radio_priority_t * priorities)

Sets the radio priority levels.

Parameters

Type	Direction	Argument Name	Description
sl_connect_radio_priority_t *	[in]	priorities	The priority levels to be set.

This function sets the priority levels for the radio operations.

Returns

- An EmberStatus value indicating the success or failure of the operation.

emberGetRadioPriority

> EmberStatus emberGetRadioPriority (sl_connect_radio_priority_t * priorities)

Gets the current radio priority levels.

Parameters

Type	Direction	Argument Name	Description
sl_connect_radio_priority_t *	[out]	priorities	A pointer to a variable where the current priority levels will be stored.

This function retrieves the current priority levels for the radio operations.

Returns

- An EmberStatus value indicating the success or failure of the operation.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Parent Support

Parent Support

Handlers

void emberChildJoinHandler(EmberNodeType nodeType, EmberNodeId nodeId)
Invoked at coordinator, range extender, or mac mode nodes when a new child has joined the device.

Handlers

The Application Framework implements all handlers, directly calling their associated callbacks. By default, Connect projects declare such callbacks as stubs in flex-callbacks-stubs.c. Hence, to use an enabled Connect feature, applications should replace the stub with their own implementation of the associated callback (typically in flex-callbacks.c). See UG235.04 for more info.

EmberStatus emberPurgeIndirectMessages(void)
Purge all indirect transmissions from the indirect message queue.

EmberStatus emberSetIndirectQueueTimeout(uint32_t timeoutMs)
Set indirect queue timeout value. The indirect queue timeout is set by default to EMBER_INDIRECT_TRANSMISSION_TIMEOUT_MS.

Functions

EmberStatus emberFormNetwork(EmberNetworkParameters *parameters)
Form a new network by becoming the coordinator. This API requires the parent-support library to be present.

EmberStatus emberGetChildFlags(EmberMacAddress *address, EmberChildFlags *flags)
Return an EmberChildFlags bitmask indicating the child flags of the child corresponding to the passed MAC address.

EmberStatus emberGetChildInfo(EmberMacAddress *address, EmberMacAddress *addressResp, EmberChildFlags *flags)
Return info on the child corresponding to the passed MAC address.

EmberStatus emberRemoveChild(EmberMacAddress *address)
Remove the node corresponding to the passed MAC address from the child table.

Handlers Documentation

emberChildJoinHandler

> void emberChildJoinHandler (EmberNodeType nodeType, EmberNodeId nodeId)

Invoked at coordinator, range extender, or mac mode nodes when a new child has joined the device.

Parameters

Type	Direction	Argument Name	Description
EmberNodeType	[in]	nodeType	The role of the joining device (EMBER_STAR_RANGE_EXTENDER, EMBER_STAR_END_DEVICE, EMBER_STAR_SLEEPY_END_DEVICE, EMBER_MAC_MODE_DEVICE or EMBER_MAC_MODE_SLEEPY_DEVICE).

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
EmberNodeId	[in]	nodeId	The node ID of the joining device.

Handlers Documentation

emberPurgeIndirectMessages

> EmberStatus emberPurgeIndirectMessages (void)

Purge all indirect transmissions from the indirect message queue.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- an EmberStatus value of EMBER_SUCCESS if all indirect messages were purged, or another EmberStatus value indicating the reason of failure.

emberSetIndirectQueueTimeout

> EmberStatus emberSetIndirectQueueTimeout (uint32_t timeoutMs)

Set indirect queue timeout value. The indirect queue timeout is set by default to EMBER_INDIRECT_TRANSMISSION_TIMEOUT_MS.

Parameters

Type	Direction	Argument Name	Description
uint32_t	N/A	timeoutMs	The timeout in milliseconds to be set.

Returns

- an EmberStatus value of EMBER_SUCCESS if the passed timeout was successfully set, or a value of EMBER_BAD_ARGUMENT if the passed value is invalid.

Function Documentation

emberFormNetwork

> EmberStatus emberFormNetwork (EmberNetworkParameters * parameters)

Form a new network by becoming the coordinator. This API requires the parent-support library to be present.

Parameters

Type	Direction	Argument Name	Description
EmberNetworkParameters *	[in]	parameters	An EmberNetworkParameters value that specifies the network parameters of the network to be formed.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Returns

- An EmberStatus value that indicates either the successful formation of the new network or an EmberStatus value indicating the reason of failure.

emberGetChildFlags

```
EmberStatus emberGetChildFlags (EmberMacAddress * address, EmberChildFlags * flags)
```

Return an EmberChildFlags bitmask indicating the child flags of the child corresponding to the passed MAC address.

Parameters

Type	Direction	Argument Name	Description
EmberMacAddress *	[in]	address	A pointer to an EmberMacAddress that specifies the MAC address of the child.
EmberChildFlags *	[out]	flags	A pointer to an EmberChildFlags containing the child flags of the child corresponding to the passed MAC address.

Note

- Deprecated, use emberGetChildInfo instead

Returns

- An EmberStatus value of EMBER_SUCCESS if the child was found in the child table, or another EmberStatus value indicating the reason of failure.

emberGetChildInfo

```
EmberStatus emberGetChildInfo (EmberMacAddress * address, EmberMacAddress * addressResp, EmberChildFlags * flags)
```

Return info on the child corresponding to the passed MAC address.

Parameters

Type	Direction	Argument Name	Description
EmberMacAddress *	[in]	address	A pointer to an EmberMacAddress that specifies the MAC address of the child (short or long).
EmberMacAddress *	[out]	addressResp	A pointer to an EmberMacAddress that returns the other address (respectively long or short).
EmberChildFlags *	[out]	flags	A pointer to an EmberChildFlags containing the child flags of the child corresponding to the passed MAC address.

Note

- For star coordinators, if the input address is short, the corresponding child will also be searched in the list of devices connected through range extender. Long address and additional flags for these devices are not available to the coordinator.

Note

- Both out parameters are optional. If set to NULL, the API will at least indicate if the child was found in the network.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Returns

- An EmberStatus value of EMBER_SUCCESS if the child was found in the child table, EMBER_CHILD_NOT_FOUND if it was not, or another EmberStatus value indicating the reason of failure.

emberRemoveChild

> EmberStatus emberRemoveChild (EmberMacAddress * address)

Remove the node corresponding to the passed MAC address from the child table.

Parameters

Type	Direction	Argument Name	Description
EmberMacAddress *	[in]	address	A pointer to an EmberMacAddress that specifies the MAC address of the child to be removed.

Returns

- An EmberStatus value of EMBER_SUCCESS if the node was successfully removed from the child table, or another EmberStatus value indicating the reason of failure.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Radio Stream

Radio Stream

Radio stream API.

License

Copyright 2018 Silicon Laboratories Inc. www.silabs.com

SPDX-License-Identifier: Zlib

The licensor of this software is Silicon Laboratories Inc.

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

Connect API managing radio stream for RF testing purpose

See radio-stream.h for source code.

Functions

EmberStatus	emberStartTxStream(EmberTxStreamParameters parameters, uint16_t channel) Start a continuous TX stream to test RF.
EmberStatus	emberStopTxStream(void) Stop an RF stream in progress.

Function Documentation

emberStartTxStream

> EmberStatus emberStartTxStream (EmberTxStreamParameters parameters, uint16_t channel)

Start a continuous TX stream to test RF.

Parameters

Type	Direction	Argument Name	Description
EmberTxStreamParameters	[in]	parameters	Stream mode. See EmberTxStreamParameters.
uint16_t	[in]	channel	RF channel.

Returns

- EMBER_INVALID_CALL if the stack can not process the request
  EMBER_BAD_ARGUMENT if the parameters are wrong

Copyright © 2025 Silicon Laboratories. All rights reserved.

EMBER_SUCCESS if the stream can be started

emberStopTxStream

> EmberStatus emberStopTxStream (void)

Stop an RF stream in progress.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- EMBER_INVALID_CALL if no stream is in progress
  EMBER_SUCCESS otherwise

Copyright © 2025 Silicon Laboratories. All rights reserved.

Configuration

Configuration

User-configurable stack configuration macros and defaults.

Connect stack provides various interfaces to apply configuration:

- APIs to change configuration run-time
- Manufacturing tokens to change configuration during device flashing
- Macros to change configuration at compile-time

The compile time configuration macros and their default values are listed here.

The default values are always chosen to make it usable for most applications, but in some cases, you might need to change it. The recommended way is to either define the macro you need to change in the compilation command (e.g. gcc -D) or you by manipulating the configuration options in the configurator GUI.

See ember-configuration-defaults.h for source code.

Macros

#define EMBER_HEAP_SIZE 2000
The size in bytes of the Ember heap. See Memory Buffer for more details.

#define EMBER_CHILD_TABLE_SIZE 0
The maximum number of children supported by the device. Can be configured from 0 to 64. 11B of token space is allocated for each child in the table.

#define EMBER_CHILD_TABLE_TOKEN_SIZE EMBER_CHILD_TABLE_SIZE

#define EMBER_CHILD_TIMEOUT_SEC 3600
Every child should exchange regularly some sort of traffic with the parent. Eventually, if traffic is not exchanged for a prolonged period of time, the parent may remove the child from the child table. In particular the parent shall remove the oldest stale child whenever the child table is full and there is the need of making room for a new child. Range extenders periodically exchange network-level commands with the coordinator. End devices and sleepy end devices can use emberPollForData() as keep alive mechanism, or use the Poll Plugin plugin. The maximum allowed timeout value is EMBER_CHILD_TABLE_MAX_TIMEOUT_S. Setting the timeout value to EMBER_CHILD_TABLE_AGING_DISABLED disables aging of the child table.

#define EMBER_INDIRECT_QUEUE_SIZE 0
Indirect queue is used on a parent to store a message intended for a sleepy end device, this configures the size of that queue. Configure it to 0 if parent support plugin is not used.

#define EMBER_MAC_OUTGOING_QUEUE_SIZE 0
MAC Outgoing paclet queue is to store messages until the radio is available to send it (In most cases, the radio is unavailable because it's already transmitting). The configures the size of that queue.

#define EMBER_INDIRECT_TRANSMISSION_TIMEOUT_MS 8000
The maximum amount of time (in milliseconds) that the MAC will hold a message for indirect transmission to a child. The maximum value is 30 seconds (30000 milliseconds).

#define EMBER_NWK_RANGE_EXTENDER_UPDATE_PERIOD_SEC 60
The period in seconds a range extender sends an update command to the coordinator containing the list of its children.

Copyright © 2025 Silicon Laboratories. All rights reserved.

#define EMBER_MAC_ACK_TIMEOUT_MS 25
The ACK timeout in milliseconds. This parameter should be fine-tuned to reduce energy consumption for sleepy devices and depends on the data rate of the PHY configuration used. The maximum allowed value is 65.

#define EMBER_RADIO_CCA_THRESHOLD -65
The CCA threshold used at the MAC layer for CSMA/CA, in dBm.

#define EMBER_FREQUENCY_HOPPING_SEED 0
The frequency hopping channel sequence generation seed. Can be configured between 0 and 65535. See the Frequency Hopping chapter of UG235.03 for more details.

#define EMBER_FREQUENCY_HOPPING_START_CHANNEL 0
The lowest channel on the frequency hopping list. See the Frequency Hopping chapter of UG235.03 for more details.

#define EMBER_FREQUENCY_HOPPING_END_CHANNEL 24
The highest channel on the frequency hopping list. See the Frequency Hopping chapter of UG235.03 for more details.

#define EMBER_FREQUENCY_HOPPING_CHANNEL_DURATION_MS 400
The time in milliseconds to stay on each channel for frequency hopping. See the Frequency Hopping chapter of UG235.03 for more details.

#define EMBER_FREQUENCY_HOPPING_CHANNEL_GUARD_DURATION_MS 20
The time in milliseconds to guard each channel while frequency hopping. No MAC activity is allowed when entering or exiting the slot. See the Frequency Hopping chapter of UG235.03 for more details.

#define EMBER_FREQUENCY_HOPPING_SERVER_FREQ_INFO_BROADCAST_PERIOD_S 15
The duration in seconds after which the server should broadcast its frequency hopping information to allow clients to realign. See the Frequency Hopping chapter of UG235.03 for more details.

#define EMBER_FREQUENCY_HOPPING_SLEEPY_CLIENT_RESYNC_PERIOD_S 60
The duration in seconds after which a sleepy client should resync with the server if the last resync happened more than this duration ago. See the Frequency Hopping chapter of UG235.03 for more details.

#define EMBER_FREQUENCY_HOPPING_ALWAYS_ON_CLIENT_SYNC_TIMEOUT_S (100)
The maximum duration in seconds a non sleepy client would keep hopping without receiving frequency hopping information from the server, after which the synchronization with the server is deemed lost. A special value of EMBER_FREQUENCY_HOPPING_ALWAYS_ON_CLIENT_SYNC_DISABLE_TIMEOUT disables this timeout.

#define EMBER_FREQUENCY_HOPPING_SERVER_ADVERTISING_ITERATION_COUNT (3)
When a node is started a frequency hopping server, it will first advertise on all channels to resynchronize all existing clients in case the server was started as result of a reboot. This parameter defines the number of iterations over the entire hopping sequence.

#define EMBER_COORDINATOR_FIRST_SHORT_ID_TO_BE_ASSIGNED 1
An EMBER_STAR_COORDINATOR assigns short IDs to other nodes in the star network sequentially starting from this short ID. This option provides a simple effective way to reserve an pool of short addresses for commissioning.

#define EMBER_SECURITY_SHORT_TO_LONG_MAPPING_TABLE_SIZE 10
The size of the short-to-long address mapping table. See emberMacAddShortToLongAddressMapping for more details.

#define EMBER_DEFAULT_TX_POWER_DECI_DBM 0
The default TX power that the radio should use. If no TX power has been specified by emberSetRadioPower, the stack will use this value.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Status Codes

Status Codes

Return-code definitions for Connect stack API functions.

Many Connect API functions return an EmberStatus value to indicate the success or failure of the call.

Return codes are one byte long.

This page documents the possible status codes and their meanings.

See error-def.h for source code.

See also error.h for information on how the values for the return codes are built up from these definitions. The file error-def.h is separated from error.h because utilities will use this file to parse the return codes.

Note

- Do not include error-def.h directly. It is included by error.h inside an enum typedef, which is in turn included by ember.h.

Enumerations

Copyright © 2025 Silicon Laboratories. All rights reserved.

enum EmberStatus {

EMBER_SUCCESS = 0x00
EMBER_ERR_FATAL = 0x01
EMBER_BAD_ARGUMENT = 0x02
EMBER_EEPROM_MFG_STACK_VERSION_MISMATCH = 0x04
EMBER_INVALID_CALL = 0x70
EMBER_EEPROM_MFG_VERSION_MISMATCH = 0x06
EMBER_EEPROM_STACK_VERSION_MISMATCH = 0x07
EMBER_NO_BUFFERS = 0x18
EMBER_SERIAL_INVALID_BAUD_RATE = 0x20
EMBER_SERIAL_INVALID_PORT = 0x21
EMBER_SERIAL_TX_OVERFLOW = 0x22
EMBER_SERIAL_RX_OVERFLOW = 0x23
EMBER_SERIAL_RX_FRAME_ERROR = 0x24
EMBER_SERIAL_RX_PARITY_ERROR = 0x25
EMBER_SERIAL_RX_EMPTY = 0x26
EMBER_SERIAL_RX_OVERRUN_ERROR = 0x27
EMBER_MAC_NO_DATA = 0x31
EMBER_MAC_SYNC_TIMEOUT = 0x33
EMBER_MAC_SYNC_WRONG_SEED = 0x34
EMBER_MAC_SECURITY_FAILED = 0x35
EMBER_MAC_UNKNOWN_DESTINATION = 0x37
EMBER_MAC_SECURITY_NOT_SUPPORTED = 0x38
EMBER_MAC_TRANSMIT_QUEUE_FULL = 0x39
EMBER_MAC_ACK_HEADER_TYPE = 0x3B
EMBER_MAC_SCANNING = 0x3D
EMBER_MAC_BUSY = 0x3E
EMBER_MAC_NO_ACK_RECEIVED = 0x40
EMBER_MAC_INDIRECT_TIMEOUT = 0x41
EMBER_MAC_INDIRECT_MESSAGE_PURGED = 0x42
EMBER_ERR_FLASH_PROG_FAIL = 0x4B
EMBER_MESSAGE_TOO_LONG = 0x74
EMBER_ADC_CONVERSION_DONE = 0x80
EMBER_ADC_CONVERSION_BUSY = 0x81
EMBER_ADC_CONVERSION_DEFERRED = 0x82
EMBER_ADC_NO_CONVERSION_PENDING = 0x84
EMBER_SLEEP_INTERRUPTED = 0x85
EMBER_PHY_TX_INCOMPLETE = 0x89
EMBER_PHY_INVALID_CHANNEL = 0x8A
EMBER_PHY_INVALID_POWER = 0x8B
EMBER_PHY_TX_BUSY = 0x8C
EMBER_PHY_TX_CCA_FAIL = 0x8D
EMBER_PHY_CALIBRATING = 0x8E
EMBER_PHY_ACK_RECEIVED = 0x8F
EMBER_NETWORK_UP = 0x90
EMBER_NETWORK_DOWN = 0x91
EMBER_JOIN_SCAN_FAILED = 0x92
EMBER_JOIN_FAILED = 0x94
EMBER_JOIN_DENIED = 0x95
EMBER_JOIN_TIMEOUT = 0x96
EMBER_NO_VALID_BEACONS = 0xAB
EMBER_SECURITY_DATA_INVALID = 0xBD
EMBER_NOT_JOINED = 0x93
EMBER_TABLE_FULL = 0xB4
EMBER_LIBRARY_NOT_PRESENT = 0xB5
EMBER_CHILD_NOT_FOUND = 0xB6
EMBER_NVM3_ERR_OPENED_WITH_OTHER_PARAMETERS = 0xC1
EMBER_NVM3_ERR_ALIGNMENT_INVALID = 0xC2
EMBER_NVM3_ERR_SIZE_TOO_SMALL = 0xC3
EMBER_NVM3_ERR_PAGE_SIZE_NOT_SUPPORTED = 0xC4
EMBER_NVM3_ERR_TOKEN_INIT = 0xC5
EMBER_NVM3_ERR_UNKNOWN = 0xC7
EMBER_NCP_UNKNOWN_COMMAND_ID = 0xD0
EMBER_APPLICATION_ERROR_0 = 0xF0
EMBER_APPLICATION_ERROR_1 = 0xF1
EMBER_APPLICATION_ERROR_2 = 0xF2
EMBER_APPLICATION_ERROR_3 = 0xF3
EMBER_APPLICATION_ERROR_4 = 0xF4
EMBER_APPLICATION_ERROR_5 = 0xF5
EMBER_APPLICATION_ERROR_6 = 0xF6

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
    EMBER_APPLICATION_ERROR_7
    = 0xF7
    EMBER_APPLICATION_ERROR_8
    = 0xF8
    EMBER_APPLICATION_ERROR_9
    = 0xF9
    EMBER_APPLICATION_ERROR_10
    = 0xFA
    EMBER_APPLICATION_ERROR_11
    = 0xFB
    EMBER_APPLICATION_ERROR_12
    = 0xFC
    EMBER_APPLICATION_ERROR_13
    = 0xFD
    EMBER_APPLICATION_ERROR_14
    = 0xFE
    EMBER_APPLICATION_ERROR_15
    = 0xFF
}
```

Enumeration Documentation

EmberStatus

> EmberStatus

	Enumerator
EMBER_SUCCESS	The generic "no error" message.
EMBER_ERR_FATAL	The generic "fatal error" message.
EMBER_BAD_ARGUMENT	An invalid value was passed as an argument to a function.
EMBER_EEPROM_MFG_STACK_VERSION_MISMATCH	The manufacturing and stack token format in non-volatile memory is different than what the stack expects (returned at initialization).
EMBER_INVALID_CALL	The API call is not allowed given the current state of the stack.
EMBER_EEPROM_MFG_VERSION_MISMATCH	The manufacturing token format in non-volatile memory is different than what the stack expects (returned at initialization).
EMBER_EEPROM_STACK_VERSION_MISMATCH	The stack token format in non-volatile memory is different than what the stack expects (returned at initialization).
EMBER_NO_BUFFERS	There are no more buffers (either in the stack heap or the queue used by the associated module, such as indirect queue).
EMBER_SERIAL_INVALID_BAUD_RATE	Specified an invalid baud rate.
EMBER_SERIAL_INVALID_PORT	Specified an invalid serial port.
EMBER_SERIAL_TX_OVERFLOW	Tried to send too much data.
EMBER_SERIAL_RX_OVERFLOW	There was not enough space to store a received character and some characters were dropped.
EMBER_SERIAL_RX_FRAME_ERROR	Detected a UART framing error.
EMBER_SERIAL_RX_PARITY_ERROR	Detected a UART parity error.
EMBER_SERIAL_RX_EMPTY	There is no received data to process.
EMBER_SERIAL_RX_OVERRUN_ERROR	The receive interrupt was not handled in time and some characters were dropped.
EMBER_MAC_NO_DATA	No pending data exists for device doing a data poll.

Copyright © 2025 Silicon Laboratories. All rights reserved.

EMBER_MAC_SYNC_TIMEOUT	The frequency hopping client failed the frequency hopping synchronization procedure. It timed out trying to reach the frequency hopping server.
EMBER_MAC_SYNC_WRONG_SEED	The frequency hopping client failed the frequency hopping synchronization procedure. The server is currently using a different seed.
EMBER_MAC_SECURITY_FAILED	MAC security operation failed.
EMBER_MAC_UNKNOWN_DESTINATION	Transmission failed: the destination node does not appear in the neighbor or child tables.
EMBER_MAC_SECURITY_NOT_SUPPORTED	Transmission failed: the local node does not support security or a secured transmission has been requested to a child that does not support security.
EMBER_MAC_TRANSMIT_QUEUE_FULL	The MAC transmit queue is full.
EMBER_MAC_ACK_HEADER_TYPE	MAC ACK header received.
EMBER_MAC_SCANNING	The MAC can't complete this task because it is scanning.
EMBER_MAC_BUSY	The requested operation cannot be completed because MAC is currently busy performing a high-priority task.
EMBER_MAC_NO_ACK_RECEIVED	Expected to receive an ACK following the transmission, but the MAC level ACK was never received.
EMBER_MAC_INDIRECT_TIMEOUT	Indirect data message timed out before polled.
EMBER_MAC_INDIRECT_MESSAGE_PURGED	Transmission failed: the indirect message was purged because the destination child has been removed or updated, or because emberPurgeIndirectMessages() was called.
EMBER_ERR_FLASH_PROG_FAIL	A fatal error has occurred while trying to write data to the flash, possibly due to write protection or an invalid address. The data in the flash cannot be trusted after this error, and it is possible this error is the result of exceeding the life cycles of the flash.
EMBER_MESSAGE_TOO_LONG	The message to be transmitted is too big to fit into a single over-the-air packet.
EMBER_ADC_CONVERSION_DONE	Conversion is complete.
EMBER_ADC_CONVERSION_BUSY	Conversion cannot be done because a request is being processed.
EMBER_ADC_CONVERSION_DEFERRED	Conversion is deferred until the current request has been processed.
EMBER_ADC_NO_CONVERSION_PENDING	No results are pending.
EMBER_SLEEP_INTERRUPTED	Sleeping (for a duration) has been abnormally interrupted and exited prematurely.
EMBER_PHY_TX_INCOMPLETE	The transmit hardware did not finish transmitting a packet.
EMBER_PHY_INVALID_CHANNEL	An unsupported channel setting was specified.
EMBER_PHY_INVALID_POWER	An unsupported power setting was specified.
EMBER_PHY_TX_BUSY	The requested operation cannot be completed because the radio is currently busy, either transmitting a packet or performing calibration.
EMBER_PHY_TX_CCA_FAIL	The transmit attempt failed because all CCA attempts indicated that the channel was busy.
EMBER_PHY_CALIBRATING	The requested operation cannot be completed because the stack has triggered a radio calibration. Wait for about one second for it to complete.
EMBER_PHY_ACK_RECEIVED	The expected ACK was received after the last transmission.
EMBER_NETWORK_UP	The stack software has completed initialization required to be in a network and is ready to send and receive packets over the air.
EMBER_NETWORK_DOWN	The network is not operating.
EMBER_JOIN_SCAN_FAILED	The node failed to initiate the scanning process during the joining process.
EMBER_JOIN_FAILED	An attempt to join a network failed.
EMBER_JOIN_DENIED	An attempt to join a network was rejected.

Copyright © 2025 Silicon Laboratories. All rights reserved.

EMBER_JOIN_TIMEOUT	The node timed out waiting for a response during the joining process.
EMBER_NO_VALID_BEACONS	An attempt to join or rejoin the network failed because no valid beacons was received by the joining node.
EMBER_SECURITY_DATA_INVALID	The security data provided was not valid, or an integrity check failed.
EMBER_NOT_JOINED	The node has not joined a network. Returned by emberNetworkInit() if there was no connection data saved in tokens.
EMBER_TABLE_FULL	There are no empty entries left in the table.
EMBER_LIBRARY_NOT_PRESENT	The requested function cannot be executed because the library (plugin) that contains the necessary functionality is not present.
EMBER_CHILD_NOT_FOUND	The requested NodeId has not been found in the child list or grandchildren list (start network only).
EMBER_NVM3_ERR_OPENED_WITH_OTHER_PARAMETERS	NVM3 is telling the application that the initialization was aborted as the NVM3 instance was already opened with other parameters.
EMBER_NVM3_ERR_ALIGNMENT_INVALID	NVM3 is telling the application that the initialization was aborted as the NVM3 instance is not aligned properly in memory.
EMBER_NVM3_ERR_SIZE_TOO_SMALL	NVM3 is telling the application that the initialization was aborted as the size of the NVM3 instance is too small.
EMBER_NVM3_ERR_PAGE_SIZE_NOT_SUPPORTED	NVM3 is telling the application that the initialization was aborted as the NVM3 page size is not supported.
EMBER_NVM3_ERR_TOKEN_INIT	NVM3 is telling the application that there was an error initializing some of the tokens.
EMBER_NVM3_ERR_UNKNOWN	NVM3 is telling the application that there has been an unknown error.
EMBER_NCP_UNKNOWN_COMMAND_ID	The NCP does not know the command ID that the host sent. This can correspond to a version mismatch, where an API is available on the host but not on the NCP.
EMBER_APPLICATION_ERROR_0	This error is reserved for customer application use. This will never be returned from any portion of the network stack or HAL.
EMBER_APPLICATION_ERROR_1	This error is reserved for customer application use. This will never be returned from any portion of the network stack or HAL.
EMBER_APPLICATION_ERROR_2	This error is reserved for customer application use. This will never be returned from any portion of the network stack or HAL.
EMBER_APPLICATION_ERROR_3	This error is reserved for customer application use. This will never be returned from any portion of the network stack or HAL.
EMBER_APPLICATION_ERROR_4	This error is reserved for customer application use. This will never be returned from any portion of the network stack or HAL.
EMBER_APPLICATION_ERROR_5	This error is reserved for customer application use. This will never be returned from any portion of the network stack or HAL.

Copyright © 2025 Silicon Laboratories. All rights reserved.

EMBER_APPLICATION_ERROR_6	This error is reserved for customer application use. This will never be returned from any portion of the network stack or HAL.
EMBER_APPLICATION_ERROR_7	This error is reserved for customer application use. This will never be returned from any portion of the network stack or HAL.
EMBER_APPLICATION_ERROR_8	This error is reserved for customer application use. This will never be returned from any portion of the network stack or HAL.
EMBER_APPLICATION_ERROR_9	This error is reserved for customer application use. This will never be returned from any portion of the network stack or HAL.
EMBER_APPLICATION_ERROR_10	This error is reserved for customer application use. This will never be returned from any portion of the network stack or HAL.
EMBER_APPLICATION_ERROR_11	This error is reserved for customer application use. This will never be returned from any portion of the network stack or HAL.
EMBER_APPLICATION_ERROR_12	This error is reserved for customer application use. This will never be returned from any portion of the network stack or HAL.
EMBER_APPLICATION_ERROR_13	This error is reserved for customer application use. This will never be returned from any portion of the network stack or HAL.
EMBER_APPLICATION_ERROR_14	This error is reserved for customer application use. This will never be returned from any portion of the network stack or HAL.
EMBER_APPLICATION_ERROR_15	This error is reserved for customer application use. This will never be returned from any portion of the network stack or HAL.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Stack Tokens

Stack Tokens

Definitions for stack tokens.

Stack tokens are used by the stack to store information in non-volatile memory. A typical use case is to store the network information, so after an accidental reset it can be part of the network without going through the association process again.

Note

- For the application tokens, refer to Token Access and AN1154.

Warnings

- While stack tokens can be accessed through the Token Access API, they **must not be written directly**. Most stack tokens have APIs to read/write them. This documentation is intended for those who need more information on the internal details of the connect stack.

See token-stack.h for source code.

Modules

tokTypeStackKey

tokTypeStackNodeData

tokTypeStackChildTableEntry

Token types

The types used for each stack token.

typedef uint16_t **tokTypeStackNvdataVersion**
Type for TOKEN_STACK_NVDATA_VERSION. Keeps the version number of stack tokens.

typedef uint32_t **tokTypeStackNonceCounter**
Type for TOKEN_STACK_NONCE_COUNTER. Used to make sure that Nonce used for security is not repeated even after unexpected reboot.

typedef uint32_t **tokTypeStackKeyID**
Type for TOKEN_STACK_SECURITY_KEY_ID. Used to make sure that Nonce used for security is not repeated even after unexpected reboot.

typedef uint16_t **tokTypeStackLastAllocatedId**
Type for TOKEN_STACK_LAST_ASSIGNED_ID. Stores the last assigned NodeId if the device is EMBER_STAR_COORDINATOR.

typedef uint32_t **tokTypeStackBootCounter**
Type for TOKEN_STACK_BOOT_COUNTER. Increments at boot (during emberInit()).

typedef EmberEUI64 **tokTypeParentLongId**
Type for TOKEN_STACK_PARENT_LONG_ID. Stores the Long Id of the parent of this device. Only used for EMBER_MAC_MODE_DEVICE and EMBER_MAC_MODE_SLEEPY_DEVICE device types.

Macros

Copyright © 2025 Silicon Laboratories. All rights reserved.

#define CURRENT_STACK_TOKEN_VERSION 0×03FC

The current version number of the stack tokens. MSB is the version. LSB is a complement.

Token types Documentation

tokTypeStackNvdataVersion

```
typedef uint16_t tokTypeStackNvdataVersion
```

Type for TOKEN_STACK_NVDATA_VERSION. Keeps the version number of stack tokens.

tokTypeStackNonceCounter

```
typedef uint32_t tokTypeStackNonceCounter
```

Type for TOKEN_STACK_NONCE_COUNTER. Used to make sure that Nonce used for security is not repeated even after unexpected reboot.

tokTypeStackKeyID

```
typedef uint32_t tokTypeStackKeyID
```

Type for TOKEN_STACK_SECURITY_KEY_ID. Used to make sure that Nonce used for security is not repeated even after unexpected reboot.

tokTypeStackLastAllocatedId

```
typedef uint16_t tokTypeStackLastAllocatedId
```

Type for TOKEN_STACK_LAST_ASSIGNED_ID. Stores the last assigned NodeId if the device is EMBER_STAR_COORDINATOR.

tokTypeStackBootCounter

```
typedef uint32_t tokTypeStackBootCounter
```

Type for TOKEN_STACK_BOOT_COUNTER. Increments at boot (during emberInit()).

tokTypeParentLongId

```
typedef EmberEUI64 tokTypeParentLongId
```

Type for TOKEN_STACK_PARENT_LONG_ID. Stores the Long Id of the parent of this device. Only used for EMBER_MAC_MODE_DEVICE and EMBER_MAC_MODE_SLEEPY_DEVICE device types.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Copyright © 2025 Silicon Laboratories. All rights reserved.

tokTypeStackKey

Type for TOKEN_STACK_SECURITY_KEY. Keeps the security key for MAC layer security.

Public Attributes

uint8_t	networkKey

Public Attribute Documentation

networkKey

```
uint8_t tokTypeStackKey::networkKey[16]
```

The key itself

Copyright © 2025 Silicon Laboratories. All rights reserved.

tokTypeStackNodeData

Type for TOKEN_STACK_NODE_DATA. Generic information of the node is stored in this token.

Public Attributes

uint16_t	panId
int16_t	radioTxPower
uint16_t	radioFreqChannel
uint8_t	nodeType
uint16_t	nodeId
uint16_t	parentId

Public Attribute Documentation

panId

```
uint16_t tokTypeStackNodeData::panId
```

The PanId of the device

radioTxPower

```
int16_t tokTypeStackNodeData::radioTxPower
```

The TX power configured for the device in deci-dBm

radioFreqChannel

```
uint16_t tokTypeStackNodeData::radioFreqChannel
```

The radio channel configured for the device

nodeType

```
uint8_t tokTypeStackNodeData::nodeType
```

The EmberNodeType configured for the device

nodeId

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
uint16_t tokTypeStackNodeData::nodeId
```

The NodeId (short address) of the device

parentId

```
uint16_t tokTypeStackNodeData::parentId
```

The NodeId of the device's parent, if any

Copyright © 2025 Silicon Laboratories. All rights reserved.

tokTypeStackChildTableEntry

Type of an element of TOKEN_STACK_CHILD_TABLE (indexed token). Keeps children information of a device, which has parent support enabled.

Public Attributes

EmberEUI64	longId
EmberNodeId	shortId
uint8_t	flags

Public Attribute Documentation

longId

```
EmberEUI64 tokTypeStackChildTableEntry::longId
```

The Long Id of the child

shortId

```
EmberNodeId tokTypeStackChildTableEntry::shortId
```

The NodeId of the child

flags

```
uint8_t tokTypeStackChildTableEntry::flags
```

Flags for the child required by the stack

Copyright © 2025 Silicon Laboratories. All rights reserved.

Event Scheduling

Event Scheduling

Scheduling events for future execution.

See Event Scheduling for documentation. These macros implement an event abstraction that allows the application to schedule code to run after a specified time interval. An event consists of a procedure to be called at some point in the future and a control object that determines which procedure should be called. Events are also useful when an ISR needs to initiate an action that should run outside the ISR context.

See event.h for source code.

Note that, while not required, it is recommended that the event-handling procedure explicitly define the recurrence of the next event, either by rescheduling via some kind of **emberEventControlSetDelayXX()** call or by deactivating via a call to emberEventControlSetInactive().

When the handler does not explicitly reschedule or cancel the event, the default behavior of the event control system is to keep the event immediately active as if the handler function had called emberEventControlSetActive(someEvent) or emberEventControlSetDelayMS(someEvent, 0).

The base time units for events are ticks. A tick equals 1 ms on every platform supported by Connect. Note, however, that the accuracy of the base tick depends on the timer source, which by default is the LF RC oscillator on EFR32 platforms.

Furthermore, the scheduled delay is the minimum delay. If emberRunEvents() or emberRunTask() are not called frequently enough, the actual delay may be longer than the scheduled delay.

Additionally, the APIs for quarter second and minute delays (emberEventControlSetDelayQS() and emberEventControlSetDelayMinutes()) use "binary" units. One quarter second is 256 ticks and one minute is 65536 ticks. These APIs are therefore doesn't actually mean a quarter of second or a minute on platforms supported by Connect.

However, in the future, Connect support might become available on platforms where one tick is not exactly 1 ms. For example, on the EM357 SoC, 1 s is 1024 ticks, so each tick is 1000 / 1024 = ~0.98 milliseconds. If you need platform independent accurate delays, use the macros MILLISECOND_TICKS_PER_SECOND and MILLISECOND_TICKS_PER_MINUTE. For example, calling emberEventControlSetDelayMS(someEvent, 3 * MILLISECOND_TICKS_PER_MINUTE) will delay for 3 minutes on any platform.

The following are brief usage examples.

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
EmberEventControl delayEvent;
EmberEventControl signalEvent;
EmberEventControl periodicEvent;

void delayEventHandler(void)
{
  // Disable this event until its next use.
  emberEventControlSetInactive(delayEvent);
}

void signalEventHandler(void)
{
  // Disable this event until its next use.
  emberEventControlSetInactive(signalEvent);

  // Sometimes an action has to occur 100 ms later.
  if (somethingIsExpected)
    emberEventControlSetDelayMS(delayEvent, 100);
}

void periodicEventHandler(void)
{
  emberEventControlSetDelayQS(periodicEvent, 4);
}

void someIsr(void)
{
  // Set the signal event to run at the first opportunity.
  emberEventControlSetActive(signalEvent);
}

// Put the controls and handlers in an array.  They will be run in
// this order (this is usually generated)
EmberEventData events[] =
{
  { &delayEvent,    delayEventHandler },
  { &signalEvent,   signalEentHandler },
  { &periodicEvent, periodicEventHandler },
  { NULL, NULL }                    // terminator
};

void main(void)
{
  // Cause the periodic event to occur once a second.
  emberEventControlSetDelayQS(periodicEvent, 4);

  while (true) {
    emberRunEvents(events);
  }
}
```

Time Manipulation Macros

void sli_event_control_set_active(EmberEventControl *event)
Set EmberEventControl to run at the next available opportunity.

void emEventControlSetDelayMS(EmberEventControl *event, uint32_t delay)
Set EmberEventControl to run some milliseconds in the future.

uint32_t emEventControlGetRemainingMS(EmberEventControl *event)
Check when the event is scheduled to run.

void emberRunEvents(EmberEventData *events)
Start an event handler if anything is scheduled when this function is called.

Copyright © 2025 Silicon Laboratories. All rights reserved.

void emberRunTask(EmberTaskId taskid)
Start an event handler if there is anything scheduled at the moment this function is called.

uint32_t emberMsToNextEvent(EmberEventData *events, uint32_t maxMs)
Check when the next event is scheduled to run.

uint32_t emberMsToNextEventExtended(EmberEventData *events, uint32_t maxMs, uint8_t *returnIndex)
Check when the next event is scheduled to run.

uint32_t emberMsToNextStackEvent(void)
Check when the next stack event is scheduled to run.

EmberTaskId emberTaskInit(EmberEventData *events)
Initialize a task for managing events and processor idling state.

bool emberMarkTaskIdle(EmberTaskId taskid)
Try to idle the CPU, unless any events in any tasks are pending.

void emTaskEnableIdling(bool allow)
Enable or disable idling.

void emMarkTaskActive(EmberTaskId taskid)
Calling it indicates that a task has something to do, so it should prevent the CPU from idling until emberMarkTaskIdle is next called on this task.

#define elapsedTimeInt8u (oldTime, newTime)
Returns the elapsed time between two 8 bit values. Result may not be valid if the time samples differ by more than 127.

#define elapsedTimeInt16u (oldTime, newTime)
Returns the elapsed time between two 16 bit values. Result may not be valid if the time samples differ by more than 32767.

#define elapsedTimeInt32u (oldTime, newTime)
Returns the elapsed time between two 32 bit values. Result may not be valid if the time samples differ by more than 2147483647.

#define MAX_INT8U_VALUE (0xFF)
Returns true if t1 is greater than t2. Can only account for 1 wrap around of the variable before it is wrong.

#define HALF_MAX_INT8U_VALUE (0×80)
Returns the elapsed time between two 8 bit values. Result may not be valid if the time samples differ by more than 127.

#define timeGTorEqualInt8u (t1, t2)
Returns the elapsed time between two 8 bit values. Result may not be valid if the time samples differ by more than 127.

#define MAX_INT16U_VALUE (0xFFFF)
Returns true if t1 is greater than t2. Can only account for 1 wrap around of the variable before it is wrong.

#define HALF_MAX_INT16U_VALUE (0×8000)
Returns the elapsed time between two 8 bit values. Result may not be valid if the time samples differ by more than 127.

#define timeGTorEqualInt16u (t1, t2)
Returns the elapsed time between two 8 bit values. Result may not be valid if the time samples differ by more than 127.

#define MAX_INT32U_VALUE (0xFFFFFFFFUL)
Returns true if t1 is greater than t2. Can only account for 1 wrap around of the variable before it is wrong.

#define HALF_MAX_INT32U_VALUE (0×80000000UL)
Returns the elapsed time between two 8 bit values. Result may not be valid if the time samples differ by more than 127.

#define timeGTorEqualInt32u (t1, t2)
Returns the elapsed time between two 8 bit values. Result may not be valid if the time samples differ by more than 127.

Copyright © 2025 Silicon Laboratories. All rights reserved.

#define MILLISECOND_TICKS_PER_SECOND 1000UL
Returns the elapsed time between two 8 bit values. Result may not be valid if the time samples differ by more than 127.

#define MILLISECOND_TICKS_PER_DECISECOND (MILLISECOND_TICKS_PER_SECOND / 10)
Returns the elapsed time between two 8 bit values. Result may not be valid if the time samples differ by more than 127.

#define MILLISECOND_TICKS_PER_QUARTERSECOND (MILLISECOND_TICKS_PER_SECOND >> 2)
Returns the elapsed time between two 8 bit values. Result may not be valid if the time samples differ by more than 127.

#define MILLISECOND_TICKS_PER_MINUTE (60UL * MILLISECOND_TICKS_PER_SECOND)
Returns the elapsed time between two 8 bit values. Result may not be valid if the time samples differ by more than 127.

#define MILLISECOND_TICKS_PER_HOUR (60UL * MILLISECOND_TICKS_PER_MINUTE)
Returns the elapsed time between two 8 bit values. Result may not be valid if the time samples differ by more than 127.

#define MILLISECOND_TICKS_PER_DAY (24UL * MILLISECOND_TICKS_PER_HOUR)
Returns the elapsed time between two 8 bit values. Result may not be valid if the time samples differ by more than 127.

#define EMBER_TASK_COUNT (3)
The number of event tasks that can be used to schedule and run events. Connect stack requires one, while another is used for Application Framework events.

#define emberEventControlSetInactive (control)
Set EmberEventControl as inactive (no pending event).

#define emberEventControlGetActive (control)
Check whether EmberEventControl is currently active. An event is considered active if it is set to run some time in the future (activated by emberEventControlSetActive(), emberEventControlSetDelayMS() or any other emberEventControlSetDelay* functions)

#define emberEventControlSetActive (control)
Set EmberEventControl to run at the next available opportunity.

#define EMBER_MAX_EVENT_CONTROL_DELAY_MS (HALF_MAX_INT32U_VALUE - 1)
The maximum delay that may be passed to emberEventControlSetDelayMS().

#define emberEventControlSetDelayMS (control, delay)
Set EmberEventControl to run some milliseconds in the future.

#define EMBER_MAX_EVENT_CONTROL_DELAY_QS (EMBER_MAX_EVENT_CONTROL_DELAY_MS >> 8)
The maximum delay that may be passed to emberEventControlSetDelayQS().

#define emberEventControlSetDelayQS (control, delay)
Set EmberEventControl to run some quarter seconds in the future.

#define EMBER_MAX_EVENT_CONTROL_DELAY_MINUTES (EMBER_MAX_EVENT_CONTROL_DELAY_MS >> 16)
The maximum delay that may be passed to emberEventControlSetDelayMinutes().

#define emberEventControlSetDelayMinutes (control, delay)
Set EmberEventControl to run some minutes in the future.

#define emberEventControlGetRemainingMS (control)
Check when the event is scheduled to run.

#define emberTaskEnableIdling (allow)
Enable or disable idling.

#define emberMarkTaskActive (taskid)
Calling it indicates that a task has something to do, so it should prevent the CPU from idling until emberMarkTaskIdle is next called on this task.

Macros

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
#define __EVENT_H__
```

Time Manipulation Macros Documentation

sli_event_control_set_active

> void sli_event_control_set_active (EmberEventControl * event)

Set EmberEventControl to run at the next available opportunity.

Parameters

Type	Direction	Argument Name	Description
EmberEventControl *	[in]	event	Pointer to the control of the event to set active

Warnings

- Applications should use emberEventControlSetActive() instead.

emEventControlSetDelayMS

> void emEventControlSetDelayMS (EmberEventControl * event, uint32_t delay)

Set EmberEventControl to run some milliseconds in the future.

Parameters

Type	Direction	Argument Name	Description
EmberEventControl *	[in]	event	Pointer to the control of the event to run.
uint32_t	[in]	delay	The delay in milliseconds. Must be less than EMBER_MAX_EVENT_CONTROL_DELAY_MS

Warnings

- Applications should use emberEventControlSetDelayMS() instead.

emEventControlGetRemainingMS

> uint32_t emEventControlGetRemainingMS (EmberEventControl * event)

Check when the event is scheduled to run.

Parameters

Type	Direction	Argument Name	Description
EmberEventControl *	[in]	event	Pointer to the control of the event in question.

Returns

- Return the amount of milliseconds remaining before the event is scheduled to run. If the event is inactive, MAX_INT32U_VALUE is returned.

Warnings

Copyright © 2025 Silicon Laboratories. All rights reserved.

- Applications should use emberEventControlGetRemainingMS() instead.

emberRunEvents

```
void emberRunEvents (EmberEventData * events)
```

Start an event handler if anything is scheduled when this function is called.

Parameters

Type	Direction	Argument Name	Description
EmberEventData *	[in]	events	Pointer to the array of events.

An application typically creates an array of events along with their handlers. This function should be called in the main loop to run those events. **Warnings**

- This is normally handled by emberRunTask() in the main plugin.

emberRunTask

```
void emberRunTask (EmberTaskId taskid)
```

Start an event handler if there is anything scheduled at the moment this function is called.

Parameters

Type	Direction	Argument Name	Description
EmberTaskId	N/A	taskid	

If an application has initialized a task via emberTaskInit(), to run the events associated with that task, it should call emberRunTask() instead of emberRunEvents().

Warnings

- This is normally handled by the main plugin.

emberMsToNextEvent

```
uint32_t emberMsToNextEvent (EmberEventData * events, uint32_t maxMs)
```

Check when the next event is scheduled to run.

Parameters

Type	Direction	Argument Name	Description
EmberEventData *	[in]	events	An array of events to check.
uint32_t	[in]	maxMs	If no event is scheduled before maxMs, maxMs will be returned

Returns

- Returns the number of milliseconds before the next event is scheduled to expire, or `maxMs` if no event is scheduled to expire within that time.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Note

- If any events are modified within an interrupt, to guarantee the accuracy of this API, it must be called with interrupts disabled.

See Also

- emberMsToNextEventExtended()

emberMsToNextEventExtended

```
uint32_t emberMsToNextEventExtended (EmberEventData * events, uint32_t maxMs, uint8_t * returnIndex)
```

Check when the next event is scheduled to run.

Parameters

Type	Direction	Argument Name	Description
EmberEventData *	[in]	events	An array of events to check.
uint32_t	[in]	maxMs	If no event is scheduled before maxMs, maxMs will be returned
uint8_t *	[out]	returnIndex	If not NULL pointer was passed, the index of the next event will be returned here, or 0xFF if no event is scheduled before maxMs.

Returns

- Returns the number of milliseconds before the next event is scheduled to expire, or `maxMs` if no event is scheduled to expire within that time.

Note

- If any events are modified within an interrupt, to guarantee the accuracy of this API, it must be called with interrupts disabled.

See Also

- emberMsToNextEvent()

emberMsToNextStackEvent

```
uint32_t emberMsToNextStackEvent (void )
```

Check when the next stack event is scheduled to run.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Returns the number of milliseconds before the next stack event is scheduled to run.

emberTaskInit

Copyright © 2025 Silicon Laboratories. All rights reserved.

> EmberTaskId emberTaskInit (EmberEventData * events)

Initialize a task for managing events and processor idling state.

Parameters

Type	Direction	Argument Name	Description
EmberEventData *	[in]	events	Pointer to the array of events to manage

Returns

- Returns the EmberTaskId which represents the newly created task.

Note

- After the task is created emberRunTask() should be called periodically.

emberMarkTaskIdle

> bool emberMarkTaskIdle (EmberTaskId taskid)

Try to idle the CPU, unless any events in any tasks are pending.

Parameters

Type	Direction	Argument Name	Description
EmberTaskId	[in]	taskid	the task which should handle the idling.

Returns

- Returns **true** if the processor was idled **false** if idling wasn't permitted because a task has something to do.

Note

- This API should always be called with interrupts disabled. It will forcibly re-enable interrupts before returning.

emTaskEnableIdling

> void emTaskEnableIdling (bool allow)

Enable or disable idling.

Parameters

Type	Direction	Argument Name	Description
bool	[in]	allow	Setting it to **true** will enable, while setting it to **false** will disable idling.

Warnings

- Applications should use emberTaskEnableIdling() instead.

emMarkTaskActive

> void emMarkTaskActive (EmberTaskId taskid)

Copyright © 2025 Silicon Laboratories. All rights reserved.

Calling it indicates that a task has something to do, so it should prevent the CPU from idling until emberMarkTaskIdle is next called on this task.

Parameters

Type	Direction	Argument Name	Description
EmberTaskId	[in]	taskid	The task to mark active.

Warnings

- Applications should use emberMarkTaskActive() instead.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Memory Buffer

Memory Buffer

Ember Connect API dynamically allocates and frees memory.

Generally, dynamic memory allocation in embedded code is not recommended. However, in some cases, the drawbacks of avoiding them is even bigger. Using C standard library dynamic memory is still not recommended due because it could cause fragmented memory.

For these reasons, Connect allocates some (configurable in the stack common plugin) memory as HEAP at compile-time. Memory allocation from this heap is possible through the API below.

See memory-buffer.h for source code.

Handlers

The Application Framework implements all handlers, directly calling their associated callbacks. By default, Connect projects declare such callbacks as stubs in flex-callbacks-stubs.c. Hence, to use an enabled Connect feature, applications should replace the stub with their own implementation of the associated callback (typically in flex-callbacks.c). See UG235.04 for more info.

void	emberMarkApplicationBuffersHandler(void) This handler is invoked by the memory buffers system garbage collector and allows the application to properly mark the application-defined EmberBuffer variables with emberMarkBuffer().

APIs

EmberBuffer	emberAllocateBuffer(uint16_t dataSizeInBytes) Dynamically allocates memory.
void	emberMarkBuffer(EmberBuffer *buffer) Prevent the garbage collector from reclaiming the memory associated with the passed EmberBuffer. The application should call this API within the emberMarkApplicationBuffersHandler() stack handler for each EmberBuffer object.
uint8_t *	emberGetBufferPointer(EmberBuffer buffer) Return a pointer to the memory segment corresponding to the passed EmberBuffer buffer. Notice that the garbage collector can move memory segments to defragment the available memory. As result, the application should always use this API to obtain an updated pointer prior to accessing the memory.
uint16_t	emberGetBufferLength(EmberBuffer buffer) Return the length in bytes of the passed EmberBuffer buffer.
uint16_t	emberGetAvailableBufferMemory(void) Return the available memory at the buffer manager in bytes.

Macros

#define	EMBER_NULL_BUFFER 0×0000u A special EmberBuffer ID indicating that no memory is currently allocated.

Handlers Documentation

Copyright © 2025 Silicon Laboratories. All rights reserved.

emberMarkApplicationBuffersHandler

```
void emberMarkApplicationBuffersHandler (void )
```

This handler is invoked by the memory buffers system garbage collector and allows the application to properly mark the application-defined EmberBuffer variables with emberMarkBuffer().

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Warnings

- Implement associated callback emberAfMarkApplicationBuffersCallback() to use. See Handlers for additional information.

APIs Documentation

emberAllocateBuffer

```
EmberBuffer emberAllocateBuffer (uint16_t dataSizeInBytes)
```

Dynamically allocates memory.

Parameters

Type	Direction	Argument Name	Description
uint16_t	[in]	dataSizeInBytes	The size in bytes of the memory to be allocated.

Returns

- An EmberBuffer value of EMBER_NULL_BUFFER if the memory management system could not allocate the requested memory, or any other EmberBuffer value indicating that the requested memory was successfully allocated. The allocated memory can easily be freed by assigning an EmberBuffer variable to EMBER_NULL_BUFFER. The memory will be freed by the garbage collector during the next emberTick() call.

emberMarkBuffer

```
void emberMarkBuffer (EmberBuffer * buffer)
```

Prevent the garbage collector from reclaiming the memory associated with the passed EmberBuffer. The application should call this API within the emberMarkApplicationBuffersHandler() stack handler for each EmberBuffer object.

Parameters

Type	Direction	Argument Name	Description
EmberBuffer *	[in]	buffer	A pointer to the EmberBuffer buffer to be marked.

emberGetBufferPointer

```
uint8_t * emberGetBufferPointer (EmberBuffer buffer)
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Return a pointer to the memory segment corresponding to the passed EmberBuffer buffer. Notice that the garbage collector can move memory segments to defragment the available memory. As result, the application should always use this API to obtain an updated pointer prior to accessing the memory.

Parameters

Type	Direction	Argument Name	Description
EmberBuffer	[in]	buffer	A pointer to the EmberBuffer buffer for which the corresponding memory pointer should be returned.

Returns

- A NULL pointer if the passed EmberBuffer value is EMBER_NULL_BUFFER. Otherwise, a pointer to the corresponding memory segment.

emberGetBufferLength

> uint16_t emberGetBufferLength (EmberBuffer buffer)

Return the length in bytes of the passed EmberBuffer buffer.

Parameters

Type	Direction	Argument Name	Description
EmberBuffer	[in]	buffer	A pointer to the EmberBuffer buffer for which the corresponding length in bytes should be returned.

Returns

- The length in bytes of a memory segment corresponding to the passed EmberBuffer buffer.

emberGetAvailableBufferMemory

> uint16_t emberGetAvailableBufferMemory (void)

Return the available memory at the buffer manager in bytes.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- The number of available bytes.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Messaging

Messaging

Connect APIs and handlers for sending and receiving messages.

t_stack

Note that MAC mode and Extended star/direct mode use different APIs for messaging.

See message.h for source code.

Handlers

The Application Framework implements all handlers, directly calling their associated callbacks. By default, Connect projects declare such callbacks as stubs in flex-callbacks-stubs.c. Hence, to use an enabled Connect feature, applications should replace the stub with their own implementation of the associated callback (typically in flex-callbacks.c). See UG235.04 for more info.

void	emberMessageSentHandler(EmberStatus status, EmberOutgoingMessage *message) This handler is invoked when the stack has completed sending a message.
void	emberMacMessageSentHandler(EmberStatus status, EmberOutgoingMacMessage *message) This handler is invoked when a node of EMBER_MAC_MODE_DEVICE type or EMBER_MAC_MODE_SLEEPY_DEVICE type has completed sending a MAC frame.
void	emberIncomingMessageHandler(EmberIncomingMessage *message) This handler is invoked when a packet has been received from a node type other than EMBER_MAC_MODE_DEVICE or EMBER_MAC_MODE_SLEEPY_DEVICE.
void	emberIncomingMacMessageHandler(EmberIncomingMacMessage *message) This handler is invoked when a node of EMBER_MAC_MODE_DEVICE type or EMBER_MAC_MODE_SLEEPY_DEVICE has received a MAC frame.
EmberStatus	emberMessageSend(EmberNodeId destination, uint8_t endpoint, uint8_t messageTag, EmberMessageLength messageLength, uint8_t *message, EmberMessageOptions options) Send a message to the passed destination short ID.
EmberStatus	emberMacMessageSend(EmberMacFrame *macFrame, uint8_t messageTag, EmberMessageLength messageLength, uint8_t *message, EmberMessageOptions options) Create a MAC level frame and sends it to the passed destination. This API can only be used for nodes of EMBER_MAC_MODE_DEVICE node type or EMBER_MAC_MODE_SLEEPY_DEVICE node type.
EmberStatus	emberPollForData(void) Send a data request command to the parent node. Note that if the node short ID is a value of EMBER_USE_LONG_ADDRESS, the node shall use its long ID as source address.
EmberStatus	emberSetPollDestinationAddress(EmberMacAddress *destination) Set data polls destination address for nodes of EMBER_MAC_MODE_DEVICE node type or EMBER_MAC_MODE_SLEEPY_DEVICE node type.
uint16_t	emberGetMaximumPayloadLength(EmberMacAddressMode srcAddressMode, EmberMacAddressMode dstAddressMode, bool interpan, bool secured) Return the maximum payload according to the passed source and destination addressing modes, the passed secured flag, and the current configuration of the node.

Copyright © 2025 Silicon Laboratories. All rights reserved.

bool	emberUsingLongMessages(void) Indicates if the stack is currently using long messages or not.
EmberStatus	emberNcpSetLongMessagesUse(bool useLongMessages) Set the current message length that the stack uses.
EmberStatus	emberPurgeIndirectMessages(void) Purge all indirect transmissions from the indirect message queue.
EmberStatus	emberSetIndirectQueueTimeout(uint32_t timeoutMs) Set indirect queue timeout value. The indirect queue timeout is set by default to EMBER_INDIRECT_TRANSMISSION_TIMEOUT_MS.

Macros

#define	EMBER_MAX_UNSECURED_APPLICATION_PAYLOAD_LENGTH 111
#define	EMBER_MAX_SECURED_APPLICATION_PAYLOAD_LENGTH 102
#define	EMBER_MAX_ENDPOINT 0xF The maximum allowed endpoint value.

Handlers Documentation

emberMessageSentHandler

> void emberMessageSentHandler (EmberStatus status, EmberOutgoingMessage * message)

This handler is invoked when the stack has completed sending a message.

Parameters

Type	Direction	Argument Name	Description
EmberStatus	[in]	status	An EmberStatus value of: • EMBER_SUCCESS if an ACK was received from the destination or no ACK was requested. • EMBER_MAC_NO_ACK_RECEIVED if an ACK was requested and no ACK was received. • EMBER_MAC_INDIRECT_TIMEOUT if the destination is a sleepy node and the packet timed-out before the sleepy node sent a data request. • EMBER_MAC_INDIRECT_MESSAGE_PURGED if the destination is a sleepy node and it was removed from the child table while the packet was stored in the indirect queue. • EMBER_PHY_TX_CCA_FAIL if the node failed all the clear channel assessment attempts. • EMBER_PHY_TX_INCOMPLETE if the transmission was not completed correctly.
EmberOutgoingMessage *	[in]	message	An EmberOutgoingMessage describing the outgoing packet.

Warnings

- Implement associated callback emberAfMessageSentCallback() to use. See Handlers for additional information.

Copyright © 2025 Silicon Laboratories. All rights reserved.

emberMacMessageSentHandler

> void emberMacMessageSentHandler (EmberStatus status, EmberOutgoingMacMessage * message)

This handler is invoked when a node of EMBER_MAC_MODE_DEVICE type or EMBER_MAC_MODE_SLEEPY_DEVICE type has completed sending a MAC frame.

Parameters

Type	Direction	Argument Name	Description
EmberStatus	[in]	status	An EmberStatus value of: • EMBER_SUCCESS if an ACK was received from the destination or no ACK was requested. • EMBER_MAC_NO_ACK_RECEIVED if an ACK was requested and no ACK was received. • EMBER_MAC_INDIRECT_TIMEOUT if the MAC frame was sent out via the indirect queue and the it timed-out before a data request was received. • EMBER_MAC_INDIRECT_MESSAGE_PURGED if the MAC frame was sent out via the indirect queue and it was removed prior to a data request being received. See emberPurgeIndirectMessages(). • EMBER_MAC_SECURITY_FAILED if the stack failed to encrypt the message. This typically occurs when a node is sending a message using short source addressing with an address other than the node's short address and the no mapping to a corresponding address was found in the short-to-long address mapping table. The application should use the emberMacAddShortToLongAddressMapping to populate such table. • EMBER_PHY_TX_CCA_FAIL if the node failed all the clear channel assessment attempts. • EMBER_PHY_TX_INCOMPLETE if the transmission was not completed correctly.
EmberOutgoingMacMessage *	[in]	message	An EmberOutgoingMacMessage describing the outgoing MAC frame.

Warnings

- Implement associated callback emberAfMacMessageSentCallback() to use. See Handlers for additional information.

emberIncomingMessageHandler

> void emberIncomingMessageHandler (EmberIncomingMessage * message)

This handler is invoked when a packet has been received from a node type other than EMBER_MAC_MODE_DEVICE or EMBER_MAC_MODE_SLEEPY_DEVICE.

Parameters

Type	Direction	Argument Name	Description
EmberIncomingMessage *	[in]	message	An EmberIncomingMessage describing the incoming packet.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Warnings

- Implement associated callback emberAfIncomingMessageCallback() to use. See Handlers for additional information.

emberIncomingMacMessageHandler

> void emberIncomingMacMessageHandler (EmberIncomingMacMessage * message)

This handler is invoked when a node of EMBER_MAC_MODE_DEVICE type or EMBER_MAC_MODE_SLEEPY_DEVICE has received a MAC frame.

Parameters

Type	Direction	Argument Name	Description
EmberIncomingMacMessage *	[in]	message	An EmberIncomingMacMessage describing the incoming packet.

Warnings

- Implement associated callback emberAfIncomingMacMessageCallback() to use. See Handlers for additional information.

emberMessageSend

> EmberStatus emberMessageSend (EmberNodeId destination, uint8_t endpoint, uint8_t messageTag, EmberMessageLength messageLength, uint8_t * message, EmberMessageOptions options)

Send a message to the passed destination short ID.

Parameters

Type	Direction	Argument Name	Description
EmberNodeId	[in]	destination	The destination node short ID.
uint8_t	[in]	endpoint	The destination endpoint of the outgoing message. This value can't exceed EMBER_MAX_ENDPOINT.
uint8_t	[in]	messageTag	A value chosen by the application. This value will be passed in the corresponding emberMessageSentHandler() call.
EmberMessageLength	[in]	messageLength	The size of the message payload in bytes. Use the emberGetMaximumPayloadLength() API to determine the maximum message length allowed.
uint8_t *	[in]	message	A pointer to an array of bytes containing the message payload.
EmberMessageOptions	[in]	options	Specifies the EmberMessageOptions for the outgoing message.

Returns

- an EmberStatus value of:
  - EMBER_SUCCESS if the message was accepted by the stack. If a success status is returned, the emberMessageSentHandler() callback is invoked by the stack to indicate whether the message was successfully delivered or the reason for failure.
  - EMBER_INVALID_CALL if the node is not joined to a network or the node is of EMBER_MAC_MODE_DEVICE device type or EMBER_MAC_MODE_SLEEPY_DEVICE (use emberMacMessageSend instead).
  - EMBER_BAD_ARGUMENT if the packet length is 0, the passed TX options indicates some feature that is not supported, the passed endpoint exceeds EMBER_MAX_ENDPOINT

Copyright © 2025 Silicon Laboratories. All rights reserved.

EMBER_MESSAGE_TOO_LONG if the message does not fit in a single frame.
- EMBER_PHY_TX_BUSY if the message cannot be sent since the node does not support MAC queuing and the radio is currently busy.
- EMBER_MAC_TRANSMIT_QUEUE_FULL if the outgoing MAC queue is currently full.
- EMBER_NO_BUFFERS if the stack could not allocate enough RAM to store the submitted message.
- EMBER_MAC_UNKNOWN_DESTINATION if the node is part of a star network and the destination node does not appear in the node's routing table.
- EMBER_MAC_SECURITY_NOT_SUPPORTED if the message was requested to be sent out secured and either the local node does not support security or the destination node is known to not support security.
- EMBER_MAC_BUSY if the message was not accepted because the MAC is currently performing some critical operation.

emberMacMessageSend

> EmberStatus emberMacMessageSend (EmberMacFrame * macFrame, uint8_t messageTag, EmberMessageLength messageLength, uint8_t * message, EmberMessageOptions options)

Create a MAC level frame and sends it to the passed destination. This API can only be used for nodes of EMBER_MAC_MODE_DEVICE node type or EMBER_MAC_MODE_SLEEPY_DEVICE node type.

Parameters

Type	Direction	Argument Name	Description
EmberMacFrame *	[in]	macFrame	A pointer to an EmberMacFrame struct that specifies the source and destination addresses and the source and destination PAN IDs for the message to be sent. Note that if the source/destination PAN ID is not specified, it defaults to the node's PAN ID. Also, the destination address mode must be either EMBER_MAC_ADDRESS_MODE_SHORT or EMBER_MAC_ADDRESS_MODE_LONG.
uint8_t	[in]	messageTag	A value chosen by the application. This value will be passed in the corresponding emberMacMessageSentHandler() call.
EmberMessageLength	[in]	messageLength	The size in bytes of the message payload. The application can use the emberGetMaximumPayloadLength() API to determine the maximum allowable payload, given a permutation of source and destination addressing and other TX options.
uint8_t *	[in]	message	A pointer to an array of bytes containing the message payload.
EmberMessageOptions	[in]	options	Specifies the EmberMessageOptions for the outgoing message.

Returns

- an EmberStatus value of:
  - EMBER_SUCCESS if the message was accepted by the stack. If a success status is returned, the emberMacMessageSentHandler() callback will be invoked by the stack to indicate whether the message was successfully delivered or the reason for failure.
  - EMBER_INVALID_CALL if the node is of a node type other than EMBER_MAC_MODE_DEVICE or EMBER_MAC_MODE_SLEEPY_DEVICE.
  - EMBER_BAD_ARGUMENT if the packet length is 0, the passed TX options indicates some feature that is not supported or the destination address mode is set to EMBER_MAC_ADDRESS_MODE_NONE.
  - EMBER_MESSAGE_TOO_LONG if the message does not fit in a single frame.
  - EMBER_PHY_TX_BUSY if the message cannot be sent since the node does not support MAC queuing and the radio is currently busy.
  - EMBER_MAC_TRANSMIT_QUEUE_FULL if the outgoing MAC queue is currently full.
  - EMBER_NO_BUFFERS if the stack could not allocate enough RAM to store the submitted message.
  - EMBER_MAC_SECURITY_NOT_SUPPORTED if the message was requested to be sent out with a security but no security plugin was enabled.

Copyright © 2025 Silicon Laboratories. All rights reserved.

EMBER_MAC_BUSY if the message was not accepted because the MAC is currently performing a critical operation.

emberPollForData

EmberStatus emberPollForData (void)

Send a data request command to the parent node. Note that if the node short ID is a value of EMBER_USE_LONG_ADDRESS, the node shall use its long ID as source address.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- and EmberStatus value of:
  - EMBER_SUCCESS if the data poll was accepted by the MAC layer.
  - EMBER_INVALID_CALL if the node is not joined to a network, the node is not an end device, an EMBER_MAC_MODE_DEVICE or an EMBER_MAC_MODE_SLEEPY_DEVICE, or the node is of EMBER_MAC_MODE_DEVICE or EMBER_MAC_MODE_SLEEPY_DEVICE node type, is not joined to a coordinator and the poll destination was not correctly set via the emberSetPollDestinationAddress() API.
  - EMBER_MAC_BUSY if the MAC is currently performing a critical operation.

emberSetPollDestinationAddress

EmberStatus emberSetPollDestinationAddress (EmberMacAddress * destination)

Set data polls destination address for nodes of EMBER_MAC_MODE_DEVICE node type or EMBER_MAC_MODE_SLEEPY_DEVICE node type.

Parameters

Type	Direction	Argument Name	Description
EmberMacAddress *	N/A	destination	

Returns

- and EmberStatus value of EMBER_SUCCESS if the data poll destination was correctly set, or another EmberStatus value indicating the reason of failure.

emberGetMaximumPayloadLength

uint16_t emberGetMaximumPayloadLength (EmberMacAddressMode srcAddressMode, EmberMacAddressMode dstAddressMode, bool interpan, bool secured)

Return the maximum payload according to the passed source and destination addressing modes, the passed secured flag, and the current configuration of the node.

Parameters

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
EmberMacAddressMode	[in]	srcAddressMode	An EmberMacAddressMode value indicating the mode of the source address. Note, this parameter is only meaningful if the node was started as EMBER_MAC_MODE_DEVICE or EMBER_MAC_MODE_SLEEPY_DEVICE.
EmberMacAddressMode	[in]	dstAddressMode	An EmberMacAddressMode value indicating the mode of the destination address. Note, this parameter is only meaningful if the node was started as EMBER_MAC_MODE_DEVICE or EMBER_MAC_MODE_SLEEPY_DEVICE.
bool	[in]	interpan	Indicates whether the frame is an interpan frame or not. Note, this parameter is only meaningful if the node was started as EMBER_MAC_MODE_DEVICE or EMBER_MAC_MODE_SLEEPY_DEVICE.
bool	[in]	secured	Indicates whether the frame should be secured or not.

Returns

- The maximum payload length in bytes achievable according to the passed parameters or **0xFF** if the node is currently active on a network or any of the passed parameters are invalid.

emberUsingLongMessages

```
bool emberUsingLongMessages (void )
```

Indicates if the stack is currently using long messages or not.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- True if the stack currently uses long messages (length stored in a uint16_t) or false if it is not the case (length stored in a uint8_t).

emberNcpSetLongMessagesUse

```
EmberStatus emberNcpSetLongMessagesUse (bool useLongMessages)
```

Set the current message length that the stack uses.

Parameters

Type	Direction	Argument Name	Description
bool	[in]	useLongMessages	True to use long messages (length stored in a uint16_t), false to use short messages (length stored in a uint8_t).

Note

- This API is here to assure retro compatibility with old NCP Host lib versions. In NCP Host lib versions that do not support OFDM features (v1.1 and older), only short messaging is supported. For the NCP, short messages are used by default. The Host lib needs to call this API with useLongMessages set to true if it supports OFDM.

Warnings

Copyright © 2025 Silicon Laboratories. All rights reserved.

- This API changes the behavior of the Connect Serialization Protocol. It only has effect when using a RTOS or the NCP. Changing it may result in packets being incorrectly transfered through CSP when using a SUN-OFDM or SUN-FSK PHY.

emberPurgeIndirectMessages

> EmberStatus emberPurgeIndirectMessages (void)

Purge all indirect transmissions from the indirect message queue.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- an EmberStatus value of EMBER_SUCCESS if all indirect messages were purged, or another EmberStatus value indicating the reason of failure.

emberSetIndirectQueueTimeout

> EmberStatus emberSetIndirectQueueTimeout (uint32_t timeoutMs)

Set indirect queue timeout value. The indirect queue timeout is set by default to EMBER_INDIRECT_TRANSMISSION_TIMEOUT_MS.

Parameters

Type	Direction	Argument Name	Description
uint32_t	N/A	timeoutMs	The timeout in milliseconds to be set.

Returns

- an EmberStatus value of EMBER_SUCCESS if the passed timeout was successfully set, or a value of EMBER_BAD_ARGUMENT if the passed value is invalid.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Connect Application Framework API Reference

Connect Application Framework API Reference

Application Framework includes plugins that are built on top of the Connect stack.

Modules

Application Framework Common

Command Interpreter Plugin

Debug Print Plugin

Mailbox Client Plugin

Mailbox Server Plugin

Mailbox Common

Ota Unicast Bootloader Client Plugin

Ota Unicast Bootloader Server Plugin

Ota Unicast Bootloader Common

Ota Broadcast Bootloader Client Plugin

Ota Broadcast Bootloader Server Plugin

Ota Broadcast Bootloader Common

Poll Plugin

WSTK Sensors Plugin

Connect RAIL DMP Component

Copyright © 2025 Silicon Laboratories. All rights reserved.

Application Framework Common

Application Framework Common

Application framework common.

Declare all required application framework globals, initialize the Connect stack, and dispatch stack callbacks calls as needed to the application components.

Callbacks

void emberAfInitCallback(void)
Application Framework Initialization Callback.

void emberAfTickCallback(void)
Application Framework Tick Callback.

void emberAfStackStatusCallback(EmberStatus status)
Application framework equivalent of emberStackStatusHandler.

void emberAfIncomingMessageCallback(EmberIncomingMessage *message)
Application framework equivalent of emberIncomingMessageHandler.

void emberAfIncomingMacMessageCallback(EmberIncomingMacMessage *message)
Application framework equivalent of emberIncomingMacMessageHandler.

void emberAfMessageSentCallback(EmberStatus status, EmberOutgoingMessage *message)
Application framework equivalent of emberMessageSentHandler.

void emberAfMacMessageSentCallback(EmberStatus status, EmberOutgoingMacMessage *message)
Application framework equivalent of emberMacMessageSentHandler.

void emberAfChildJoinCallback(EmberNodeType nodeType, EmberNodeId nodeId)
Application framework equivalent of emberChildJoinHandler.

void emberAfActiveScanCompleteCallback(void)
Application framework equivalent of emberActiveScanCompleteHandler.

void emberAfEnergyScanCompleteCallback(int8_t mean, int8_t min, int8_t max, uint16_t variance)
Application framework equivalent of emberEnergyScanCompleteHandler.

void emberAfMarkApplicationBuffersCallback(void)
Application framework equivalent of emberMarkApplicationBuffersHandler.

void emberAfIncomingBeaconCallback(EmberPanId panId, EmberMacAddress *source, int8_t rssi, bool permitJoining, uint8_t beaconFieldsLength, uint8_t *beaconFields, uint8_t beaconPayloadLength, uint8_t *beaconPayload)
Application framework equivalent of emberIncomingBeaconHandler.

void emberAfFrequencyHoppingStartClientCompleteCallback(EmberStatus status)
Application framework equivalent of emberFrequencyHoppingStartClientCompleteHandler.

void emberAfRadioNeedsCalibratingCallback(void)
Application framework equivalent of emberRadioNeedsCalibratingHandler.

bool emberAfStackIdleCallback(uint32_t *idleTimeMs)
Application framework equivalent of emberStackIdleHandler.

Copyright © 2025 Silicon Laboratories. All rights reserved.

bool emberAfCommonOkToEnterLowPowerCallback(bool enter_em2, uint32_t duration_ms)
Application framework Low Power notification Callback.

Functions

uint32_t emberAfGetResetCause(void)
Get the last reset cause mask.

EmberStatus emberAfAllocateEvent(EmberEventControl **control, void(*handler)(void))
Allocate a new event to the app event table.

Callbacks Documentation

emberAfInitCallback

```
void emberAfInitCallback (void )
```

Application Framework Initialization Callback.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

A callback invoked once during the initialization. It is called after the stack and plugins initialization.

emberAfTickCallback

```
void emberAfTickCallback (void )
```

Application Framework Tick Callback.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

A callback invoked in each iteration of the application super loop and can be used to perform periodic functions. The frequency with which this function is called depends on how quickly the main loop runs. If the application blocks at any time during the main loop, this function will not be called until execution resumes.

emberAfStackStatusCallback

```
void emberAfStackStatusCallback (EmberStatus status)
```

Application framework equivalent of emberStackStatusHandler.

Parameters

Type	Direction	Argument Name	Description
EmberStatus	N/A	status	

Copyright © 2025 Silicon Laboratories. All rights reserved.

emberAfIncomingMessageCallback

```
void emberAfIncomingMessageCallback (EmberIncomingMessage * message)
```

Application framework equivalent of emberIncomingMessageHandler.

Parameters

Type	Direction	Argument Name	Description
EmberIncomingMessage *	N/A	message	

emberAfIncomingMacMessageCallback

```
void emberAfIncomingMacMessageCallback (EmberIncomingMacMessage * message)
```

Application framework equivalent of emberIncomingMacMessageHandler.

Parameters

Type	Direction	Argument Name	Description
EmberIncomingMacMessage *	N/A	message	

emberAfMessageSentCallback

```
void emberAfMessageSentCallback (EmberStatus status, EmberOutgoingMessage * message)
```

Application framework equivalent of emberMessageSentHandler.

Parameters

Type	Direction	Argument Name	Description
EmberStatus	N/A	status	
EmberOutgoingMessage *	N/A	message	

emberAfMacMessageSentCallback

```
void emberAfMacMessageSentCallback (EmberStatus status, EmberOutgoingMacMessage * message)
```

Application framework equivalent of emberMacMessageSentHandler.

Parameters

Type	Direction	Argument Name	Description
EmberStatus	N/A	status	
EmberOutgoingMacMessage *	N/A	message	

emberAfChildJoinCallback

Copyright © 2025 Silicon Laboratories. All rights reserved.

> void emberAfChildJoinCallback (EmberNodeType nodeType, EmberNodeId nodeId)

Application framework equivalent of emberChildJoinHandler.

Parameters

Type	Direction	Argument Name	Description
EmberNodeType	N/A	nodeType	
EmberNodeId	N/A	nodeId	

Warnings

- Requires the parent support plugin installed.

emberAfActiveScanCompleteCallback

> void emberAfActiveScanCompleteCallback (void)

Application framework equivalent of emberActiveScanCompleteHandler.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

emberAfEnergyScanCompleteCallback

> void emberAfEnergyScanCompleteCallback (int8_t mean, int8_t min, int8_t max, uint16_t variance)

Application framework equivalent of emberEnergyScanCompleteHandler.

Parameters

Type	Direction	Argument Name	Description
int8_t	N/A	mean	
int8_t	N/A	min	
int8_t	N/A	max	
uint16_t	N/A	variance	

emberAfMarkApplicationBuffersCallback

> void emberAfMarkApplicationBuffersCallback (void)

Application framework equivalent of emberMarkApplicationBuffersHandler.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Copyright © 2025 Silicon Laboratories. All rights reserved.

emberAfIncomingBeaconCallback

void emberAfIncomingBeaconCallback (EmberPanId panId, EmberMacAddress * source, int8_t rssi, bool permitJoining, uint8_t beaconFieldsLength, uint8_t * beaconFields, uint8_t beaconPayloadLength, uint8_t * beaconPayload)

Application framework equivalent of emberIncomingBeaconHandler.

Parameters

Type	Direction	Argument Name	Description
EmberPanId	N/A	panId	
EmberMacAddress *	N/A	source	
int8_t	N/A	rssi	
bool	N/A	permitJoining	
uint8_t	N/A	beaconFieldsLength	
uint8_t *	N/A	beaconFields	
uint8_t	N/A	beaconPayloadLength	
uint8_t *	N/A	beaconPayload	

emberAfFrequencyHoppingStartClientCompleteCallback

void emberAfFrequencyHoppingStartClientCompleteCallback (EmberStatus status)

Application framework equivalent of emberFrequencyHoppingStartClientCompleteHandler.

Parameters

Type	Direction	Argument Name	Description
EmberStatus	N/A	status	

emberAfRadioNeedsCalibratingCallback

void emberAfRadioNeedsCalibratingCallback (void)

Application framework equivalent of emberRadioNeedsCalibratingHandler.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

emberAfStackIdleCallback

bool emberAfStackIdleCallback (uint32_t * idleTimeMs)

Application framework equivalent of emberStackIdleHandler.

Parameters

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
uint32_t *	N/A	idleTimeMs	

emberAfCommonOkToEnterLowPowerCallback

> bool emberAfCommonOkToEnterLowPowerCallback (bool enter_em2, uint32_t duration_ms)

Application framework Low Power notification Callback.

Parameters

Type	Direction	Argument Name	Description
bool	[in]	enter_em2	**true** if the system is about to sleep or **false** to idle.
uint32_t	[in]	duration_ms	Duration of the low power period. Time to the next event.

A callback invoked when the system is about to go sleeping.

Returns

- **true** if the application allows the system to go to sleep.

Function Documentation

emberAfGetResetCause

> uint32_t emberAfGetResetCause (void)

Get the last reset cause mask.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- A reset cause mask.

Note

- This API replaces halGetResetInfo() or halGetExtendedResetInfo. emberAfGetResetCause() is a RMU_ResetCauseGet() overhaul. See the reference manual of the EMLIB RMU for a description of the returned reset cause mask.

emberAfAllocateEvent

> EmberStatus emberAfAllocateEvent (EmberEventControl ** control, void(*)(void) handler)

Allocate a new event to the app event table.

Parameters

Type	Direction	Argument Name	Description
EmberEventControl **	[out]	control	The EmberEventControl to allocate

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
void(*)(void)	[in]	handler	Pointer to the handler function associated to the event

Returns

- An EmberStatus value of:
  - EMBER_SUCCESS if the event was successfully allocated.
  - EMBER_TABLE_FULL if no more event could be allocated.

See Also

- emberAfAllocateEvent()

Copyright © 2025 Silicon Laboratories. All rights reserved.

Command Interpreter Plugin

Command Interpreter Plugin

Copyright © 2025 Silicon Laboratories. All rights reserved.

Debug Print Plugin

Debug Print Plugin

Copyright © 2025 Silicon Laboratories. All rights reserved.

Mailbox Client Plugin

Mailbox Client Plugin

APIs for mailbox client.

Mailbox protocol is designed for devices that can't be online on the network all the time. The most common example for this is a sleepy end device.

Mailbox clients and the server can submit messages into the mailbox, which is stored in RAM on the mailbox server. Clients can then query the mailbox server for available messages.

The mailbox server will notify clients who submit messages when a message was delivered or when it couldn't be delivered due to an error.

Mailbox uses a plugin-configurable protocol endpoint, which is 15 by default.

The server can also configure the size of the mailbox (in number of packets, 25 by default) and the packet timeout, after which the server drops the message and notifies the source of the error.

The mailbox protocol uses standard data messages, so in case of sleepy end devices, it will use the indirect queue. This means that if a sleepy end device sends a request to a mailbox server, the end device should poll for the response.

Note

- Mailbox is not available in MAC mode due to the lack of endpoints.

See mailbox-client.h and mailbox-client.c for source code.

Callbacks

void emberAfPluginMailboxClientMessageSubmitCallback(EmberAfMailboxStatus status, EmberNodeId mailboxServer, EmberNodeId messageDestination, uint8_t tag)

Mailbox Client Message Submit Callback.

void emberAfPluginMailboxClientMessageDeliveredCallback(EmberAfMailboxStatus status, EmberNodeId mailboxServer, EmberNodeId messageDestination, uint8_t tag)

Mailbox Client Message Delivered Callback.

void emberAfPluginMailboxClientCheckInboxCallback(EmberAfMailboxStatus status, EmberNodeId mailboxServer, EmberNodeId messageSource, uint8_t *message, EmberMessageLength messageLength, uint8_t tag, bool moreMessages)

Mailbox Client Check Inbox Callback.

Functions

EmberAfMailboxStatus emberAfPluginMailboxClientMessageSubmit(EmberNodeId mailboxServer, EmberNodeId messageDestination, uint8_t *message, EmberMessageLength messageLength, uint8_t tag, bool useSecurity)

Submit a data message to a mailbox server. If this API returns an EmberAfMailboxStatus value of EMBER_MAILBOX_STATUS_SUCCESS, the corresponding asynchronous callback emberAfPluginMailboxClientMessageSubmitCallback() will be invoked to indicate whether the message was successfully submitted to the mailbox server or to inform the application of the reason of failure.

Copyright © 2025 Silicon Laboratories. All rights reserved.

EmberAfMailboxStatus	emberAfPluginMailboxClientCheckInbox(EmberNodeId mailboxServer, bool useSecurity) Query a mailbox server for pending messages. If this API returns an EmberAfMailboxStatus value of EMBER_MAILBOX_STATUS_SUCCESS, the corresponding asynchronous callback emberAfPluginMailboxClientCheckInboxCallback() will be invoked either to provide the retrieved message or to indicate the reason for failure.

Callbacks Documentation

emberAfPluginMailboxClientMessageSubmitCallback

> void emberAfPluginMailboxClientMessageSubmitCallback (EmberAfMailboxStatus status, EmberNodeId mailboxServer, EmberNodeId messageDestination, uint8_t tag)

Mailbox Client Message Submit Callback.

Parameters

Type	Direction	Argument Name	Description
EmberAfMailboxStatus	[in]	status	An EmberAfMailboxStatus value of: • EMBER_MAILBOX_STATUS_SUCCESS if the data message was accepted by the mailbox server. • EMBER_MAILBOX_STATUS_STACK_ERROR if the message couldn't be delivered to the mailbox server. • EMBER_MAILBOX_STATUS_MESSAGE_NO_RESPONSE if the client timed-out waiting for a response from the server. • EMBER_MAILBOX_STATUS_MESSAGE_TABLE_FULL if the mailbox server table is currently full. • EMBER_MAILBOX_STATUS_MESSAGE_NO_BUFFERS if the server can't allocate enough memory to store the message.
EmberNodeId	[in]	mailboxServer	The node ID of the mailbox server.
EmberNodeId	[in]	messageDestination	The node ID of the destination.
uint8_t	N/A	tag	The tag value passed in the emberAfPluginMailboxClientMessageSubmit() API.

A callback invoked when a message arrived to the mailbox server after a call of emberAfPluginMailboxClientMessageSubmit().

Note

- Receiving this callback requires the reception of a mailbox command message, which is only possible by polling if the message was submitted on a EMBER_STAR_SLEEPY_END_DEVICE.

emberAfPluginMailboxClientMessageDeliveredCallback

> void emberAfPluginMailboxClientMessageDeliveredCallback (EmberAfMailboxStatus status, EmberNodeId mailboxServer, EmberNodeId messageDestination, uint8_t tag)

Mailbox Client Message Delivered Callback.

Parameters

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
EmberAfMailboxStatus	[in]	status	An EmberAfMailboxStatus value of: • EMBER_MAILBOX_STATUS_SUCCESS indicates that the message was successfully delivered to the final destination. • EMBER_MAILBOX_STATUS_MESSAGE_TIMED_OUT indicates that the message timed-out and was removed from the server queue.
EmberNodeId	[in]	mailboxServer	The node ID of the mailbox server where the message was submitted to.
EmberNodeId	[in]	messageDestination	The node ID of the destination.
uint8_t	[in]	tag	The tag value passed in the emberAfPluginMailboxClientMessageSubmit() API.

A callback that may be invoked on the submitter of the message either if the message that was submitted to a mailbox server reached its final destination or it timed-out. Note that the callback is not always called. If the status message from the server is lost, the callback won't be called.

Note

- Receiving this callback requires the reception of a mailbox command message, which is only possible by polling if the message was submitted on a EMBER_STAR_SLEEPY_END_DEVICE.

emberAfPluginMailboxClientCheckInboxCallback

> void emberAfPluginMailboxClientCheckInboxCallback (EmberAfMailboxStatus status, EmberNodeId mailboxServer, EmberNodeId messageSource, uint8_t * message, EmberMessageLength messageLength, uint8_t tag, bool moreMessages)

Mailbox Client Check Inbox Callback.

Parameters

Type	Direction	Argument Name	Description
EmberAfMailboxStatus	[in]	status	An EmberAfMailboxStatus value of: • EMBER_MAILBOX_STATUS_SUCCESS if a message was retrieved from the mailbox server. • EMBER_MAILBOX_STATUS_MESSAGE_NO_DATA if the server has currently no message for this mailbox client. • EMBER_MAILBOX_STATUS_MESSAGE_NO_RESPONSE if the client timed-out waiting for a query response from the mailbox server. • EMBER_MAILBOX_STATUS_STACK_ERROR if the stack failed to deliver the query message to the mailbox server.
EmberNodeId	[in]	mailboxServer	The node id of the mailbox server responding.
EmberNodeId	[in]	messageSource	The source node ID of the retrieved message. Note that this parameter is meaningful only if the status parameter has an EmberAfMailboxStatus value of EMBER_MAILBOX_STATUS_SUCCESS.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
uint8_t *	[in]	message	A pointer to the retrieved message payload. Note that this parameter is meaningful only if the status parameter has an EmberAfMailboxStatus value of EMBER_MAILBOX_STATUS_SUCCESS.
EmberMessageLength	[in]	messageLength	The length in bytes of the retrieved message payload. Note that this parameter is meaningful only if the status parameter has an EmberAfMailboxStatus value of EMBER_MAILBOX_STATUS_SUCCESS.
uint8_t	[in]	tag	The tag value passed in the emberAfPluginMailboxClientMessageSubmit() API. Note that this parameter is meaningful only if the status parameter has an EmberAfMailboxStatus value of EMBER_MAILBOX_STATUS_SUCCESS.
bool	[in]	moreMessages	This flag is **true** if the mailbox server has more pending messages for this mailbox client. Note that this parameter is meaningful only if the status parameter has an EmberAfMailboxStatus value of EMBER_MAILBOX_STATUS_SUCCESS.

This callback is invoked after a successful call to the emberAfPluginMailboxClientCheckInbox() API. If a message was retrieved from the mailbox server, this callback passes it to the application. Otherwise, it indicates the reason for failure to the application.

Note

- Receiving this callback requires the reception of a mailbox command message, which is only possible by polling if the message was submitted on a EMBER_STAR_SLEEPY_END_DEVICE.

Function Documentation

emberAfPluginMailboxClientMessageSubmit

> EmberAfMailboxStatus emberAfPluginMailboxClientMessageSubmit (EmberNodeId mailboxServer, EmberNodeId messageDestination, uint8_t * message, EmberMessageLength messageLength, uint8_t tag, bool useSecurity)

Submit a data message to a mailbox server. If this API returns an EmberAfMailboxStatus value of EMBER_MAILBOX_STATUS_SUCCESS, the corresponding asynchronous callback emberAfPluginMailboxClientMessageSubmitCallback() will be invoked to indicate whether the message was successfully submitted to the mailbox server or to inform the application of the reason of failure.

Parameters

Type	Direction	Argument Name	Description
EmberNodeId	[in]	mailboxServer	The node ID of the mailbox server.
EmberNodeId	[in]	messageDestination	The node ID of the destination for this data message.
uint8_t *	[in]	message	A pointer to the message to be sent.
EmberMessageLength	[in]	messageLength	The length in bytes of the message to be sent.
uint8_t	[in]	tag	A tag value which will be returned in all of the corresponding callbacks: emberAfPluginMailboxClientMessageSubmitCallback(), emberAfPluginMailboxClientMessageDeliveredCallback() and emberAfPluginMailboxClientCheckInboxCallback(). The application can use it to match the callbacks with the call.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
bool	[in]	useSecurity	Set it **true** if the data message should be sent to the server using security.

Returns

- An EmberAfMailboxStatus value of:
  - EMBER_MAILBOX_STATUS_SUCCESS if the message was successfully passed to the network layer to be transmitted to the mailbox server.
  - EMBER_MAILBOX_STATUS_INVALID_CALL if the passed data message is invalid.
  - EMBER_MAILBOX_STATUS_INVALID_ADDRESS if the server ID or the destination ID is an invalid address.
  - EMBER_MAILBOX_STATUS_MESSAGE_TOO_LONG if the passed message does not fit in a single mailbox data message.
  - EMBER_MAILBOX_STATUS_BUSY if the client is still performing a submit message or a query for message action.
  - EMBER_MAILBOX_STATUS_STACK_ERROR if the network layer refused the message (the outgoing queue is currently full).

Note

- Receiving the emberAfPluginMailboxClientMessageSubmitCallback() requires the reception of a mailbox command message, which is only possible by polling if the message was submitted on a EMBER_STAR_SLEEPY_END_DEVICE.

See Also

- emberAfPluginMailboxServerAddMessage()

emberAfPluginMailboxClientCheckInbox

> EmberAfMailboxStatus emberAfPluginMailboxClientCheckInbox (EmberNodeId mailboxServer, bool useSecurity)

Query a mailbox server for pending messages. If this API returns an EmberAfMailboxStatus value of EMBER_MAILBOX_STATUS_SUCCESS, the corresponding asynchronous callback emberAfPluginMailboxClientCheckInboxCallback() will be invoked either to provide the retrieved message or to indicate the reason for failure.

Parameters

Type	Direction	Argument Name	Description
EmberNodeId	[in]	mailboxServer	The node ID of the mailbox server.
bool	[in]	useSecurity	Set it **true** if the request command and the responses to it should be sent secured. If a pending message was sent to a server securely, it will be always retrieved securely. This option only affects the request command and the pending messages that were sent without security to the server.

Returns

- An EmberAfMailboxStatus value of:
  - EMBER_MAILBOX_STATUS_SUCCESS if the query command was successfully passed to the network layer to be transmitted to the mailbox server.
  - EMBER_MAILBOX_STATUS_INVALID_ADDRESS if the passed mailbox server short ID is an invalid address.
  - EMBER_MAILBOX_STATUS_BUSY if the client is still performing a submit message or a query for message action.
  - EMBER_MAILBOX_STATUS_STACK_ERROR if the network layer refused the command (the outgoing queue is currently full).

Note

- Receiving the emberAfPluginMailboxClientCheckInboxCallback() requires the reception of a mailbox command message, which is only possible by polling if the message was submitted on a EMBER_STAR_SLEEPY_END_DEVICE.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Mailbox Server Plugin

Mailbox Server Plugin

APIs for mailbox server.

Mailbox protocol is designed for devices that can't be online on the network all the time. The most common example for this is a sleepy end device.

Mailbox clients and the server can submit messages into the mailbox, which is stored in RAM on the mailbox server. Clients can then query the mailbox server for available messages.

The mailbox server will notify clients who submit messages when a message was delivered or when it couldn't be delivered due to an error.

Mailbox uses a plugin-configurable protocol endpoint, which is 15 by default.

The server can also configure the size of the mailbox (in number of packets, 25 by default) and the packet timeout, after which the server drops the message and notifies the source of the error.

The mailbox protocol uses standard data messages, so in case of sleepy end devices, it will use the indirect queue. This means that if a sleepy end device sends a request to a mailbox server, the end device should poll for the response.

Note

- Mailbox is not available in MAC mode due to the lack of endpoints.

See mailbox-server.h and mailbox-server.c for source code.

Callbacks

void	emberAfPluginMailboxServerMessageDeliveredCallback(EmberAfMailboxStatus status, EmberNodeId messageDestination, uint8_t tag) Mailbox Server Message Delivered Callback.

Functions

EmberAfMailboxStatus	emberAfPluginMailboxServerAddMessage(EmberNodeId destination, uint8_t *message, EmberMessageLength messageLength, uint8_t tag, bool useSecurity) Add a message to the mailbox server queue. The message is stored in the internal queue until the destination node queries the mailbox server node for messages or upon timeout.

Callbacks Documentation

emberAfPluginMailboxServerMessageDeliveredCallback

> void emberAfPluginMailboxServerMessageDeliveredCallback (EmberAfMailboxStatus status, EmberNodeId messageDestination, uint8_t tag)

Mailbox Server Message Delivered Callback.

Parameters

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
EmberAfMailboxStatus	[in]	status	An EmberAfMailboxStatus value of: • EMBER_MAILBOX_STATUS_SUCCESS indicates that the message was successfully delivered to the final destination. • EMBER_MAILBOX_STATUS_MESSAGE_TIMED_OUT indicates that the message timed-out and was removed from the server queue.
EmberNodeId	[in]	messageDestination	The node ID of the destination.
uint8_t	[in]	tag	The tag value passed in the emberAfPluginMailboxServerAddMessage() API.

This callback is invoked at the server when a message submitted locally by the server was successfully delivered or when it timed-out.

See Also

- emberAfPluginMailboxClientMessageDeliveredCallback()

Function Documentation

emberAfPluginMailboxServerAddMessage

> EmberAfMailboxStatus emberAfPluginMailboxServerAddMessage (EmberNodeId destination, uint8_t * message, EmberMessageLength messageLength, uint8_t tag, bool useSecurity)

Add a message to the mailbox server queue. The message is stored in the internal queue until the destination node queries the mailbox server node for messages or upon timeout.

Parameters

Type	Direction	Argument Name	Description
EmberNodeId	[in]	destination	The node ID of the destination for this data message.
uint8_t *	[in]	message	A pointer to the message to be enqueued.
EmberMessageLength	[in]	messageLength	The length in bytes of the message to be enqueued.
uint8_t	[in]	tag	A tag value which will be returned in the corresponding emberAfPluginMailboxServerMessageDeliveredCallback() callback. The application can use to match the callbacks with the call.
bool	[in]	useSecurity	Set it **true** if the data message should be sent to the server using security.

Returns

- An EmberAfMailboxStatus value of:
  - EMBER_MAILBOX_STATUS_SUCCESS if the message was successfully added to the packet queue.
  - EMBER_MAILBOX_STATUS_INVALID_CALL if the passed message is invalid.
  - EMBER_MAILBOX_STATUS_INVALID_ADDRESS if the passed destination address is invalid.
  - EMBER_MAILBOX_STATUS_MESSAGE_TOO_LONG if the payload size of the passed message exceeds the maximum allowable payload for the passed transmission options.
  - EMBER_MAILBOX_STATUS_MESSAGE_TABLE_FULL if the packet table is already full.
  - EMBER_MAILBOX_STATUS_MESSAGE_NO_BUFFERS if not enough memory buffers are available for storing the message content.

See Also

Copyright © 2025 Silicon Laboratories. All rights reserved.

emberAfPluginMailboxClientMessageSubmit()

Copyright © 2025 Silicon Laboratories. All rights reserved.

Mailbox Common

Mailbox Common

Types defined for mailbox.

Mailbox protocol is designed for devices that can't be online on the network all the time. The most common example for this is a sleepy end device.

Mailbox clients and the server can submit messages into the mailbox, which is stored in RAM on the mailbox server. Clients can then query the mailbox server for available messages.

The mailbox server will notify clients who submit messages when a message was delivered or when it couldn't be delivered due to an error.

Mailbox uses a plugin-configurable protocol endpoint, which is 15 by default.

The server can also configure the size of the mailbox (in number of packets, 25 by default) and the packet timeout, after which the server drops the message and notifies the source of the error.

The mailbox protocol uses standard data messages, so in case of sleepy end devices, it will use the indirect queue. This means that if a sleepy end device sends a request to a mailbox server, the end device should poll for the response.

Note

- Mailbox is not available in MAC mode due to the lack of endpoints.

Enumerations

enum EmberAfMailboxStatus {

EMBER_MAILBOX_STATUS_SUCCESS = 0×00
EMBER_MAILBOX_STATUS_INVALID_CALL = 0×01
EMBER_MAILBOX_STATUS_BUSY = 0×02
EMBER_MAILBOX_STATUS_STACK_ERROR = 0×03
EMBER_MAILBOX_STATUS_INVALID_ADDRESS = 0×04
EMBER_MAILBOX_STATUS_MESSAGE_TOO_LONG = 0×05
EMBER_MAILBOX_STATUS_MESSAGE_TABLE_FULL = 0×06
EMBER_MAILBOX_STATUS_MESSAGE_NO_BUFFERS = 0×07
EMBER_MAILBOX_STATUS_MESSAGE_NO_RESPONSE = 0×08
EMBER_MAILBOX_STATUS_MESSAGE_TIMED_OUT = 0×09
EMBER_MAILBOX_STATUS_MESSAGE_NO_DATA = 0×0A

}

Mailbox return status codes.

Enumeration Documentation

EmberAfMailboxStatus

> EmberAfMailboxStatus

Mailbox return status codes.

Enumerator	
EMBER_MAILBOX_STATUS_SUCCESS	The generic "no error" message.
EMBER_MAILBOX_STATUS_INVALID_CALL	Indicates invalid message pointer or length.

Copyright © 2025 Silicon Laboratories. All rights reserved.

EMBER_MAILBOX_STATUS_BUSY	Indicates that the local mailbox implementation is busy performing a previously requested task.
EMBER_MAILBOX_STATUS_STACK_ERROR	Indicates that the Connect stack returned an error to the mailbox plugin.
EMBER_MAILBOX_STATUS_INVALID_ADDRESS	Indicates that an address passed to the API was invalid.
EMBER_MAILBOX_STATUS_MESSAGE_TOO_LONG	Indicates that the message passed to the plugin was too long.
EMBER_MAILBOX_STATUS_MESSAGE_TABLE_FULL	Indicates that the message table on the server is full.
EMBER_MAILBOX_STATUS_MESSAGE_NO_BUFFERS	Indicates that the server was unable to allocate memory from the heap (See Memory Buffer for details).
EMBER_MAILBOX_STATUS_MESSAGE_NO_RESPONSE	Indicates that the mailbox server did not respond to the client's request.
EMBER_MAILBOX_STATUS_MESSAGE_TIMED_OUT	Indicates that the message timed out on the mailbox server.
EMBER_MAILBOX_STATUS_MESSAGE_NO_DATA	Indicates that there are no pending messages on the server to this device

Copyright © 2025 Silicon Laboratories. All rights reserved.

Ota Unicast Bootloader Client Plugin

Ota Unicast Bootloader Client Plugin

APIs/callbacks for ota-unicast-bootloader clients.

OTA bootloading plugins are usable to send firmware images Over The Air when the application is running. When the firmware is downloaded to a device, a bootloader can be started to replace the application in the flash to the one just downloaded.

All Connect bootloader related code relies on the Gecko Bootloader for bootloading and it must be installed on the device for these plugins to work. For details on the Gecko Bootloader, see UG266.

The Unicast OTA plugins implement the OTA download operation in a unicast, addressed way, so only a single client can be addressed from a server in an OTA session, and downloading images to multiple devices will require the server to send the image multiple times. Communication relies on standard unicast data messages, which also means that the routing provided by the Connect stack is availble.

Although bootloading sleepy end devices is theoretically possible with polling, it is not very effective, and it's probably simpler to reconnect as a normal end device while the OTA is active.

Unicast OTA uses a plugin configurable endpoint, which is 13 by default.

Security can be also enabled as plugin configuration on the server, as well as the interval of the messages. The client has a timeout plugin configuration after which it stops the OTA session with an error.

See UG235.06 for further details.

Note

- OTA Unicast Bootloading plugins are not available in MAC mode due to the lack of endpoints.

See ota-unicast-bootloader-client.h and ota-unicast-bootloader-client.c for source code.

Callbacks

bool emberAfPluginOtaUnicastBootloaderClientNewIncomingImageCallback(EmberNodeId serverId, uint8_t imageTag, uint32_t imageSize, uint32_t *startIndex)

A callback invoked when the client starts receiving a new image. The application can choose to start receiving the image or ignore it. If the application chooses to receive the image, other images sent out by other servers shall be ignored until the client completes the download.

void emberAfPluginOtaUnicastBootloaderClientIncomingImageSegmentCallback(EmberNodeId serverId, uint32_t startIndex, uint32_t endIndex, uint8_t imageTag, uint8_t *imageSegment)

A callback invoked when an image segment that is part of an image the application chose to download was received.

void emberAfPluginOtaUnicastBootloaderClientImageDownloadCompleteCallback(EmberAfOtaUnicastBootloaderStatus status, uint8_t imageTag, uint32_t imageSize)

A callback invoked to indicate that an image download has completed.

bool emberAfPluginOtaUnicastBootloaderClientIncomingRequestBootloadCallback(EmberNodeId serverId, uint8_t imageTag, uint32_t bootloadDelayMs)

A callback invoked to indicate that a server has requested to perform a bootload operation at a certain point in time in the future.

Functions

Copyright © 2025 Silicon Laboratories. All rights reserved.

EmberAfOtaUnicastBootloaderStatus	emberAfPluginOtaUnicastBootloaderClientAbortImageDownload(uint8_t imageTag) An API for aborting an ongoing image download process.

Callbacks Documentation

emberAfPluginOtaUnicastBootloaderClientNewIncomingImageCallback

> bool emberAfPluginOtaUnicastBootloaderClientNewIncomingImageCallback (EmberNodeId serverId, uint8_t imageTag, uint32_t imageSize, uint32_t * startIndex)

A callback invoked when the client starts receiving a new image. The application can choose to start receiving the image or ignore it. If the application chooses to receive the image, other images sent out by other servers shall be ignored until the client completes the download.

Parameters

Type	Direction	Argument Name	Description
EmberNodeId	[in]	serverId	The node ID of the server that initiated the new image distribution process.
uint8_t	[in]	imageTag	A 1-byte tag that identifies the incoming image.
uint32_t	[in]	imageSize	The size in bytes of the new image.
uint32_t *	[out]	startIndex	The index of the first byte at which the image download shall be started/resumed. The client can use this argument to resume a partially downloaded image. If this value is not set, it defaults to 0 (that is, the download starts at the beginning of the image). Note, this is ignored in case the server does not support download resume.

Returns

- Return **true** to accept the image or **false** to ignore it.

emberAfPluginOtaUnicastBootloaderClientIncomingImageSegmentCallback

> void emberAfPluginOtaUnicastBootloaderClientIncomingImageSegmentCallback (EmberNodeId serverId, uint32_t startIndex, uint32_t endIndex, uint8_t imageTag, uint8_t * imageSegment)

A callback invoked when an image segment that is part of an image the application chose to download was received.

Parameters

Type	Direction	Argument Name	Description
EmberNodeId	[in]	serverId	The node ID of the server that initiated the image distribution process.
uint32_t	[in]	startIndex	The index of the first byte of the passed segment.
uint32_t	[in]	endIndex	The index of the last byte of the passed segment.
uint8_t	[in]	imageTag	A 1-byte tag of the image the passed segment belongs to.
uint8_t *	[in]	imageSegment	An array containing the image segment.

emberAfPluginOtaUnicastBootloaderClientImageDownloadCompleteCallback

> void emberAfPluginOtaUnicastBootloaderClientImageDownloadCompleteCallback (EmberAfOtaUnicastBootloaderStatus status, uint8_t imageTag, uint32_t imageSize)

Copyright © 2025 Silicon Laboratories. All rights reserved.

A callback invoked to indicate that an image download has completed.

Parameters

Type	Direction	Argument Name	Description
EmberAfOtaUnicastBootloaderStatus	[in]	status	An EmberAfOtaUnicastBootloaderStatus value of: • EMBER_OTA_UNICAST_BOOTLOADER_STATUS_SUCCESS indicating t the full image corresponding to the passed tag has been received. If the case, the client previously handed all the image segments to the application using the emberAfPluginOtaUnicastBootloaderClientIncomingImageSegmentCall callback. • EMBER_OTA_UNICAST_BOOTLOADER_STATUS_FAILED indicating tha client failed to fully download the image and the download process w terminated. • EMBER_OTA_UNICAST_BOOTLOADER_STATUS_TIMEOUT indicating th the client timed out waiting for a message from the server. • EMBER_OTA_UNICAST_BOOTLOADER_STATUS_ABORTED indicating t the application aborted the ongoing image download process as resu calling the API emberAfPluginOtaUnicastBootloaderClientAbortImageDownload().
uint8_t	[in]	imageTag	A 1-byte tag of the image this callback refers to.
uint32_t	[in]	imageSize	The total size of the downloaded image in bytes. This parameter is meanin only in case the status parameter is set to EMBER_OTA_UNICAST_BOOTLOADER_STATUS_SUCCESS.

emberAfPluginOtaUnicastBootloaderClientIncomingRequestBootloadCallback

> bool emberAfPluginOtaUnicastBootloaderClientIncomingRequestBootloadCallback (EmberNodeId serverId, uint8_t imageTag, uint32_t bootloadDelayMs)

A callback invoked to indicate that a server has requested to perform a bootload operation at a certain point in time in the future.

Parameters

Type	Direction	Argument Name	Description
EmberNodeId	[in]	serverId	The ID of the server the request came from.
uint8_t	[in]	imageTag	A 1-byte tag of the image this callback refers to.
uint32_t	[in]	bootloadDelayMs	The delay in milliseconds after which the client has been requested to perform a bootload operation.

Returns

- Return **true** if the application accepted the request of bootloading the specified image at the requested time, **false** otherwise.

Function Documentation

emberAfPluginOtaUnicastBootloaderClientAbortImageDownload

Copyright © 2025 Silicon Laboratories. All rights reserved.

> EmberAfOtaUnicastBootloaderStatus emberAfPluginOtaUnicastBootloaderClientAbortImageDownload (uint8_t imageTag)

An API for aborting an ongoing image download process.

Parameters

Type	Direction	Argument Name	Description
uint8_t	[in]	imageTag	A 1-byte tag that identifies the image the client should no longer download.

Returns

- An EmberAfOtaUnicastBootloaderStatus value of:
  - EMBER_OTA_UNICAST_BOOTLOADER_STATUS_SUCCESS If the ongoing image download process was successfully aborted.
  - EMBER_OTA_UNICAST_BOOTLOADER_STATUS_INVALID_CALL If the client was not currently involved in an image download process or it was currently downloading an image with a different tag.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Ota Unicast Bootloader Server Plugin

Ota Unicast Bootloader Server Plugin

Macros and APIs for ota-unicast-bootloader server.

OTA bootloading plugins are usable to send firmware images Over The Air when the application is running. When the firmware is downloaded to a device, a bootloader can be started to replace the application in the flash to the one just downloaded.

All Connect bootloader related code relies on the Gecko Bootloader for bootloading and it must be installed on the device for these plugins to work. For details on the Gecko Bootloader, see UG266.

The Unicast OTA plugins implement the OTA download operation in a unicast, addressed way, so only a single client can be addressed from a server in an OTA session, and downloading images to multiple devices will require the server to send the image multiple times. Communication relies on standard unicast data messages, which also means that the routing provided by the Connect stack is availble.

Although bootloading sleepy end devices is theoretically possible with polling, it is not very effective, and it's probably simpler to reconnect as a normal end device while the OTA is active.

Unicast OTA uses a plugin configurable endpoint, which is 13 by default.

Security can be also enabled as plugin configuration on the server, as well as the interval of the messages. The client has a timeout plugin configuration after which it stops the OTA session with an error.

See UG235.06 for further details.

Note

- OTA Unicast Bootloading plugins are not available in MAC mode due to the lack of endpoints.

See ota-unicast-bootloader-server.h and ota-unicast-bootloader-server.c for source code.

Callbacks

bool	emberAfPluginOtaUnicastBootloaderServerGetImageSegmentCallback(uint32_t startIndex, uint32_t endIndex, uint8_t imageTag, uint8_t *imageSegment) A callback invoked during an image distribution process to retrieve a contiguous segment of the image being distributed.
void	emberAfPluginOtaUnicastBootloaderServerImageDistributionCompleteCallback(EmberAfOtaUnicastBootloaderStatus status) A callback invoked when the image distribution process is terminated.
void	emberAfPluginOtaUnicastBootloaderServerRequestTargetBootloadCompleteCallback(EmberAfOtaUnicastBootloaderStatus status) A callback invoked when a bootload request process has completed.

Functions

EmberAfOtaUnicastBootloaderStatus	emberAfPluginOtaUnicastBootloaderServerInitiateImageDistribution(EmberNodeId targetId, uint32_t imageSize, uint8_t imageTag) Initiate the image distribution process.

Copyright © 2025 Silicon Laboratories. All rights reserved.

EmberAfOtaUnicastBootloaderStatus	emberAfPluginUnicastBootloaderServerInitiateRequestTargetBootload(uint32_t bootloadDelayMs, uint8_t imageTag, EmberNodeId targetId) Request a target device to initiate the bootload of a received image at some point in the future.
EmberAfOtaUnicastBootloaderStatus	emberAfPluginOtaUnicastBootloaderServerAbortCurrentProcess(void) Abort the ongoing process, such as image distribution or bootload request.

Macros

#define	EMBER_AF_PLUGIN_OTA_UNICAST_BOOTLOADER_SERVER_MAX_STACK_ERRORS 8 The number of consecutive stack message submission errors or stack related errors such as CSMA failures after which the plugin gives up.
#define	EMBER_AF_PLUGIN_OTA_UNICAST_BOOTLOADER_SERVER_MAX_UNICAST_ERRORS 4 The number of consecutive unicast attempts after which a target is declared unreachable. Legal values for this are in the [0,7] range.
#define	EMBER_AF_PLUGIN_OTA_UNICAST_BOOTLOADER_SERVER_RESPONSE_TIMEOUT_MS 250 The time in milliseconds after which the server gives up waiting for a response from a client.

Callbacks Documentation

emberAfPluginOtaUnicastBootloaderServerGetImageSegmentCallback

> bool emberAfPluginOtaUnicastBootloaderServerGetImageSegmentCallback (uint32_t startIndex, uint32_t endIndex, uint8_t imageTag, uint8_t * imageSegment)

A callback invoked during an image distribution process to retrieve a contiguous segment of the image being distributed.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	startIndex	The index of the first byte the application should copy into the passed array.
uint32_t	[in]	endIndex	The index of the last byte the application should copy into the passed array.
uint8_t	[in]	imageTag	A 1-byte tag of the image for which a segment is requested.
uint8_t *	[out]	imageSegment	An array of (endIndex - startIndex + 1) length to which the application should copy the requested image segment.

Returns

- A boolean indicating whether the application successfully copied the requested bytes into the passed array. If the application returns **false**, the server will abort the ongoing distribution process.

emberAfPluginOtaUnicastBootloaderServerImageDistributionCompleteCallback

> void emberAfPluginOtaUnicastBootloaderServerImageDistributionCompleteCallback (EmberAfOtaUnicastBootloaderStatus status)

A callback invoked when the image distribution process is terminated.

Parameters

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
EmberAfOtaUnicastBootloaderStatus	[in]	status	An EmberAfOtaUnicastBootloaderStatus value of: • EMBER_OTA_UNICAST_BOOTLOADER_STATUS_SUCCESS if the targe that the full image is received. • EMBER_OTA_UNICAST_BOOTLOADER_STATUS_DATA_UNDERFLOW if application failed to supply the requested image segments. • EMBER_OTA_UNICAST_BOOTLOADER_STATUS_STACK_ERROR if the encountered multiple consecutive transmission errors. The Server give image distribution process if EMBER_AF_PLUGIN_OTA_UNICAST_BOOTLOADER_SERVER_MAX_STA consecutive transmission errors are encountered. • EMBER_OTA_UNICAST_BOOTLOADER_STATUS_FAILED if the distribut terminated prematurely because the target can't be reached. • EMBER_OTA_UNICAST_BOOTLOADER_STATUS_UNREACHABLE if the not establish communication with the target client. • EMBER_OTA_UNICAST_BOOTLOADER_STATUS_ABORTED if the applic aborted the current image distribution process. • EMBER_OTA_UNICAST_BOOTLOADER_STATUS_REFUSED if the client the image.

emberAfPluginOtaUnicastBootloaderServerRequestTargetBootloadCompleteCallback

```
void emberAfPluginOtaUnicastBootloaderServerRequestTargetBootloadCompleteCallback
(EmberAfOtaUnicastBootloaderStatus status)
```

A callback invoked when a bootload request process has completed.

Parameters

Type	Direction	Argument Name	Description
EmberAfOtaUnicastBootloaderStatus	[in]	status	An EmberAfOtaUnicastBootloaderStatus value of: • EMBER_OTA_UNICAST_BOOTLOADER_STATUS_SUCCESS if the targe requested to perform a bootload. • EMBER_OTA_UNICAST_BOOTLOADER_STATUS_STACK_ERROR if the encountered multiple consecutive transmission errors. The Server give bootload request process if EMBER_AF_PLUGIN_OTA_UNICAST_BOOTLOADER_SERVER_MAX_STA consecutive transmission errors are encountered. • EMBER_OTA_UNICAST_BOOTLOADER_STATUS_ABORTED if the applic aborted the current bootload request process. • EMBER_OTA_UNICAST_BOOTLOADER_STATUS_UNREACHABLE if the not establish communication with the target client.

Function Documentation

emberAfPluginOtaUnicastBootloaderServerInitiateImageDistribution

```
EmberAfOtaUnicastBootloaderStatus emberAfPluginOtaUnicastBootloaderServerInitiateImageDistribution (EmberNodeId
targetId, uint32_t imageSize, uint8_t imageTag)
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Initiate the image distribution process.

Parameters

Type	Direction	Argument Name	Description
EmberNodeId	[in]	targetId	The node ID of the target.
uint32_t	[in]	imageSize	The image size in bytes to be distributed.
uint8_t	[in]	imageTag	A 1-byte tag that will be embedded in the server-to-client over-the-air messages. The application can use the image tag for versioning purposes and/or for distinguishing between different image types.

Returns

- An EmberAfOtaUnicastBootloaderStatus value of:
  - EMBER_OTA_UNICAST_BOOTLOADER_STATUS_BUSY if an image distribution is already in progress
  - EMBER_OTA_UNICAST_BOOTLOADER_STATUS_INVALID_CALL if the given target or the image size is invalid
  - EMBER_OTA_UNICAST_BOOTLOADER_STATUS_SUCCESS if the image distribution was successfully initiated.

emberAfPluginUnicastBootloaderServerInitiateRequestTargetBootload

> EmberAfOtaUnicastBootloaderStatus emberAfPluginUnicastBootloaderServerInitiateRequestTargetBootload (uint32_t bootloadDelayMs, uint8_t imageTag, EmberNodeId targetId)

Request a target device to initiate the bootload of a received image at some point in the future.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	bootloadDelayMs	The delay in milliseconds after which the target should perform an image bootload.
uint8_t	[in]	imageTag	A 1-byte tag that identifies the image to be bootloaded at the target device.
EmberNodeId	[in]	targetId	The node ID of the target.

Returns

- An EmberAfOtaUnicastBootloaderStatus value of:
  - EMBER_OTA_UNICAST_BOOTLOADER_STATUS_SUCCESS if the plugin successfully started the process to request a target and initiate a bootload. If this is the case, the corresponding callback emberAfPluginOtaUnicastBootloaderServerRequestTargetBootloadCompleteCallback() is invoked when the request process is completed.
  - EMBER_OTA_UNICAST_BOOTLOADER_STATUS_INVALID_CALL if some of the passed parameters are invalid.
  - EMBER_OTA_UNICAST_BOOTLOADER_STATUS_BUSY if the server is currently involved in another over-the-air process.

emberAfPluginOtaUnicastBootloaderServerAbortCurrentProcess

> EmberAfOtaUnicastBootloaderStatus emberAfPluginOtaUnicastBootloaderServerAbortCurrentProcess (void)

Abort the ongoing process, such as image distribution or bootload request.

Parameters

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
void	N/A		

Returns

- An EmberAfOtaUnicastBootloaderStatus value of:
  - EMBER_OTA_UNICAST_BOOTLOADER_STATUS_SUCCESS if the current ongoing process was successfully aborted.
  - EMBER_OTA_UNICAST_BOOTLOADER_STATUS_INVALID_CALL if the server is not currently involved in any process.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Ota Unicast Bootloader Common

Ota Unicast Bootloader Common

Macros and types defined for ota-unicast-bootloaders.

OTA bootloading plugins are usable to send firmware images Over The Air when the application is running. When the firmware is downloaded to a device, a bootloader can be started to replace the application in the flash to the one just downloaded.

All Connect bootloader related code relies on the Gecko Bootloader for bootloading and it must be installed on the device for these plugins to work. For details on the Gecko Bootloader, see UG266.

The Unicast OTA plugins implement the OTA download operation in a unicast, addressed way, so only a single client can be addressed from a server in an OTA session, and downloading images to multiple devices will require the server to send the image multiple times. Communication relies on standard unicast data messages, which also means that the routing provided by the Connect stack is availble.

Although bootloading sleepy end devices is theoretically possible with polling, it is not very effective, and it's probably simpler to reconnect as a normal end device while the OTA is active.

Unicast OTA uses a plugin configurable endpoint, which is 13 by default.

Security can be also enabled as plugin configuration on the server, as well as the interval of the messages. The client has a timeout plugin configuration after which it stops the OTA session with an error.

See UG235.06 for further details.

Note

- OTA Unicast Bootloading plugins are not available in MAC mode due to the lack of endpoints.

Enumerations

enum EmberAfOtaUnicastBootloaderStatus {

EMBER_OTA_UNICAST_BOOTLOADER_STATUS_SUCCESS = 0x00
EMBER_OTA_UNICAST_BOOTLOADER_STATUS_INVALID_CALL = 0x01
EMBER_OTA_UNICAST_BOOTLOADER_STATUS_BUSY = 0x02
EMBER_OTA_UNICAST_BOOTLOADER_STATUS_DATA_UNDERFLOW = 0x03
EMBER_OTA_UNICAST_BOOTLOADER_STATUS_STACK_ERROR = 0x04
EMBER_OTA_UNICAST_BOOTLOADER_STATUS_TIMEOUT = 0x05
EMBER_OTA_UNICAST_BOOTLOADER_STATUS_FAILED = 0x06
EMBER_OTA_UNICAST_BOOTLOADER_STATUS_ABORTED = 0x07
EMBER_OTA_UNICAST_BOOTLOADER_STATUS_REFUSED = 0x08
EMBER_OTA_UNICAST_BOOTLOADER_STATUS_UNREACHABLE = 0x09

}

OTA Unicast Bootloader return status codes.

Enumeration Documentation

EmberAfOtaUnicastBootloaderStatus

> EmberAfOtaUnicastBootloaderStatus

Copyright © 2025 Silicon Laboratories. All rights reserved.

OTA Unicast Bootloader return status codes.

Enumerator	
EMBER_OTA_UNICAST_BOOTLOADER_STATUS_SUCCESS	The generic "no error" message.
EMBER_OTA_UNICAST_BOOTLOADER_STATUS_INVALID_CALL	Indicates that some parameters are invalid, or the OTA server/client is not in the expected state for the call.
EMBER_OTA_UNICAST_BOOTLOADER_STATUS_BUSY	Indicates that the OTA server is busy performing another OTA task.
EMBER_OTA_UNICAST_BOOTLOADER_STATUS_DATA_UNDERFLOW	Indicates that the application did not provided the requested data to the OTA server.
EMBER_OTA_UNICAST_BOOTLOADER_STATUS_STACK_ERROR	Indicates that the Connect stack returned an error to the OTA server.
EMBER_OTA_UNICAST_BOOTLOADER_STATUS_TIMEOUT	Indicates that the image download is timed out on the OTA client.
EMBER_OTA_UNICAST_BOOTLOADER_STATUS_FAILED	Indicates that the image download is failed because the connection between the client and server is lost.
EMBER_OTA_UNICAST_BOOTLOADER_STATUS_ABORTED	Indicates that the application is aborted the current OTA process.
EMBER_OTA_UNICAST_BOOTLOADER_STATUS_REFUSED	Indicates that the client refused the image.
EMBER_OTA_UNICAST_BOOTLOADER_STATUS_UNREACHABLE	Indicates that the server can not establish communication with the target client.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Ota Broadcast Bootloader Client Plugin

Ota Broadcast Bootloader Client Plugin

Set of APIs for ota-broadcast-bootloader-client.

OTA bootloading plugins are usable to send firmware images Over The Air when the application is running. When the firmware is downloaded to a device, a bootloader can be started to replace the application in the flash to the one just downloaded.

All Connect bootloader-related code relies on the Gecko Bootloader for bootloading and it must be installed on the device for these plugins to work. For details on the Gecko Bootloader, see UG266.

The Broadcast OTA plugins implement the OTA download operation in broadcast, so the same image can be sent to many devices at the same time. The server however requires to know the clients downloading the image, because it implements error handling by querying all clients for missing segments, and then the server will re-broadcast those segments.

Communication relies on standard broadcast data messages, which means routing is not available, and only clients that are in the range of the server can download. However, the same device can be both client and server, i.e. after downloading the image, a client can configure itself to be a server, and provide the image to another part of the network.

Sleepy end devices cannot be addressed in broadcast, but a sleepy end device can reconnect as a normal end device while the OTA is active.

Broadcast OTA uses a plugin configurable endpoint, which is 14 by default.

Security can be also enabled as plugin configuration on the server, as well as the interval of the messages. The client has a timeout plugin configuration after which it stops the OTA session with an error.

See UG235.06 for further details.

Note

- OTA Broadcast Bootloading plugins are not available in MAC mode due to the lack of endpoints.

See ota-broadcast-bootloader-client.h and ota-broadcast-bootloader-client.c for source code.

Callbacks

bool emberAfPluginOtaBootloaderClientNewIncomingImageCallback(EmberNodeId serverId, EmberNodeId *alternateServerId, uint8_t imageTag)

A callback invoked when the OTA Bootloader Client starts receiving a new image. The application can choose to start receiving the image or it can ignore it. If the application chooses to receive the image, other images sent out by other servers are ignored until the client completes this download.

void emberAfPluginOtaBootloaderClientIncomingImageSegmentCallback(EmberNodeId serverId, uint32_t startIndex, uint32_t endIndex, uint8_t imageTag, uint8_t *imageSegment)

A callback invoked when an image segment, that is part of an image that the application elected to download, was received on the OTA Bootloader Client.

void emberAfPluginOtaBootloaderClientImageDownloadCompleteCallback(EmberAfOtaBootloaderStatus status, uint8_t imageTag, uint32_t imageSize)

A callback invoked on an OTA Bootloader Client to indicate that an image downlaod is completed.

void emberAfPluginOtaBootloaderClientIncomingRequestStatusCallback(EmberNodeId serverId, uint8_t applicationServerStatus, uint8_t *applicationStatus)

A callback invoked on the OTA Bootloader Client to indicate that an OTA Bootloader Server has requested the status of the client device.

Copyright © 2025 Silicon Laboratories. All rights reserved.

bool	emberAfPluginOtaBootloaderClientIncomingRequestBootloadCallback(EmberNodeId serverId, uint8_t imageTag, uint32_t bootloadDelayMs, uint8_t *applicationStatus) A callback invoked by the OTA Bootloader Client plugin to indicate that an OTA Bootloader Server has requested to perform a bootload operation at a certain point in time in the future.

Functions

EmberAfOtaBootloaderStatus	emberAfPluginOtaBootloaderClientAbortImageDownload(uint8_t imageTag, uint8_t applicationErrorStatus) Abort an ongoing image download process.

Callbacks Documentation

emberAfPluginOtaBootloaderClientNewIncomingImageCallback

> bool emberAfPluginOtaBootloaderClientNewIncomingImageCallback (EmberNodeId serverId, EmberNodeId * alternateServerId, uint8_t imageTag)

A callback invoked when the OTA Bootloader Client starts receiving a new image. The application can choose to start receiving the image or it can ignore it. If the application chooses to receive the image, other images sent out by other servers are ignored until the client completes this download.

Parameters

Type	Direction	Argument Name	Description
EmberNodeId	[in]	serverId	The node ID of the server that initiated the new image distribution process.
EmberNodeId *	[out]	alternateServerId	This node ID can be set by the application to include a well-known alternate server. If this is set to a valid address, the client allows segments also from this alternate server. If this is set to EMBER_BROADCAST_ADDRESS, the client accepts segments with the same image tag from any server.
uint8_t	[in]	imageTag	A 1-byte tag that identifies the incoming image.

Returns

- Return **true** to accept the image or **false** to ignore it.

emberAfPluginOtaBootloaderClientIncomingImageSegmentCallback

> void emberAfPluginOtaBootloaderClientIncomingImageSegmentCallback (EmberNodeId serverId, uint32_t startIndex, uint32_t endIndex, uint8_t imageTag, uint8_t * imageSegment)

A callback invoked when an image segment, that is part of an image that the application elected to download, was received on the OTA Bootloader Client.

Parameters

Type	Direction	Argument Name	Description
EmberNodeId	[in]	serverId	The node ID of the server that initiated the image distribution process.
uint32_t	[in]	startIndex	The index of the first byte of the passed segment.
uint32_t	[in]	endIndex	The index of the last byte of the passed segment.
uint8_t	[in]	imageTag	A 1-byte tag of the image the passed segment belongs to.
uint8_t *	[in]	imageSegment	An array containing the image segment.

Copyright © 2025 Silicon Laboratories. All rights reserved.

emberAfPluginOtaBootloaderClientImageDownloadCompleteCallback

> void emberAfPluginOtaBootloaderClientImageDownloadCompleteCallback (EmberAfOtaBootloaderStatus status, uint8_t imageTag, uint32_t imageSize)

A callback invoked on an OTA Bootloader Client to indicate that an image downlaod is completed.

Parameters

Type	Direction	Argument Name	Description
EmberAfOtaBootloaderStatus	[in]	status	An EmberAfOtaBootloaderStatus value of: • EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_SUCCESS indicating that the full image corresponding to the passed tag has been received. If this is the case, the client previously handed all the image segments to the application using the emberAfPluginOtaBootloaderClientIncomingImageSegmentCallback() callback. • EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_FAILED indicating that the client failed to fully download the image and the download process was terminated. • EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_TIMEOUT indicating that the client timed out waiting for a message from the server. • EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_ABORTED indicating that the application aborted the ongoing image download process as result of calling the API emberAfPluginOtaBootloaderClientAbortImageDownload().
uint8_t	[in]	imageTag	A 1-byte tag of the image this callback refers to.
uint32_t	[in]	imageSize	The total size of the downloaded image in bytes. This parameter is meaningful only in case the status parameter is set to EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_SUCCESS.

emberAfPluginOtaBootloaderClientIncomingRequestStatusCallback

> void emberAfPluginOtaBootloaderClientIncomingRequestStatusCallback (EmberNodeId serverId, uint8_t applicationServerStatus, uint8_t * applicationStatus)

A callback invoked on the OTA Bootloader Client to indicate that an OTA Bootloader Server has requested the status of the client device.

Parameters

Type	Direction	Argument Name	Description
EmberNodeId	[in]	serverId	The ID of the server the request came from.
uint8_t	[in]	applicationServerStatus	The server application status, which was set by emberAfPluginBootloaderServerInitiateRequestTargetsStatus()
uint8_t *	[out]	applicationStatus	A 1-byte status set by the client application that is reported to the server.

emberAfPluginOtaBootloaderClientIncomingRequestBootloadCallback

Copyright © 2025 Silicon Laboratories. All rights reserved.

bool emberAfPluginOtaBootloaderClientIncomingRequestBootloadCallback (EmberNodeId serverId, uint8_t imageTag, uint32_t bootloadDelayMs, uint8_t * applicationStatus)

A callback invoked by the OTA Bootloader Client plugin to indicate that an OTA Bootloader Server has requested to perform a bootload operation at a certain point in time in the future.

Parameters

Type	Direction	Argument Name	Description
EmberNodeId	[in]	serverId	The ID of the server the request came from.
uint8_t	[in]	imageTag	A 1-byte tag of the image this callback refers to.
uint32_t	[in]	bootloadDelayMs	The delay in milliseconds after which the client has been requested to perform a bootload operation.
uint8_t *	[out]	applicationStatus	A 1-byte status set by the client application that is reported to the server.

Returns

- Return **true** if the application accepted the request of bootloading the specified image at the requested time, **false** otherwise.

Function Documentation

emberAfPluginOtaBootloaderClientAbortImageDownload

EmberAfOtaBootloaderStatus emberAfPluginOtaBootloaderClientAbortImageDownload (uint8_t imageTag, uint8_t applicationErrorStatus)

Abort an ongoing image download process.

Parameters

Type	Direction	Argument Name	Description
uint8_t	[in]	imageTag	A 1-byte tag that identifies the image the client should no longer download.
uint8_t	[in]	applicationErrorStatus	A 1-byte error code reported to the server.

Returns

- An EmberAfOtaBootloaderStatus value of:
  - EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_SUCCESS If the ongoing image download process was successfully aborted.
  - EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_INVALID_CALL If the client was not currently involved in an image download process or it was currently downloading an image with a different tag.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Ota Broadcast Bootloader Server Plugin

Ota Broadcast Bootloader Server Plugin

Set of APIs for ota-broadcast-bootloader-server.

OTA bootloading plugins are usable to send firmware images Over The Air when the application is running. When the firmware is downloaded to a device, a bootloader can be started to replace the application in the flash to the one just downloaded.

All Connect bootloader-related code relies on the Gecko Bootloader for bootloading and it must be installed on the device for these plugins to work. For details on the Gecko Bootloader, see UG266.

The Broadcast OTA plugins implement the OTA download operation in broadcast, so the same image can be sent to many devices at the same time. The server however requires to know the clients downloading the image, because it implements error handling by querying all clients for missing segments, and then the server will re-broadcast those segments.

Communication relies on standard broadcast data messages, which means routing is not available, and only clients that are in the range of the server can download. However, the same device can be both client and server, i.e. after downloading the image, a client can configure itself to be a server, and provide the image to another part of the network.

Sleepy end devices cannot be addressed in broadcast, but a sleepy end device can reconnect as a normal end device while the OTA is active.

Broadcast OTA uses a plugin configurable endpoint, which is 14 by default.

Security can be also enabled as plugin configuration on the server, as well as the interval of the messages. The client has a timeout plugin configuration after which it stops the OTA session with an error.

See UG235.06 for further details.

Note

- OTA Broadcast Bootloading plugins are not available in MAC mode due to the lack of endpoints.

See ota-broadcast-bootloader-server.h and ota-broadcast-bootloader-server.c for source code.

Callbacks

bool emberAfPluginOtaBootloaderServerGetImageSegmentCallback(uint32_t startIndex, uint32_t endIndex, uint8_t imageTag, uint8_t *imageSegment)

A callback invoked on the OTA Bootloader Server during an image distribution process to retrieve a contiguous segment of the image being distributed.

void emberAfPluginOtaBootloaderServerImageDistributionCompleteCallback(EmberAfOtaBootloaderStatus status)

A callback invoked on the OTA Bootloader Server when the image distribution process is terminated. The application can use the emberAfPluginBootloaderServerGetTargetStatus() API to retrieve the status reported by each target device.

void emberAfPluginBootloaderServerRequestTargetsStatusCompleteCallback(EmberAfOtaBootloaderStatus status)

A callback invoked on the OTA Bootloader Server when bootload request process has completed. Within this callback, the application should use the emberAfPluginBootloaderServerGetTargetStatus() API to retrieve the status and application status reported by each target.

Copyright © 2025 Silicon Laboratories. All rights reserved.

void	emberAfPluginBootloaderServerRequestTargetsBootloadCompleteCallback(EmberAfOtaBootloaderStatus status) A callback invoked on the OTA Bootloader Server when a bootload request process has completed. Within this callback, the application should use the emberAfPluginBootloaderServerGetTargetStatus() API to retrieve the status and the application status reported by each target.

Functions

EmberAfOtaBootloaderStatus	emberAfPluginOtaBootloaderServerInitiateImageDistribution(uint32_t imageSize, uint8_t imageTag, EmberNodeId *targetList, uint16_t targetListLength) Initiate the image distribution process. The application can use the emberAfPluginBootloaderServerGetTargetStatus() API at any time during the image distribution process to check the status of each target.
EmberAfOtaBootloaderStatus	emberAfPluginBootloaderServerInitiateRequestTargetsStatus(EmberNodeId *targetList, uint16_t targetListLength, uint8_t applicationServerStatus) Initiate the process to request the status of a set of target devices. The application can use the emberAfPluginBootloaderServerGetTargetStatus() API at any time during the target status request process to check the status of each target.
EmberAfOtaBootloaderStatus	emberAfPluginBootloaderServerInitiateRequestTargetsBootload(uint32_t bootloadDelayMs, uint8_t imageTag, EmberNodeId *targetList, uint16_t targetListLength) Start the process where a server requests a set of target devices to initiate the bootload of a received image at some point in the future. The application can use the emberAfPluginBootloaderServerGetTargetStatus() API at any time during the bootload request process to check the status of each target.
EmberAfOtaBootloaderTargetStatus	emberAfPluginBootloaderServerGetTargetStatus(EmberNodeId targetId, uint8_t *applicationTargetStatus) Retrieve the locally stored status of an individual target in the distribution list. The locally stored status can be updated by calling emberAfPluginBootloaderServerInitiateRequestTargetsStatus().
EmberAfOtaBootloaderStatus	emberAfPluginOtaBootloaderServerAbortCurrentProcess(void) Abort the ongoing process (image distribution, status request or bootload request). Note that aborting a bootload request process likely results in some targets performing the bootload while some others do not.

Macros

#define	EMBER_AF_PLUGIN_OTA_BROADCAST_BOOTLOADER_SERVER_MAX_STACK_ERRORS 8 The number of consecutive stack message submission errors or stack-related errors, such as CSMA failures, after which the plugin gives up.
#define	EMBER_AF_PLUGIN_OTA_BROADCAST_BOOTLOADER_SERVER_MAX_UNICAST_ERRORS 4 The number of consecutive unicast attempts after which a target is declared unreachable. Legal values for this are in the [0,7] range.
#define	EMBER_AF_PLUGIN_OTA_BROADCAST_BOOTLOADER_SERVER_RESPONSE_TIMEOUT_MS 250 The time in milliseconds after which the server gives up waiting for a response from a client.
#define	EMBER_AF_PLUGIN_OTA_BROADCAST_BOOTLOADER_SERVER_MAX_BROADCAST_ROUNDS 5 The maximum number of image broadcast rounds the server performs before declaring an image distribution process failed.

Callbacks Documentation

emberAfPluginOtaBootloaderServerGetImageSegmentCallback

> bool emberAfPluginOtaBootloaderServerGetImageSegmentCallback (uint32_t startIndex, uint32_t endIndex, uint8_t imageTag, uint8_t * imageSegment)

Copyright © 2025 Silicon Laboratories. All rights reserved.

A callback invoked on the OTA Bootloader Server during an image distribution process to retrieve a contiguous segment of the image being distributed.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	startIndex	The index of the first byte the application should copy into the passed array.
uint32_t	[in]	endIndex	The index of the last byte the application should copy into the passed array.
uint8_t	[in]	imageTag	A 1-byte tag of the image for which a segment is being requested.
uint8_t *	[out]	imageSegment	An array of (endIndex - startIndex + 1) length to which the application should copy the requested image segment.

Returns

- A boolean indicating whether the application successfully copied the requested bytes into the passed array. If the application returns **false**, the OTA Server plugin aborts the ongoing distribution process.

emberAfPluginOtaBootloaderServerImageDistributionCompleteCallback

```
void emberAfPluginOtaBootloaderServerImageDistributionCompleteCallback (EmberAfOtaBootloaderStatus status)
```

A callback invoked on the OTA Bootloader Server when the image distribution process is terminated. The application can use the emberAfPluginBootloaderServerGetTargetStatus() API to retrieve the status reported by each target device.

Parameters

Type	Direction	Argument Name	Description
EmberAfOtaBootloaderStatus	[in]	status	An EmberAfOtaBootloaderStatus value of: • EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_SUCCESS if all targets ha that the full image was received except for those that have been declared " • EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_DATA_UNDERFLOW if the failed to supply the requested image segments. • EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_STACK_ERROR if the serv multiple consecutive transmission errors. The Server gives up the image distr if EMBER_AF_PLUGIN_OTA_BROADCAST_BOOTLOADER_SERVER_MAX_STAC consecutive transmission errors are encountered. • EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_FAILED if the distribution terminated prematurely because all targets have been declared unreachable • EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_TIMEOUT if the server pe allowable broadcast rounds and there are still missing segments at one or mo maximum allowable rounds are defined by EMBER_AF_PLUGIN_OTA_BROADCAST_BOOTLOADER_SERVER_MAX_BROAD • EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_ABORTED if the applicati current image distribution process.

emberAfPluginBootloaderServerRequestTargetsStatusCompleteCallback

```
void emberAfPluginBootloaderServerRequestTargetsStatusCompleteCallback (EmberAfOtaBootloaderStatus status)
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

A callback invoked on the OTA Bootloader Server when bootload request process has completed. Within this callback, the application should use the emberAfPluginBootloaderServerGetTargetStatus() API to retrieve the status and application status reported by each target.

Parameters

Type	Direction	Argument Name	Description
EmberAfOtaBootloaderStatus	[in]	status	An EmberAfOtaBootloaderStatus value of: • EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_SUCCESS if all the target been queried for their status. Notice that some targets might have been dec unreachable. • EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_STACK_ERROR if the serv encountered multiple consecutive transmission errors. The Server gives up th targets status request process if EMBER_AF_PLUGIN_OTA_BROADCAST_BOOTLOADER_SERVER_MAX_STACK_ consecutive transmission errors are encountered. • EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_ABORTED if the applicati aborted the current targets status request process.

emberAfPluginBootloaderServerRequestTargetsBootloadCompleteCallback

```
void emberAfPluginBootloaderServerRequestTargetsBootloadCompleteCallback (EmberAfOtaBootloaderStatus status)
```

A callback invoked on the OTA Bootloader Server when a bootload request process has completed. Within this callback, the application should use the emberAfPluginBootloaderServerGetTargetStatus() API to retrieve the status and the application status reported by each target.

Parameters

Type	Direction	Argument Name	Description
EmberAfOtaBootloaderStatus	[in]	status	An EmberAfOtaBootloaderStatus value of: • EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_SUCCESS if all targets ha requested to perform a bootload. Notice that some targets might have been declared unreachable. • EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_STACK_ERROR if the serv encountered multiple consecutive transmission errors. The Server gives up th bootload request process if EMBER_AF_PLUGIN_OTA_BROADCAST_BOOTLOADER_SERVER_MAX_STACK_ consecutive transmission errors are encountered. • EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_ABORTED if the applicati aborted the current bootload request process.

Function Documentation

emberAfPluginOtaBootloaderServerInitiateImageDistribution

```
EmberAfOtaBootloaderStatus emberAfPluginOtaBootloaderServerInitiateImageDistribution (uint32_t imageSize, uint8_t imageTag, EmberNodeId * targetList, uint16_t targetListLength)
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Initiate the image distribution process. The application can use the emberAfPluginBootloaderServerGetTargetStatus() API at any time during the image distribution process to check the status of each target.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	imageSize	The image size in bytes to be distributed.
uint8_t	[in]	imageTag	A 1-byte tag that will be embedded in the server-to-client over-the-air messages. The application can use the image tag for versioning purposes and/or for distinguishing between different image types.
EmberNodeId *	[out]	targetList	An array of EmberNodeId indicating the node IDs of the target devices.
uint16_t	[in]	targetListLength	The length of the passed `targetList`

Returns

- An EmberAfOtaBootloaderStatus value of:
  - EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_SUCCESS if the image distribution is successfully initiated. If this is the case, the emberAfPluginOtaBootloaderServerImageDistributionCompleteCallback() callback is invoked when the distribution process terminates.
  - EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_INVALID_CALL if some of the passed parameters are invalid.
  - EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_BUSY if the server is already performing another image distribution or some other over-the-air process.
  - EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_NO_BUFFERS if the server can't allocate memory from the heap to store the passed target list. (See Memory Buffer for details).

emberAfPluginBootloaderServerInitiateRequestTargetsStatus

> EmberAfOtaBootloaderStatus emberAfPluginBootloaderServerInitiateRequestTargetsStatus (EmberNodeId * targetList, uint16_t targetListLength, uint8_t applicationServerStatus)

Initiate the process to request the status of a set of target devices. The application can use the emberAfPluginBootloaderServerGetTargetStatus() API at any time during the target status request process to check the status of each target.

Parameters

Type	Direction	Argument Name	Description
EmberNodeId *	[in]	targetList	An array of EmberNodeId indicating the node IDs of the target devices that are queried for their status.
uint16_t	[in]	targetListLength	The length of the passed `targetlist` .
uint8_t	[in]	applicationServerStatus	The application can set a status here which will be sent to the clients in emberAfPluginOtaBootloaderClientIncomingRequestStatusCallback()

Returns

- An EmberAfOtaBootloaderStatus value of:
  - EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_SUCCESS if the plugin successfully initiated the process of requesting the status of a set of targets. If this is the case, the corresponding callback emberAfPluginBootloaderServerRequestTargetsStatusCompleteCallback() is invoked when the request process to all targets completes.
  - EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_INVALID_CALL if some of the passed parameters are invalid.
  - EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_BUSY if the server is currently involved in another over-the-air process.

Copyright © 2025 Silicon Laboratories. All rights reserved.

emberAfPluginBootloaderServerInitiateRequestTargetsBootload

> EmberAfOtaBootloaderStatus emberAfPluginBootloaderServerInitiateRequestTargetsBootload (uint32_t bootloadDelayMs, uint8_t imageTag, EmberNodeId * targetList, uint16_t targetListLength)

Start the process where a server requests a set of target devices to initiate the bootload of a received image at some point in the future. The application can use the emberAfPluginBootloaderServerGetTargetStatus() API at any time during the bootload request process to check the status of each target.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	bootloadDelayMs	The delay in milliseconds after which all the targets should perform an image bootload.
uint8_t	[in]	imageTag	A 1-byte tag that identifies the image to be bootloaded at the target devices.
EmberNodeId *	[in]	targetList	An array of EmberNodeId indicating the node IDs of the target devices that is requested to bootload an image.
uint16_t	[in]	targetListLength	The length of the passed `targetlist` .

Returns

- An EmberAfOtaBootloaderStatus value of:
  - EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_SUCCESS if the plugin successfully initiated the process of requesting a set of targets to initiate a bootload. If this is the case, the corresponding callback emberAfPluginBootloaderServerRequestTargetsBootloadCompleteCallback() shall be invoked when the request process to all targets has completed.
  - EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_INVALID_CALL if some of the passed parameters are invalid.
  - EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_BUSY if the server is currently involved in another over-the-air process.

emberAfPluginBootloaderServerGetTargetStatus

> EmberAfOtaBootloaderTargetStatus emberAfPluginBootloaderServerGetTargetStatus (EmberNodeId targetId, uint8_t * applicationTargetStatus)

Retrieve the locally stored status of an individual target in the distribution list. The locally stored status can be updated by calling emberAfPluginBootloaderServerInitiateRequestTargetsStatus().

Parameters

Type	Direction	Argument Name	Description
EmberNodeId	[in]	targetId	The node ID of the target device whose status is being requested.
uint8_t *	[out]	applicationTargetStatus	The application status reported by the client side application. This parameter is valid only for certain return status codes (see return status documentation).

Returns

- An EmberAfOtaBootloaderTargetStatus value of:
  - EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_INVALID if the passed node ID does not appear in the current server target list of the current ongoing process or if there is no current ongoing process.
  - EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_UNREACHABLE if the target has not responded to any of the server's unicast messages.
  - EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_DISTRIBUTION_COMPLETED if the server is currently performing an image distribution process and the target confirmed that it received the full image.

Copyright © 2025 Silicon Laboratories. All rights reserved.

EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_DISTRIBUTION_ONGOING if the server is currently performing an image distribution process and the target has partially received the image and distribution is continuing.
- EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_DISTRIBUTION_REFUSED if the target has refused the current image.
- EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_DISTRIBUTION_FAILED if the server is currently performing an image distribution process and the target reported that an error was encountered.
- EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_DISTRIBUTION_ABORTED if the server is currently performing an image distribution process and the target decided to abort the image download process. In this case, the client also reports an application status. Therefore, the applicationTargetStatus parameter is valid.
- EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_STATUS_REQUEST_COMPLETED if the server is currently performing a target status request process and the target has responded to the server's inquiry. In this case, the client also reports an application status. Therefore, the applicationTargetStatus parameter is valid.
- EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_STATUS_REQUEST_ONGOING if the server is currently performing a target status request process and the target is not yet queried by the server.
- EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_BOOTLOAD_REQUEST_ACCEPTED if the server is currently performing a bootload request process and the target has accepted to perform the requested image bootload.
- EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_BOOTLOAD_REQUEST_REFUSED if the server is currently performing a bootload request process and the target has refused to perform the requested image bootload.
- EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_BOOTLOAD_REQUEST_ONGOING if the server is currently performing a bootload request process and the target is not yet reached by the server.

emberAfPluginOtaBootloaderServerAbortCurrentProcess

> EmberAfOtaBootloaderStatus emberAfPluginOtaBootloaderServerAbortCurrentProcess (void)

Abort the ongoing process (image distribution, status request or bootload request). Note that aborting a bootload request process likely results in some targets performing the bootload while some others do not.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- An EmberAfOtaBootloaderStatus value of:
  - EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_SUCCESS if the current ongoing process was successfully aborted.
  - EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_INVALID_CALL if the server is not currently involved in any process.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Ota Broadcast Bootloader Common

Ota Broadcast Bootloader Common

Set of types defined for ota-broadcast-bootloader.

OTA bootloading plugins are usable to send firmware images Over The Air when the application is running. When the firmware is downloaded to a device, a bootloader can be started to replace the application in the flash to the one just downloaded.

All Connect bootloader-related code relies on the Gecko Bootloader for bootloading and it must be installed on the device for these plugins to work. For details on the Gecko Bootloader, see UG266.

The Broadcast OTA plugins implement the OTA download operation in broadcast, so the same image can be sent to many devices at the same time. The server however requires to know the clients downloading the image, because it implements error handling by querying all clients for missing segments, and then the server will re-broadcast those segments.

Communication relies on standard broadcast data messages, which means routing is not available, and only clients that are in the range of the server can download. However, the same device can be both client and server, i.e. after downloading the image, a client can configure itself to be a server, and provide the image to another part of the network.

Sleepy end devices cannot be addressed in broadcast, but a sleepy end device can reconnect as a normal end device while the OTA is active.

Broadcast OTA uses a plugin configurable endpoint, which is 14 by default.

Security can be also enabled as plugin configuration on the server, as well as the interval of the messages. The client has a timeout plugin configuration after which it stops the OTA session with an error.

See UG235.06 for further details.

Note

- OTA Broadcast Bootloading plugins are not available in MAC mode due to the lack of endpoints.

Enumerations

enum EmberAfOtaBootloaderStatus {

EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_SUCCESS = 0×00
EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_INVALID_CALL = 0×01
EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_BUSY = 0×02
EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_NO_BUFFERS = 0×03
EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_DATA_UNDERFLOW = 0×04
EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_STACK_ERROR = 0×05
EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_TIMEOUT = 0×06
EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_FAILED = 0×07
EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_ABORTED = 0×08

}

OTA Broadcast Bootloader return status codes.

Copyright © 2025 Silicon Laboratories. All rights reserved.

enum EmberAfOtaBootloaderTargetStatus {

EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_INVALID = 0x00
EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_UNREACHABLE = 0x01
EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_DISTRIBUTION_COMPLETED = 0x02
EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_DISTRIBUTION_ONGOING = 0x03
EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_DISTRIBUTION_FAILED = 0x04
EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_DISTRIBUTION_REFUSED = 0x05
EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_DISTRIBUTION_ABORTED = 0x06
EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_STATUS_REQUEST_COMPLETED = 0x07
EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_STATUS_REQUEST_ONGOING = 0x08
EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_BOOTLOAD_REQUEST_ACCEPTED = 0x09
EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_BOOTLOAD_REQUEST_ONGOING = 0x0A
EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_BOOTLOAD_REQUEST_REFUSED = 0x0B

}

OTA Broadcast Bootloader target status codes, returned by emberAfPluginBootloaderServerGetTargetStatus().

Macros

#define EMBER_AF_PLUGIN_OTA_BROADCAST_BOOTLOADER_INVALID_APPLICATION_TARGET_STATUS 0xFF

A value indicating that client application did not set the application level target status in any of the client callbacks.

Enumeration Documentation

EmberAfOtaBootloaderStatus

EmberAfOtaBootloaderStatus

OTA Broadcast Bootloader return status codes.

Enumerator	
EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_SUCCESS	The generic "no error" message.
EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_INVALID_CALL	Indicates that some parameters are invalid, or the OTA server/client is not in the expected state for the call
EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_BUSY	Indicates that the OTA server is busy performing another OTA task
EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_NO_BUFFERS	Indicates that the OTA server couldn't allocate memory from the heap (See Memory Buffer for details).
EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_DATA_UNDERFLOW	Indicates that the application did not provided the requested data to the OTA server
EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_STACK_ERROR	Indicates that the Connect stack returned an error to the OTA client/server.
EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_TIMEOUT	Indicates that the OTA image download is timed out during missing segment requests.
EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_FAILED	Indicates that the OTA image download is failed during the broadcast phase.
EMBER_OTA_BROADCAST_BOOTLOADER_STATUS_ABORTED	Indicates that the application aborted the OTA process.

EmberAfOtaBootloaderTargetStatus

EmberAfOtaBootloaderTargetStatus

Copyright © 2025 Silicon Laboratories. All rights reserved.

OTA Broadcast Bootloader target status codes, returned by emberAfPluginBootloaderServerGetTargetStatus().

Enumerator	
EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_INVALID	The passed node ID does not appear in the current server target list of the current ongoing process or there is no current ongoing process.
EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_UNREACHABLE	The target has not responded to any of the server's unicast messages.
EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_DISTRIBUTION_COMPLETED	The server is currently performing an image distribution process and the target confirmed that it received the full image.
EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_DISTRIBUTION_ONGOING	The server is currently performing an image distribution process and the target has partially received the image and distribution is continuing.
EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_DISTRIBUTION_FAILED	The server is currently performing an image distribution process and the target reported that an error was encountered.
EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_DISTRIBUTION_REFUSED	The target has refused the current image.
EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_DISTRIBUTION_ABORTED	The server is currently performing an image distribution process and the target decided to abort the image download process. In this case, the client also reports an application status.
EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_STATUS_REQUEST_COMPLETED	The server is currently performing a target status request process and the target has responded to the server's inquiry. In this case, the client also reports an application status.
EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_STATUS_REQUEST_ONGOING	The server is currently performing a target status request process and the target is not yet queried by the server.

Copyright © 2025 Silicon Laboratories. All rights reserved.

EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_BOOTLOAD_REQUEST_ACCEPTED	The server is currently performing a bootload request process and the target has accepted to perform the requested image bootload.
EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_BOOTLOAD_REQUEST_ONGOING	The server is currently performing a bootload request process and the target is not yet reached by the server.
EMBER_OTA_BROADCAST_BOOTLOADER_TARGET_STATUS_BOOTLOAD_REQUEST_REFUSED	The server is currently performing a bootload request process and the target has refused to perform the requested image bootload.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Poll Plugin

Poll Plugin

APIs for the poll plugin.

The Connect stack supports polling which enables (sleepy) end devices to retrieve pending messages from the parent node (coordinator or range extender).

To use this feature, the Poll plugin must be enabled on the end devices. If polling is enabled, the end device sends a data request to the parent node, which notifies the device whether a message is pending or not using the acknowledge with the pending bit cleared or set. If a message is not pending, the communication ends with the acknowledge. If a message is pending, the parent node sends a data packet containing the pending message which will be acknowledged by the end device.

For convenience, Connect supports two polling intervals, long and short, which behave the same only the polling period differs. For long polling, the period is specified in seconds while for short polling, the period is in quarter seconds. The API provides a function to easily switch between the two. The purpose of long polling is maintaining the connection between the end device and the parent.

The application will receive the polled message via the emberAfIncomingMessageCallback() function.

The poll plugin uses emberPollForData() to retrieve the pending message. If the poll plugin is enabled, using emberPollForData() is strongly not recommended.

See poll.h for source code.

Functions

void emberAfPluginPollSetShortPollInterval(uint8_t intervalQS)
Set the short poll interval.

void emberAfPluginPollSetLongPollInterval(uint16_t intervalS)
Set the long poll interval.

void emberAfPluginPollEnableShortPolling(bool enable)
Enable/disable short polling.

Function Documentation

emberAfPluginPollSetShortPollInterval

> void emberAfPluginPollSetShortPollInterval (uint8_t intervalQS)

Set the short poll interval.

Parameters

Type	Direction	Argument Name	Description
uint8_t	[in]	intervalQS	The short poll interval in quarter seconds.

emberAfPluginPollSetLongPollInterval

Copyright © 2025 Silicon Laboratories. All rights reserved.

> void emberAfPluginPollSetLongPollInterval (uint16_t intervalS)

Set the long poll interval.

Parameters

Type	Direction	Argument Name	Description
uint16_t	[in]	intervalS	The long poll interval in seconds.

emberAfPluginPollEnableShortPolling

> void emberAfPluginPollEnableShortPolling (bool enable)

Enable/disable short polling.

Parameters

Type	Direction	Argument Name	Description
bool	[in]	enable	If this parameter is true, short polling is enabled. Otherwise, the node switches back to long polling.

Copyright © 2025 Silicon Laboratories. All rights reserved.

WSTK Sensors Plugin

WSTK Sensors Plugin

Copyright © 2025 Silicon Laboratories. All rights reserved.

Connect RAIL DMP Component

Connect RAIL DMP Component

Sets of API for Connect RAIL Dynamic Multi Protocol component.

The Connect-RAIL DMP component is designed to facilitate communication using the RAIL (Radio Abstraction Interface Layer) within the Connect framework. This component allows for both transmission and reception of packets using RAIL.

Functions

void	rail_app_task_init(void)
void	rail_app_task(void *p_arg)
void	handle_os_pending(void)
void	post_os_flag(void)
sl_rail_time_t	calculate_transaction_time(sl_rail_handle_t rail_handle)

Macros

#define	SL_RAIL_SCHEDULER_PRIORITY_SEND 50 Scheduler priority for sending operations.
#define	SL_RAIL_SCHEDULER_PRIORITY_RECEIVE 100 Scheduler priority for receive operations.

Function Documentation

rail_app_task_init

> void rail_app_task_init (void)

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Initialize the RAIL DMP application task.

rail_app_task

> void rail_app_task (void * p_arg)

Parameters

Type	Direction	Argument Name	Description
void *	N/A	p_arg	Pointer to the task argument.

Copyright © 2025 Silicon Laboratories. All rights reserved.

RAIL DMP application task.

handle_os_pending

```
void handle_os_pending (void )
```

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Handle the pending OS events.

post_os_flag

```
void post_os_flag (void )
```

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Post the OS flag.

calculate_transaction_time

```
sl_rail_time_t calculate_transaction_time (sl_rail_handle_t rail_handle)
```

Parameters

Type	Direction	Argument Name	Description
sl_rail_handle_t	N/A	rail_handle	

Calculate the transaction time.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Hardware Abstraction Layer (HAL) API Reference

Hardware Abstraction Layer (HAL) API Reference

HAL function names have the following prefix conventions:

halCommon: API that is used by the EmberZNet stack and can also be called from an application. This API must be implemented. Custom applications can change the implementation of the API but its functionality must remain the same.

hal: API that is used by sample applications. Custom applications can remove this API or change its implementation as they see fit.

halStack: API used only by the EmberZNet stack. This API must be implemented and should not be directly called from any application. Custom applications can change the implementation of the API, but its functionality must remain the same.

halInternal: API that is internal to the HAL. The EmberZNet stack and applications must never call this API directly. Custom applications can change this API as they see fit. However, be careful not to impact the functionalty of any halStack or halCommon APIs.

See also hal.h.

Modules

Hardware Abstraction Layer (HAL)

Common Microcontroller Functions

Token Access

Sample APIs for Peripheral Access

System Timer Control

Symbol Timer Control

HAL Configuration

HAL Utilities

Copyright © 2025 Silicon Laboratories. All rights reserved.

Hardware Abstraction Layer (HAL)

Hardware Abstraction Layer (HAL)

The Hardware Abstraction Layer (HAL) is program code between a system's hardware and its software that provides a consistent interface for applications that can run on several different hardware platforms. To take advantage of this capability, applications should access hardware through the API provided by the HAL, rather than directly. Then, when you move to new hardware, you only need to update the HAL. In some cases, due to extreme differences in hardware, the HAL API may also change slightly to accommodate the new hardware. In these cases, the limited scope of the update makes moving the application easier with the HAL than without.

HAL function names have the following prefix conventions:

- **halCommon:** API that is used by the EmberZNet stack and can also be called from an application. This API must be implemented. Custom applications can change the implementation of the API but its functionality must remain the same.
- **hal:** API that is used by sample applications. Custom applications can remove this API or change its implementation as they see fit.
- **halStack:** API used only by the EmberZNet stack. This API must be implemented and should not be directly called from any application. Custom applications can change the implementation of the API, but its functionality must remain the same.
- **halInternal:** API that is internal to the HAL. The EmberZNet stack and applications must never call this API directly. Custom applications can change this API as they see fit. However, be careful not to impact the functionalty of any halStack or halCommon APIs.

See also hal.h.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Common Microcontroller Functions

Many of the supplied example applications use these microcontroller functions. See hal/micro/micro.h for source code.

Note

- The term SFD refers to the Start Frame Delimiter.

Many of the supplied example applications use these microcontroller functions. See hal/micro/micro-common.h for source code.

Modules

RTCCRamData

Enumerations

enum SleepModes {

SLEEPMODE_RUNNING = 0U
SLEEPMODE_IDLE = 1U
SLEEPMODE_WAKETIMER = 2U
SLEEPMODE_MAINTAINTIMER = 3U
SLEEPMODE_NOTIMER = 4U
SLEEPMODE_HIBERNATE = 5U
SLEEPMODE_RESERVED = 6U
SLEEPMODE_POWERDOWN = 7U
SLEEPMODE_POWERSAVE = 8U

}

Enumerations for the possible microcontroller sleep modes.

Typedefs

typedef uint32_t WakeEvents

typedef uint32_t WakeMask

Variables

volatile int8_t halCommonVreg1v8EnableCount

Helper variable to track the state of 1.8V regulator.

Functions

void halStackProcessBootCount(void)

Called from sli_zigbee_stack_init and provides a means for the HAL to increment a boot counter, most commonly in non-volatile memory.

uint8_t halGetResetInfo(void)

Gets information about what caused the microcontroller to reset.

const char * halGetResetString(void)

Calls halGetResetInfo() and supplies a string describing it.

Copyright © 2025 Silicon Laboratories. All rights reserved.

void halInit(void)
Initializes microcontroller-specific peripherals.

void halReboot(void)
Restarts the microcontroller and therefore everything else.

void halPowerUp(void)
Powers up microcontroller peripherals and board peripherals.

void halPowerDown(void)
Powers down microcontroller peripherals and board peripherals.

void halResume(void)
Resumes microcontroller peripherals and board peripherals.

void halSuspend(void)
Suspends microcontroller peripherals and board peripherals.

void halInternalEnableWatchDog(void)
Enables the watchdog timer.

void halInternalDisableWatchDog(uint8_t magicKey)
Disables the watchdog timer.

bool halInternalWatchDogEnabled(void)
Determines whether the watchdog has been enabled or disabled.

void halSleep(SleepModes sleepMode)
Puts the microcontroller to sleep in a specified mode.

void halSleepPreserveInts(SleepModes sleepMode)
Same as halSleep() except it preserves the current interrupt state rather than always enabling interrupts prior to returning.

void halCommonDelayMicroseconds(uint16_t us)
Blocks the current thread of execution for the specified amount of time, in microseconds.

void halCommonDisableVreg1v8(void)
Disable the 1.8V regulator. This function is to be used when the 1.8V supply is provided externally. Disabling the regulator saves current consumption. Disabling the regulator will cause ADC readings of external signals to be wrong. These exteranl signals include analog sources ADC0 thru ADC5 and VDD_PADS/4.

void halCommonEnableVreg1v8(void)
Enable the 1.8V regulator. Normally the 1.8V regulator is enabled out of reset. This function is only needed if the 1.8V regulator has been disabled and ADC conversions on external signals are needed. These exteranl signals include analog sources ADC0 thru ADC5 and VDD_PADS/4. The state of 1v8 survives deep sleep.

void halBeforeEM4(uint32_t duration, RTCCRamData input)

RTCCRamData halAfterEM4(void)

Macros

#define halGetEm2xxResetInfo halGetResetInfo
Calls ::halGetExtendedResetInfo() and translates the EM35x reset code to the corresponding value used by the EM2XX HAL. Any reset codes not present in the EM2XX are returned after being OR'ed with 0x80.

#define MICRO_DISABLE_WATCH_DOG_KEY 0xA5U
The value that must be passed as the single parameter to halInternalDisableWatchDog() in order to successfully disable the watchdog timer.

#define GPIO_MASK_SIZE 24

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
#define    GPIO_MASK 0xFFFFFF
#define    WAKE_GPIO_MASK GPIO_MASK
#define    WAKE_GPIO_SIZE GPIO_MASK_SIZE
#define    WAKE_MASK_INVALID (-1)
#define    WAKE_EVENT_SIZE WakeMask
#define    DEBUG_TOGGLE (n)
```

Enumeration Documentation

SleepModes

> SleepModes

Enumerations for the possible microcontroller sleep modes.

- SLEEPMODE_RUNNING Everything is active and running. In practice this mode is not used, but it is defined for completeness of information.
- SLEEPMODE_IDLE Only the CPU is idled. The rest of the chip continues running normally. The chip will wake from any interrupt.
- SLEEPMODE_WAKETIMER The sleep timer clock sources remain running. The RC is always running and the 32kHz XTAL depends on the board header. Wakeup is possible from both GPIO and the sleep timer. System time is maintained. The sleep timer is assumed to be configured properly for wake events.
- SLEEPMODE_MAINTAINTIMER The sleep timer clock sources remain running. The RC is always running and the 32kHz XTAL depends on the board header. Wakeup is possible from only GPIO. System time is maintained. NOTE: This mode is not available on EM2XX chips.
- SLEEPMODE_NOTIMER The sleep timer clock sources (both RC and XTAL) are turned off. Wakeup is possible from only GPIO. System time is lost.
- SLEEPMODE_HIBERNATE This maps to EM4 Hibernate on the EFM32/EFR32 devices. RAM is not retained in SLEEPMODE_HIBERNATE so waking up from this sleepmode will behave like a reset. NOTE: This mode is only available on EFM32/EFR32

Enumerator	
SLEEPMODE_RUNNING	
SLEEPMODE_IDLE	
SLEEPMODE_WAKETIMER	
SLEEPMODE_MAINTAINTIMER	
SLEEPMODE_NOTIMER	
SLEEPMODE_HIBERNATE	
SLEEPMODE_RESERVED	
SLEEPMODE_POWERDOWN	
SLEEPMODE_POWERSAVE	

Typedef Documentation

WakeEvents

> typedef uint32_t WakeEvents

Copyright © 2025 Silicon Laboratories. All rights reserved.

WakeMask

```
typedef uint32_t WakeMask
```

Variable Documentation

halCommonVreg1v8EnableCount

```
volatile int8_t halCommonVreg1v8EnableCount
```

Helper variable to track the state of 1.8V regulator.

Note

- : Only used when DISABLE_INTERNAL_1V8_REGULATOR is defined.

Function Documentation

halStackProcessBootCount

```
void halStackProcessBootCount (void )
```

Called from sli_zigbee_stack_init and provides a means for the HAL to increment a boot counter, most commonly in non-volatile memory.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

This is useful while debugging to determine the number of resets that might be seen over a period of time. Exposing this functionality allows the application to disable or alter processing of the boot counter if, for example, the application is expecting a lot of resets that could wear out non-volatile storage or some

- EmberStack Usage:\n Called from sli_zigbee_stack_init only as helpful debugging information.

  This should be left enabled by default, but this function can also be reduced to a simple return statement if boot counting is not desired.

halGetResetInfo

```
uint8_t halGetResetInfo (void )
```

Gets information about what caused the microcontroller to reset.

Parameters

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
void	N/A		

Returns

- A code identifying the cause of the reset.

halGetResetString

```
const char * halGetResetString (void )
```

Calls halGetResetInfo() and supplies a string describing it.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

- Application Usage:\n Useful for diagnostic printing of text just after program

  initialization.

Returns

- A pointer to a program space string.

halInit

```
void halInit (void )
```

Initializes microcontroller-specific peripherals.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

halReboot

```
void halReboot (void )
```

Restarts the microcontroller and therefore everything else.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

halPowerUp

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
void halPowerUp (void )
```

Powers up microcontroller peripherals and board peripherals.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

halPowerDown

```
void halPowerDown (void )
```

Powers down microcontroller peripherals and board peripherals.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

halResume

```
void halResume (void )
```

Resumes microcontroller peripherals and board peripherals.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

halSuspend

```
void halSuspend (void )
```

Suspends microcontroller peripherals and board peripherals.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

halInternalEnableWatchDog

```
void halInternalEnableWatchDog (void )
```

Enables the watchdog timer.

Parameters

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
Type	Direction	Argument Name	Description
void	N/A		

halInternalDisableWatchDog

> void halInternalDisableWatchDog (uint8_t magicKey)

Disables the watchdog timer.

Parameters

Type	Direction	Argument Name	Description
uint8_t	N/A	magicKey	A value (MICRO_DISABLE_WATCH_DOG_KEY) that enables the function.

Note

- To prevent the watchdog from being disabled accidentally, a magic key must be provided.

halInternalWatchDogEnabled

> bool halInternalWatchDogEnabled (void)

Determines whether the watchdog has been enabled or disabled.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- A bool value indicating if the watchdog is current enabled.

halSleep

> void halSleep (SleepModes sleepMode)

Puts the microcontroller to sleep in a specified mode.

Parameters

Type	Direction	Argument Name	Description
SleepModes	N/A	sleepMode	A microcontroller sleep mode

Note

- This routine always enables interrupts.

See Also

- SleepModes

Copyright © 2025 Silicon Laboratories. All rights reserved.

halSleepPreserveInts

> void halSleepPreserveInts (SleepModes sleepMode)

Same as halSleep() except it preserves the current interrupt state rather than always enabling interrupts prior to returning.

Parameters

Type	Direction	Argument Name	Description
SleepModes	N/A	sleepMode	A microcontroller sleep mode

See Also

- SleepModes

halCommonDelayMicroseconds

> void halCommonDelayMicroseconds (uint16_t us)

Blocks the current thread of execution for the specified amount of time, in microseconds.

Parameters

Type	Direction	Argument Name	Description
uint16_t	N/A	us	The specified time, in microseconds. Values should be between 1 and 65535 microseconds.

The function is implemented with cycle-counted busy loops and is intended to create the short delays required when interfacing with hardware peripherals.

The accuracy of the timing provided by this function is not specified, but a general rule is that when running off of a crystal oscillator it will be within 10us. If the micro is running off of another type of oscillator (e.g. RC) the timing accuracy will potentially be much worse.

halCommonDisableVreg1v8

> void halCommonDisableVreg1v8 (void)

Disable the 1.8V regulator. This function is to be used when the 1.8V supply is provided externally. Disabling the regulator saves current consumption. Disabling the regulator will cause ADC readings of external signals to be wrong. These exteranl signals include analog sources ADC0 thru ADC5 and VDD_PADS/4.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- : Only used when DISABLE_INTERNAL_1V8_REGULATOR is defined.

halCommonEnableVreg1v8

Copyright © 2025 Silicon Laboratories. All rights reserved.

> void halCommonEnableVreg1v8 (void)

Enable the 1.8V regulator. Normally the 1.8V regulator is enabled out of reset. This function is only needed if the 1.8V regulator has been disabled and ADC conversions on external signals are needed. These exteranl signals include analog sources ADC0 thru ADC5 and VDD_PADS/4. The state of 1v8 survives deep sleep.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- : Only used when DISABLE_INTERNAL_1V8_REGULATOR is defined.

halBeforeEM4

> void halBeforeEM4 (uint32_t duration, RTCCRamData input)

Parameters

Type	Direction	Argument Name	Description
uint32_t	N/A	duration	
RTCCRamData	N/A	input	

halAfterEM4

> RTCCRamData halAfterEM4 (void)

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Copyright © 2025 Silicon Laboratories. All rights reserved.

RTCCRamData

Public Attributes

uint32_t	outgoingNwkFrameCounter
uint32_t	incomingParentNwkFrameCounter
uint32_t	outgoingLinkKeyFrameCounter
uint32_t	incomingLinkKeyFrameCounter

Public Attribute Documentation

outgoingNwkFrameCounter

```
uint32_t RTCCRamData::outgoingNwkFrameCounter
```

incomingParentNwkFrameCounter

```
uint32_t RTCCRamData::incomingParentNwkFrameCounter
```

outgoingLinkKeyFrameCounter

```
uint32_t RTCCRamData::outgoingLinkKeyFrameCounter
```

incomingLinkKeyFrameCounter

```
uint32_t RTCCRamData::incomingLinkKeyFrameCounter
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Token Access

Token Access

The token system stores such non-volatile information as the manufacturing ID, channel number, transmit power, and various pieces of information that the application needs to be persistent between device power cycles. The token system is design to abstract implementation details and simplify interacting with differing non-volatile systems. The majority of tokens are stored in Simulated EEPROM or NVM3 (in Flash) where they can be rewritten. Manufacturing tokens are stored in dedicated regions of flash and are not designed to be rewritten.

Refer to the Tokens module for a detailed description of the token system.
Refer to the nvm3 module for a detailed description of the necessary support functions for NVM3.
Refer to token-stack.h for stack token definitions.
Refer to token-manufacturing.h for manufaturing token definitions.
Note

- Simulated EEPROM, version 2 is only supported on EM335x chips.
- NVM3 is currently only supported on EFx32 chips.

Modules

Tokens

Copyright © 2025 Silicon Laboratories. All rights reserved.

Tokens

Tokens

There are three main types of tokens:

- **Manufacturing tokens:** Tokens that are set at the factory and must not be changed through software operations.
- **Stack-level tokens:** Tokens that can be changed via the appropriate stack API calls.
- **Application level tokens:** Tokens that can be set via the token system API calls in this file.

The token system API controls writing tokens to non-volatile data and reading tokens from non-volatile data. If an application wishes to use application specific normal tokens, it must do so by creating its own token header file similar to token-stack.h. The macro `APPLICATION_TOKEN_HEADER` should be defined to equal the name of the header file in which application tokens are defined. If an application wishes to use application specific manufacturing tokens, it must do so by creating its own manufacturing token header file similar to token-manufacturing.h. The macro `APPLICATION_MFG_TOKEN_HEADER` should be defined to equal the name of the header file in which manufacturing tokens are defined.

Because the token system is based on memory locations within non-volatile storage, the token information could become out of sync without some kind of version tracking. The two defines, `CURRENT_MFG_TOKEN_VERSION` and `CURRENT_STACK_TOKEN_VERSION`, are used to make sure the stack stays in sync with the proper token set. If the application defines its own tokens, it is recommended that the application also define an application token to be a application version to ensure the application stays in sync with the proper token set.

The most general format of a token definition is:

```
#define CREATOR_name 16bit_value
#define NVM3KEY_name 20bit_value
#ifdef DEFINETYPES
  typedef data_type type
#endif
#ifdef DEFINETOKENS
  DEFINE_*_TOKEN(name, type, ... ,defaults)
#endif
```

The defined CREATOR is used as a distinct identifier tag for the token when using Simulated EEPROM or with manufacturing tokens. The CREATOR is necessary because the token name is defined differently depending on underlying implementation, so the CREATOR makes sure token definitions and data stay tagged and known. The only requirement on these creator definitions is that they all must be unique. A favorite method for picking creator codes is to use two ASCII characters inorder to make the codes more memorable. The 'name' part of the `#define CREATOR_name` must match the 'name' provided in the `DEFINE_*_TOKEN` because the token system uses this name to automatically link the two.

The defined NVM3KEY is used to map the token to an NVM3 key and is needed using NVM3 as the underlying storage mechanism. This key can also be used as an identifier for a token's NVM3 object when using the native NVM3 API. The NVM3 keys must be unique for one instance of the NVM3 backing storage. All tokens share the same NVM3 instance and hence all NVM3KEYS for tokens must be unique. The 'name' part of the `#define NVM3KEY_name` must match the 'name' provided in the `DEFINE_*_TOKEN` because the token system uses this name to automatically link the two. For indexed tokens, the 127 NVM3KEY values following the defined NVM3KEY for a token should also be reserved. This is done as one NVM3KEY is used for each index in an indexed token and hence these NVM3KEYS should not collide with the eys of other tokens.

As NVM3 is shared among several stacks and application code, the NVM3KEY values chosen must be defined in the correct region to avoid collisions.

Copyright © 2025 Silicon Laboratories. All rights reserved.

The following NVM3KEY regions are defined: 0x0xxxx : User objects 0x1xxxx : zigbee stack objects 0x2xxxx : Thread stack objects 0x3xxxx : Connect stack objects 0x4xxxx : Bluetooth stack objects

The typedef provides a convenient and efficient abstraction of the token data. Since some tokens are structs with multiple pieces of data inside of them, type defining the token type allows more efficient and readable local copies of the tokens throughout the code.

The typedef is wrapped with an `#ifdef DEFINETYPES` because the typdefs and token defs live in the same file, and DEFINETYPES is used to select only the typedefs when the file is included. Similarly, the `DEFINE_*_TOKEN` is wrapped with an `#ifdef DEFINETOKENS` as a method for selecting only the token definitions when the file is included.

The abstract definition, `DEFINE_*_TOKEN(name, type, ... ,defaults)` , has seven possible complete definitions: `DEFINE_BASIC_TOKEN(name, type, ...)` `DEFINE_INDEXED_TOKEN(name, type, arraysize, ...)` `DEFINE_COUNTER_TOKEN(name, type, ...)` `DEFINE_MFG_TOKEN(name, type, address, ...)` The three fields common to all `DEFINE_*_TOKEN` are:
name - The name of the token, which all information is tied to.
type - Type of the token which is the same as the typedef mentioned before.
... - The default value to which the token is set upon initialization.

Note

- The old DEFINE_FIXED* token definitions are no longer used. They remain defined for backwards compatibility. In current systems, the Simulated EEPROM or NVM3 is used for storing non-manufacturing tokens and the Simulated EEPROM or NVM3 intelligently manages where tokens are stored to provide wear leveling across the flash memory and increase the number of write cycles. Manufacturing tokens live at a fixed address, but they must use DEFINE_MFG_TOKEN so the token system knows they are manufacturing tokens.

DEFINE_BASIC_TOKEN is the simplest definition and will be used for the majority of tokens (tokens that are not indexed, not counters, and not manufacturing). Basic tokens are designed for data storage that is always accessed as a single element.

DEFINE_INDEXED_TOKEN should be used on tokens that look like arrays. For example, data storage that looks like:

```
uint32_t myData[5]
```

<br<blockquote> This example data storage can be a token with typedef of uint32_t and defined as INDEXED with arraysize of 5. The extra field in this token definition is: arraysize - The number of elements in the indexed token. Indexed tokens are designed for data storage that is logically grouped together, but elements are accessed individually. Note that when assigning an NVM3KEY for an indexed token, the 126 higher numbered NVM3KEYs following the NVM3KEY that you define are reserved for that token and no other tokens should be defined with NVM3KEYs in this region.

DEFINE_COUNTER_TOKEN should be used on tokens that are simple numbers where the majority of operations on the token is to increment the count. The reason for using DEFINE_COUNTER_TOKEN instead of DEFINE_BASIC_TOKEN is the special support that the token system provides for incrementing counters. The function call `halCommonIncrementCounterToken()` only operates on counter tokens and is more efficient in terms of speed, data compression, and write cyles for incrementing simple numbers in the token system.

DEFINE_MFG_TOKEN is a DEFINE_BASIC_TOKEN token at a specific address and the token is manufacturing data that is written only once. The major difference is this token is designated manufacturing, which means the token system treats it differently from stack or app tokens. Primarily, a manufacturing token is written only once and lives at a fixed address outside of the Simulated EEPROM or NVM3 system. Being a write once token, the token system will also aid in debugging by asserting if there is an attempt to write a manufacturing token.

Here is an example of two application tokens. The definition is compatible with both Simulated EEPROM and NVM3 as both CREATOR and NVM3KEY defines are included.

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
#define CREATOR_SENSOR_NAME       0x5354
#define CREATOR_SENSOR_PARAMETERS  0x5350
#define NVM3KEY_SENSOR_NAME       0x0AB54
#define NVM3KEY_SENSOR_PARAMETERS 0x00150
#ifdef DEFINETYPES
  typedef uint8_t tokTypeSensorName[10];
  typedef struct {
    uint8_t initValues[5];
    uint8_t reportInterval;
    uint16_t calibrationValue;
  } tokTypeSensorParameters;
#endif
#ifdef DEFINETOKENS
  DEFINE_BASIC_TOKEN(SENSOR_NAME,
                     tokTypeSensorName,
                     {'U','N','A','M','E','D',' ',' ',' ',' '})
  DEFINE_BASIC_TOKEN(SENSOR_PARAMETERS,
                     tokTypeSensorParameters,
                     {{0x01,0x02,0x03,0x04,0x05},5,0x0000})
#endif
```

Here is an example of how to use the two application tokens:

```
{
  tokTypeSensorName sensor;
  tokTypeSensorParameters params;

  halCommonGetToken(&sensor, TOKEN_SENSOR_NAME);
  halCommonGetToken(&params, TOKEN_SENSOR_PARAMETERS);
  if(params.calibrationValue == 0xBEEF) {
    params.reportInterval = 5;
  }
  halCommonSetToken(TOKEN_SENSOR_PARAMETERS, &params);
}
```

See token-stack.h to see the default set of tokens and their values.

The nodetest utility app can be used for generic manipulation such as loading default token values, viewing tokens, and writing tokens. **The nodetest utility cannot work with customer defined application tokens or manufacturing tokens. Using the nodetest utility will erase customer defined application tokens in the Simulated EEPROM and NVM3.**

The Simulated EEPROM or NVM3 will initialize tokens to their default values if the token does not yet exist, the token's creator code is changed, or the token's size changes.

Changing the number indexes in an INDEXED token will not alter existing entries. If the number of indexes is reduced, the entires that still fit in the token will retain their data and the entries that no longer fit will be erased. If the number of indexes is increased, the existing entries retain their data and the new entries are initialized to the token's defaults.

Further details on exact implementation can be found in code comments in token-stack.h file, the platform specific token-manufacturing.h file, the platform specific token.h file, and the platform specific token.c file.

Some functions in this file return an ::sl_status_t value. See sl_status.h for definitions of all ::sl_status_t return values.

See hal/micro/token.h for source code.

Functions

sl_status_t halStackInitTokens(void)

Initializes and enables the token system. Checks if the manufacturing and stack non-volatile data versions are correct.

Macros

Copyright © 2025 Silicon Laboratories. All rights reserved.

#define halCommonGetToken (data, token)
Macro that copies the token value from non-volatile storage into a RAM location. This macro can only be used with tokens that are defined using DEFINE_BASIC_TOKEN.

#define halCommonGetMfgToken (data, token)
Macro that copies the token value from non-volatile storage into a RAM location. This macro can only be used with tokens that are defined using DEFINE_MFG_TOKEN.

#define halCommonGetIndexedToken (data, token, index)
Macro that copies the token value from non-volatile storage into a RAM location. This macro can only be used with tokens that are defined using DEFINE_INDEXED_TOKEN.

#define halCommonSetToken (token, data)
Macro that sets the value of a token in non-volatile storage. This macro can only be used with tokens that are defined using DEFINE_BASIC_TOKEN.

#define halCommonSetIndexedToken (token, index, data)
Macro that sets the value of a token in non-volatile storage. This macro can only be used with tokens that are defined using DEFINE_INDEXED_TOKEN.

#define halCommonIncrementCounterToken (token)
Macro that increments the value of a token that is a counter. This macro can only be used with tokens that are defined using either DEFINE_COUNTER_TOKEN.

Function Documentation

halStackInitTokens

> sl_status_t halStackInitTokens (void)

Initializes and enables the token system. Checks if the manufacturing and stack non-volatile data versions are correct.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- An sl_status_t value indicating the success or failure of the command.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Sample APIs for Peripheral Access

Sample APIs for Peripheral Access

These are sample API for accessing peripherals and can be modified as needed for your applications.

Modules

Serial UART Communication

Button Control

Buzzer Control

LED Control

Flash Memory Control

Copyright © 2025 Silicon Laboratories. All rights reserved.

Serial UART Communication

Serial UART Communication

This API contains the HAL interfaces that applications must implement for the high-level serial code.

This header describes the interface between the high-level serial APIs in serial/serial.h and the low level UART implementation.

Some functions in this file return an EmberStatus value and others return ::sl_status_t. See error-def.h for definitions of all EmberStatus return values, and sl_status.h for all ::sl_status_t return values.

See hal/micro/serial.h for source code.

Serial HAL APIs

These functions must be implemented by the HAL in order for the serial code to operate. Only the higher-level serial code uses these functions, so they should not be called directly. The HAL should also implement the appropriate interrupt handlers to drain the TX queues and fill the RX FIFO queue.

sl_status_t halInternalUartInit(uint8_t port, SerialBaudRate rate, SerialParity parity, uint8_t stopBits)
Initializes the UART to the given settings (same parameters as ::sli_legacy_serial_init()).

void halInternalPowerDownUart(void)
This function is typically called by halPowerDown() and it is responsible for performing all the work internal to the UART needed to stop the UART before a sleep cycle.

void halInternalPowerUpUart(void)
This function is typically called by halPowerUp() and it is responsible for performing all the work internal to the UART needed to restart the UART after a sleep cycle.

void halInternalStartUartTx(uint8_t port)
Called by serial code whenever anything is queued for transmission to start any interrupt-driven transmission. May be called when transmission is already in progess.

void halInternalStopUartTx(uint8_t port)
Called by serial code to stop any interrupt-driven serial transmission currently in progress.

sl_status_t halInternalForceWriteUartData(uint8_t port, uint8_t *data, uint8_t length)
Directly writes a byte to the UART for transmission, regardless of anything currently queued for transmission. Should wait for anything currently in the UART hardware registers to finish transmission first, and block until `data` is finished being sent.

EmberStatus halInternalForceReadUartByte(uint8_t port, uint8_t *dataByte)
Directly reads a byte from the UART for reception, regardless of anything currently queued for reception. Does not block if a data byte has not been received.

void halInternalWaitUartTxComplete(uint8_t port)
Blocks until the UART has finished transmitting any data in its hardware registers.

void halInternalRestartUart(void)
This function is typically called by ::halInternalPowerUpBoard() and it is responsible for performing all the work internal to the UART needed to restart the UART after a sleep cycle. (For example, resyncing the DMA hardware and the serial FIFO.)

Copyright © 2025 Silicon Laboratories. All rights reserved.

bool halInternalUartFlowControlRxIsEnabled(uint8_t port)
Checks to see if the host is allowed to send serial data to the ncp - i.e., it is not being held off by nCTS or an XOFF. Returns true is the host is able to send.

bool halInternalUartXonRefreshDone(uint8_t port)
When Xon/Xoff flow control is used, returns true if the host is not being held off and XON refreshing is complete.

bool halInternalUartTxIsIdle(uint8_t port)
Returns true if the uart transmitter is idle, including the transmit shift register.

bool serialDropPacket(void)
Testing function implemented by the upper layer. Determines whether the next packet should be dropped. Returns true if the next packet should be dropped, false otherwise.

#define halInternalUartFlowControl (port)
This function is used in FIFO mode when flow control is enabled. It is called from sli_legacy_serial_read_byte(), and based on the number of bytes used in the uart receive queue, decides when to tell the host it may resume transmission.

#define halInternalUartRxPump (port)
This function exists only in software UART (SOFTUART) mode on the EM3xx. This function is called by ::sli_legacy_serial_read_byte(). It is responsible for maintaining synchronization between the emSerialRxQueue and the UART DMA.

#define halInternalUart1FlowControlRxIsEnabled ()
This function is used in FIFO mode when flow control is enabled. It is called from sli_legacy_serial_read_byte(), and based on the number of bytes used in the uart receive queue, decides when to tell the host it may resume transmission.

#define halInternalUart1XonRefreshDone ()
This function is used in FIFO mode when flow control is enabled. It is called from sli_legacy_serial_read_byte(), and based on the number of bytes used in the uart receive queue, decides when to tell the host it may resume transmission.

#define halInternalUart1TxIsIdle ()
This function is used in FIFO mode when flow control is enabled. It is called from sli_legacy_serial_read_byte(), and based on the number of bytes used in the uart receive queue, decides when to tell the host it may resume transmission.

Virtual UART API

API used by the stack in debug builds to receive data arriving over the virtual UART.

void halStackReceiveVuartMessage(uint8_t *data, uint8_t length)
When using a debug build with virtual UART support, this API is called by the stack when virtual UART data has been received over the debug channel.

Serial Mode Definitions

These are numerical definitions for the possible serial modes so that code can test for the one being used. There may be additional modes defined in the micro-specific micro.h.

#define SL_LEGACY_SERIAL_UNUSED 0
A numerical definition for a possible serial mode the code can test for.

#define SL_LEGACY_SERIAL_FIFO 1
A numerical definition for a possible serial mode the code can test for.

#define SL_LEGACY_SERIAL_LOWLEVEL 2
A numerical definition for a possible serial mode the code can test for.

Copyright © 2025 Silicon Laboratories. All rights reserved.

FIFO Utility Macros

These macros manipulate the FIFO queue data structures to add and remove data.

#define FIFO_ENQUEUE (queue, data, size)
Macro that enqueues a byte of data in a FIFO queue.

#define FIFO_DEQUEUE (queue, size)
Macro that de-queues a byte of data from a FIFO queue.

Enumerations

enum SerialBaudRate {

DEFINE_BAUD =(300) = 0
DEFINE_BAUD =(300) = 0
DEFINE_BAUD =(300) = 0
DEFINE_BAUD =(300) = 0
DEFINE_BAUD =(300) = 0
DEFINE_BAUD =(300) = 0
DEFINE_BAUD =(300) = 0
DEFINE_BAUD =(300) = 0
DEFINE_BAUD =(300) = 0
DEFINE_BAUD =(300) = 0
DEFINE_BAUD =(300) = 0
DEFINE_BAUD =(300) = 0
DEFINE_BAUD =(300) = 0
DEFINE_BAUD =(300) = 0
DEFINE_BAUD =(300) = 0
DEFINE_BAUD =(300) = 0
DEFINE_BAUD =(300) = 0
DEFINE_BAUD =(300) = 0
DEFINE_BAUD =(300) = 0
DEFINE_BAUD =(300) = 0

}
Assign numerical values for variables that hold Baud Rate parameters.

enum SerialParity {

DEFINE_PARITY =(NONE) = 0U
DEFINE_PARITY =(NONE) = 0U
DEFINE_PARITY =(NONE) = 0U

}
CORTEXM3_EFM32_MICRO.

Functions

void halHostFlushBuffers(void)

uint16_t halHostEnqueueTx(const uint8_t *data, uint16_t length)

void halHostFlushTx(void)

uint16_t serialCopyFromRx(const uint8_t *data, uint16_t length)

void emLoadSerialTx(void)

Serial HAL APIs Documentation

halInternalUartInit

sl_status_t halInternalUartInit (uint8_t port, SerialBaudRate rate, SerialParity parity, uint8_t stopBits)

Copyright © 2025 Silicon Laboratories. All rights reserved.

Initializes the UART to the given settings (same parameters as ::sli_legacy_serial_init()).

Parameters

Type	Direction	Argument Name	Description
uint8_t	N/A	port	Serial port number (0 or 1).
SerialBaudRate	N/A	rate	Baud rate (see SerialBaudRate).
SerialParity	N/A	parity	Parity value (see SerialParity).
uint8_t	N/A	stopBits	Number of stop bits.

Returns

- An error code if initialization failed (such as invalid baud rate), otherise SL_STATUS_OK.

halInternalPowerDownUart

```
void halInternalPowerDownUart (void )
```

This function is typically called by halPowerDown() and it is responsible for performing all the work internal to the UART needed to stop the UART before a sleep cycle.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

halInternalPowerUpUart

```
void halInternalPowerUpUart (void )
```

This function is typically called by halPowerUp() and it is responsible for performing all the work internal to the UART needed to restart the UART after a sleep cycle.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

halInternalStartUartTx

```
void halInternalStartUartTx (uint8_t port)
```

Called by serial code whenever anything is queued for transmission to start any interrupt-driven transmission. May be called when transmission is already in progess.

Parameters

Type	Direction	Argument Name	Description
uint8_t	N/A	port	Serial port number (0 or 1).

Copyright © 2025 Silicon Laboratories. All rights reserved.

halInternalStopUartTx

```
void halInternalStopUartTx (uint8_t port)
```

Called by serial code to stop any interrupt-driven serial transmission currently in progress.

Parameters

Type	Direction	Argument Name	Description
uint8_t	N/A	port	Serial port number (0 or 1).

halInternalForceWriteUartData

```
sl_status_t halInternalForceWriteUartData (uint8_t port, uint8_t * data, uint8_t length)
```

Directly writes a byte to the UART for transmission, regardless of anything currently queued for transmission. Should wait for anything currently in the UART hardware registers to finish transmission first, and block until `data` is finished being sent.

Parameters

Type	Direction	Argument Name	Description
uint8_t	N/A	port	Serial port number (0 or 1).
uint8_t *	N/A	data	Pointer to the data to be transmitted.
uint8_t	N/A	length	The length of data to be transmitted

halInternalForceReadUartByte

```
EmberStatus halInternalForceReadUartByte (uint8_t port, uint8_t * dataByte)
```

Directly reads a byte from the UART for reception, regardless of anything currently queued for reception. Does not block if a data byte has not been received.

Parameters

Type	Direction	Argument Name	Description
uint8_t	N/A	port	Serial port number (0 or 1).
uint8_t *	N/A	dataByte	The byte to receive data into.

halInternalWaitUartTxComplete

```
void halInternalWaitUartTxComplete (uint8_t port)
```

Blocks until the UART has finished transmitting any data in its hardware registers.

Parameters

Type	Direction	Argument Name	Description
uint8_t	N/A	port	Serial port number (0 or 1).

Copyright © 2025 Silicon Laboratories. All rights reserved.

halInternalRestartUart

```
void halInternalRestartUart (void )
```

This function is typically called by ::halInternalPowerUpBoard() and it is responsible for performing all the work internal to the UART needed to restart the UART after a sleep cycle. (For example, resyncing the DMA hardware and the serial FIFO.)

Parameters

Type	Direction	Argument Name	Description
void	N/A		

halInternalUartFlowControlRxIsEnabled

```
bool halInternalUartFlowControlRxIsEnabled (uint8_t port)
```

Checks to see if the host is allowed to send serial data to the ncp - i.e., it is not being held off by nCTS or an XOFF. Returns true is the host is able to send.

Parameters

Type	Direction	Argument Name	Description
uint8_t	N/A	port	

halInternalUartXonRefreshDone

```
bool halInternalUartXonRefreshDone (uint8_t port)
```

When Xon/Xoff flow control is used, returns true if the host is not being held off and XON refreshing is complete.

Parameters

Type	Direction	Argument Name	Description
uint8_t	N/A	port	

halInternalUartTxIsIdle

```
bool halInternalUartTxIsIdle (uint8_t port)
```

Returns true if the uart transmitter is idle, including the transmit shift register.

Parameters

Type	Direction	Argument Name	Description
uint8_t	N/A	port	

serialDropPacket

Copyright © 2025 Silicon Laboratories. All rights reserved.

bool serialDropPacket (void)

Testing function implemented by the upper layer. Determines whether the next packet should be dropped. Returns true if the next packet should be dropped, false otherwise.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Virtual UART API Documentation

halStackReceiveVuartMessage

void halStackReceiveVuartMessage (uint8_t * data, uint8_t length)

When using a debug build with virtual UART support, this API is called by the stack when virtual UART data has been received over the debug channel.

Parameters

Type	Direction	Argument Name	Description
uint8_t *	N/A	data	Pointer to the the data received
uint8_t	N/A	length	Length of the data received

Serial Mode Definitions Documentation

FIFO Utility Macros Documentation

Enumeration Documentation

SerialBaudRate

SerialBaudRate

Assign numerical values for variables that hold Baud Rate parameters.

Enumerator	
DEFINE_BAUD	
DEFINE_BAUD	
DEFINE_BAUD	
DEFINE_BAUD	
DEFINE_BAUD	
DEFINE_BAUD	
DEFINE_BAUD	
DEFINE_BAUD	
DEFINE_BAUD	
DEFINE_BAUD	
DEFINE_BAUD	

Copyright © 2025 Silicon Laboratories. All rights reserved.

DEFINE_BAUD	
DEFINE_BAUD	
DEFINE_BAUD	
DEFINE_BAUD	
DEFINE_BAUD	
DEFINE_BAUD	
DEFINE_BAUD	
DEFINE_BAUD	

SerialParity

> SerialParity

CORTEXM3_EFM32_MICRO.

Assign numerical values for the types of parity. Use for variables that hold Parity parameters.

Enumerator	
DEFINE_PARITY	
DEFINE_PARITY	
DEFINE_PARITY	

Function Documentation

halHostFlushBuffers

> void halHostFlushBuffers (void)

Parameters

Type	Direction	Argument Name	Description
void	N/A		

halHostEnqueueTx

> uint16_t halHostEnqueueTx (const uint8_t * data, uint16_t length)

Parameters

Type	Direction	Argument Name	Description
const uint8_t *	N/A	data	
uint16_t	N/A	length	

halHostFlushTx

> void halHostFlushTx (void)

Copyright © 2025 Silicon Laboratories. All rights reserved.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

serialCopyFromRx

```
uint16_t serialCopyFromRx (const uint8_t * data, uint16_t length)
```

Parameters

Type	Direction	Argument Name	Description
const uint8_t *	N/A	data	
uint16_t	N/A	length	

emLoadSerialTx

```
void emLoadSerialTx (void )
```

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Copyright © 2025 Silicon Laboratories. All rights reserved.

Button Control

Button Control

Sample API functions for using push-buttons.

See button.h for source code.

Button State Definitions

A set of numerical definitions for use with the button APIs indicating the state of a button.

#define BUTTON_PRESSED 1
Button state is pressed.

#define BUTTON_RELEASED 0
Button state is released.

Functions

void halInternalInitButton(void)
Initializes the buttons. This function is automatically called by halInit().

uint8_t halButtonState(uint8_t button)
Returns the current state (pressed or released) of a button.

void halButtonIsr(uint8_t button, uint8_t state)
A callback called in interrupt context whenever a button changes its state.

Button State Definitions Documentation

Function Documentation

halInternalInitButton

```
void halInternalInitButton (void )
```

Initializes the buttons. This function is automatically called by halInit().

Parameters

Type	Direction	Argument Name	Description
void	N/A		

halButtonState

```
uint8_t halButtonState (uint8_t button)
```

Returns the current state (pressed or released) of a button.

Parameters

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
uint8_t	N/A	button	The button being queried, either BUTTON0 or BUTTON1 as defined in the appropriate BOARD_HEADER.

Note

- This function is correlated with halButtonIsr() and so returns the shadow state rather than reading the actual state of the pin.

Returns

- BUTTON_PRESSED if the button is pressed or BUTTON_RELEASED if the button is not pressed.

halButtonIsr

```
void halButtonIsr (uint8_t button, uint8_t state)
```

A callback called in interrupt context whenever a button changes its state.

Parameters

Type	Direction	Argument Name	Description
uint8_t	N/A	button	The button which has changed state, either BUTTON0 or BUTTON1 as defined in the appropriate BOARD_HEADER.
uint8_t	N/A	state	The new state of the button referenced by the button parameter, either BUTTON_PRESSED if the button has been pressed or BUTTON_RELEASED if the button has been released.

- Application Usage:\n Must be implemented by the application. This function should

  contain the functionality to be executed in response to changes of state in each of the buttons, or callbacks to the appropriate functionality.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Buzzer Control

Buzzer Control

Copyright © 2025 Silicon Laboratories. All rights reserved.

LED Control

LED Control

Sample API funtions for controlling LEDs.

When specifying an LED to use, always use the BOARDLEDx definitions that are defined within the BOARD_HEADER.

See led.h for source code.

Typedefs

typedef enum HalBoardLedPins — HalBoardLed
Ensures that the definitions from the BOARD_HEADER are always used as parameters to the LED functions.

Functions

void — halInternalInitLed(void)
Configures GPIOs pertaining to the control of LEDs.

void — halStackIndicateActivity(bool turnOn)
Called by the stack to indicate activity over the radio (for both transmission and reception). It is called once with `turnOn` true and shortly thereafter with `turnOn` false.

Typedef Documentation

HalBoardLed

```
typedef enum HalBoardLedPins HalBoardLed
```

Ensures that the definitions from the BOARD_HEADER are always used as parameters to the LED functions.

Function Documentation

halInternalInitLed

```
void halInternalInitLed (void )
```

Configures GPIOs pertaining to the control of LEDs.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

halStackIndicateActivity

```
void halStackIndicateActivity (bool turnOn)
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Called by the stack to indicate activity over the radio (for both transmission and reception). It is called once with `turnOn` true and shortly thereafter with `turnOn` false.

Parameters

Type	Direction	Argument Name	Description
bool	N/A	turnOn	See Usage.

Typically does something interesting, such as change the state of an LED.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Flash Memory Control

Flash Memory Control

Definition and description of public flash manipulation routines.

Note

- During an erase or a write the flash is not available, which means code will not be executable from flash. These routines still execute from flash, though, since the bus architecture can support doing so. **Additonally, this also means all interrupts will be disabled.**

Hardware documentation indicates 40us for a write and 21ms for an erase.

See flash.h for source code.

Functions

bool halFlashEraseIsActive(void)
Tells the calling code if a Flash Erase operation is active.

Function Documentation

halFlashEraseIsActive

```
bool halFlashEraseIsActive (void )
```

Tells the calling code if a Flash Erase operation is active.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

This state is import to know because Flash Erasing is ATOMIC for 21ms and could disrupt interrupt latency. But if an ISR can know that it wasn't serviced immediately due to Flash Erasing, then the ISR has the opportunity to correct in whatever manner it needs to.

Returns

- A bool flag: true if Flash Erase is active, false otherwise.

Copyright © 2025 Silicon Laboratories. All rights reserved.

System Timer Control

System Timer Control

Functions that provide access to the system clock.

A single system tick (as returned by halCommonGetInt16uMillisecondTick() and halCommonGetInt32uMillisecondTick()) is approximately 1 millisecond.

- When used with a 32.768kHz crystal, the system tick is 0.976 milliseconds.
- When used with a 3.6864MHz crystal, the system tick is 1.111 milliseconds.

A single quarter-second tick (as returned by halCommonGetInt16uQuarterSecondTick()) is approximately 0.25 seconds.

The values used by the time support functions will wrap after an interval. The length of the interval depends on the length of the tick and the number of bits in the value. However, there is no issue when comparing time deltas of less than half this interval with a subtraction, if all data types are the same.

See system-timer.h for source code.

Functions

uint16_t halInternalStartSystemTimer(void)
Initializes the system tick.

uint16_t halCommonGetInt16uMillisecondTick(void)
Returns the current system time in system ticks, as a 16-bit value.

uint32_t halCommonGetInt32uMillisecondTick(void)
Returns the current system time in system ticks, as a 32-bit value.

uint64_t halCommonGetInt64uMillisecondTick(void)
Returns the current system time in system ticks, as a 64-bit value.

uint16_t halCommonGetInt16uQuarterSecondTick(void)
Returns the current system time in quarter second ticks, as a 16-bit value.

sl_status_t halSleepForQuarterSeconds(uint32_t *duration)
Uses the system timer to enter SLEEPMODE_WAKETIMER for approximately the specified amount of time (provided in quarter seconds).

sl_status_t halSleepForMilliseconds(uint32_t *duration)
Uses the system timer to enter SLEEPMODE_WAKETIMER for approximately the specified amount of time (provided in milliseconds). Note that since the system timer ticks at a rate of 1024Hz, a second is comprised of 1024 milliseconds in this function.

sl_status_t halCommonIdleForMilliseconds(uint32_t *duration)
Uses the system timer to enter SLEEPMODE_IDLE for approximately the specified amount of time (provided in milliseconds).

Macros

#define halIdleForMilliseconds (duration)

Copyright © 2025 Silicon Laboratories. All rights reserved.

Function Documentation

halInternalStartSystemTimer

```
uint16_t halInternalStartSystemTimer (void )
```

Initializes the system tick.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Time to update the async registers after RTC is started (units of 100 microseconds).

halCommonGetInt16uMillisecondTick

```
uint16_t halCommonGetInt16uMillisecondTick (void )
```

Returns the current system time in system ticks, as a 16-bit value.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- The least significant 16 bits of the current system time, in system ticks.

halCommonGetInt32uMillisecondTick

```
uint32_t halCommonGetInt32uMillisecondTick (void )
```

Returns the current system time in system ticks, as a 32-bit value.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

- EmberStack Usage:\n Unused, implementation optional.

Returns

- The least significant 32 bits of the current system time, in system ticks.

halCommonGetInt64uMillisecondTick

Copyright © 2025 Silicon Laboratories. All rights reserved.

uint64_t halCommonGetInt64uMillisecondTick (void)

Returns the current system time in system ticks, as a 64-bit value.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

- EmberStack Usage:\n Unused, implementation optional.

Returns

- 64 bits containing the current system time, in system ticks.

halCommonGetInt16uQuarterSecondTick

uint16_t halCommonGetInt16uQuarterSecondTick (void)

Returns the current system time in quarter second ticks, as a 16-bit value.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

- EmberStack Usage:\n Unused, implementation optional.

Returns

- The least significant 16 bits of the current system time, in system ticks multiplied by 256.

halSleepForQuarterSeconds

sl_status_t halSleepForQuarterSeconds (uint32_t * duration)

Uses the system timer to enter SLEEPMODE_WAKETIMER for approximately the specified amount of time (provided in quarter seconds).

Parameters

Type	Direction	Argument Name	Description
uint32_t *	N/A	duration	The amount of time, expressed in quarter seconds, that the micro should be placed into SLEEPMODE_WAKETIMER. When the function returns, this parameter provides the amount of time remaining out of the original sleep time request (normally the return value will be 0).

This function returns ::SL_STATUS_OK and the duration parameter is decremented to 0 after sleeping for the specified amount of time. If an interrupt occurs that brings the chip out of sleep, the function returns ::SL_STATUS_SUSPENDED and the duration parameter reports the amount of time remaining out of the original request.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Note

- This routine always enables interrupts.
- The maximum sleep time of the hardware is limited on AVR-based platforms to 8 seconds, on EM2XX-based platforms to 64 seconds, and on EM35x platforms to 48.5 days. Any sleep duration greater than this limit will wake up briefly (e.g. 16 microseconds) to reenable another sleep cycle.

The EM2xx has a 16 bit sleep timer, which normally runs at 1024Hz. In order to support long sleep durations, the chip will periodically wake up to manage a larger timer in software. This periodic wakeup is normally triggered once every 32 seconds. However, this period can be extended to once every 2.275 hours by building with **ENABLE_LONG_SLEEP_CYCLES** defined. This definition enables the use of a prescaler when sleeping for more than 63 seconds at a time. However, this define also imposes the following limitations:

1. The chip may only wake up from the sleep timer. (External GPIO wake events may not be used)
2. Each time a sleep cycle is performed, a loss of accuracy up to +/-750ms will be observed in the system timer.

- # EmberStack Usage:\n Unused, implementation optional.

Returns

- An sl_status_t value indicating the success or failure of the command.

halSleepForMilliseconds

```
sl_status_t halSleepForMilliseconds (uint32_t * duration)
```

Uses the system timer to enter SLEEPMODE_WAKETIMER for approximately the specified amount of time (provided in milliseconds). Note that since the system timer ticks at a rate of 1024Hz, a second is comprised of 1024 milliseconds in this function.

Parameters

Type	Direction	Argument Name	Description
uint32_t *	N/A	duration	The amount of time, expressed in milliseconds (1024 milliseconds = 1 second), that the micro should be placed into SLEEPMODE_WAKETIMER. When the function returns, this parameter provides the amount of time remaining out of the original sleep time request (normally the return value will be 0).

This function returns ::SL_STATUS_OK and the duration parameter is decremented to 0 after sleeping for the specified amount of time. If an interrupt occurs that brings the chip out of sleep, the function returns ::SL_STATUS_SUSPENDED and the duration parameter reports the amount of time remaining out of the original request.

Note

- This routine always enables interrupts.
- This function is not implemented on AVR-based platforms.
- Sleep durations less than 3 milliseconds are not allowed on on EM2XX-based platforms. Any attempt to sleep for less than 3 milliseconds on EM2XX-based platforms will cause the function to immediately exit without sleeping and return ::SL_STATUS_SUSPENDED.
- The maximum sleep time of the hardware is limited on EM2XX-based platforms to 32 seconds. Any sleep duration greater than this limit will wake up briefly (e.g. 16 microseconds) to reenable another sleep cycle. Due to this limitation, this function should not be used with durations within 3 milliseconds of a multiple 32 seconds. The short sleep cycle that results from such durations is not handled reliably by the system timer on EM2XX-based platforms. If a sleep duration within 3 milliseconds of a multiple of 32 seconds is desired, halSleepForQuarterSeconds should be used.
-

Copyright © 2025 Silicon Laboratories. All rights reserved.

EmberStack Usage:\n Unused, implementation optional.

Returns

- An sl_status_t value indicating the success or failure of the command.

halCommonIdleForMilliseconds

```
sl_status_t halCommonIdleForMilliseconds (uint32_t * duration)
```

Uses the system timer to enter SLEEPMODE_IDLE for approximately the specified amount of time (provided in milliseconds).

Parameters

Type	Direction	Argument Name	Description
uint32_t *	N/A	duration	The amount of time, expressed in milliseconds, that the micro should be placed into SLEEPMODE_IDLE. When the function returns, this parameter provides the amount of time remaining out of the original idle time request (normally the return value will be 0).

This function returns ::SL_STATUS_OK and the duration parameter is decremented to 0 after idling for the specified amount of time. If an interrupt occurs that brings the chip out of idle, the function returns ::SL_STATUS_SUSPENDED and the duration parameter reports the amount of time remaining out of the original request.

Note

- This routine always enables interrupts.

- # EmberStack Usage:\n Unused, implementation optional.

Returns

- An sl_status_t value indicating the success or failure of the command.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Symbol Timer Control

Symbol Timer Control

Copyright © 2025 Silicon Laboratories. All rights reserved.

HAL Configuration

HAL Configuration

Modules

Sample Breakout Board Configuration

IAR PLATFORM_HEADER Configuration

Common PLATFORM_HEADER Configuration

NVIC Configuration

Reset Cause Type Definitions

Copyright © 2025 Silicon Laboratories. All rights reserved.

Sample Breakout Board Configuration

Sample Breakout Board Configuration

Copyright © 2025 Silicon Laboratories. All rights reserved.

IAR PLATFORM_HEADER Configuration

IAR PLATFORM_HEADER Configuration

Compiler and Platform specific definitions and typedefs for the IAR ARM C compiler.

Note

- iar.h should be included first in all source files by setting the preprocessor macro PLATFORM_HEADER to point to it. iar.h automatically includes platform-common.h.

See iar.h and platform-common.h for source code.

Master Variable Types

LEGACY_PHY_BUILDThese are a set of typedefs to make the size of all variable declarations explicitly known.

typedef bool	boolean A typedef to make the size of the variable explicitly known.
typedef unsigned char	int8u Denotes that this platform supports 64-bit data-types.
typedef signed char	int8s Denotes that this platform supports 64-bit data-types.
typedef unsigned short	int16u Denotes that this platform supports 64-bit data-types.
typedef signed short	int16s Denotes that this platform supports 64-bit data-types.
typedef unsigned int	int32u Denotes that this platform supports 64-bit data-types.
typedef signed int	int32s Denotes that this platform supports 64-bit data-types.
typedef unsigned long long	int64u Denotes that this platform supports 64-bit data-types.
typedef signed long long	int64s Denotes that this platform supports 64-bit data-types.
typedef unsigned int	PointerType Denotes that this platform supports 64-bit data-types.
#define	HAL_HAS_INT64 Denotes that this platform supports 64-bit data-types.
#define	_HAL_USE_COMMON_PGM_ Use the Master Program Memory Declarations from platform-common.h.

Miscellaneous Macros

Copyright © 2025 Silicon Laboratories. All rights reserved.

void halInternalAssertFailed(const char *filename, int linenumber)
A prototype definition for use by the assert macro. (see hal/micro/micro.h)

void halInternalResetWatchDog(void)
Macro to reset the watchdog timer. Note: be very very careful when using this as you can easily get into an infinite loop if you are not careful.

#define BIGENDIAN_CPU false
A convenient method for code to know what endiannes processor it is running on. For the Cortex-M3, we are little endian.

#define NTOHS (val)
Define intrinsics for NTOHL and NTOHS to save code space by making endian.c compile to nothing.

#define NTOHL (val)
A convenient method for code to know what endiannes processor it is running on. For the Cortex-M3, we are little endian.

#define NO_STRIPPING __root
A friendlier name for the compiler's intrinsic for not stripping.

#define EEPROM errorerror
A friendlier name for the compiler's intrinsic for eeprom reference.

#define __SOURCEFILE__ __FILE__
The **SOURCEFILE** macro is used by asserts to list the filename if it isn't otherwise defined, set it to the compiler intrinsic which specifies the whole filename and path of the sourcefile.

#define assert (condition)
A custom implementation of the C language assert macro. This macro implements the conditional evaluation and calls the function halInternalAssertFailed(). (see hal/micro/micro.h)

#define halResetWatchdog ()
A convenient method for code to know what endiannes processor it is running on. For the Cortex-M3, we are little endian.

#define UNUSED
Declare a variable as unused to avoid a warning. Has no effect in IAR builds.

#define SIGNED_ENUM
Some platforms need to cast enum values that have the high bit set.

#define STACK_FILL_VALUE 0xCDCDCDCDU
Define the magic value that is interpreted by IAR C-SPY's Stack View.

#define RAMFUNC __ramfunc
Define a generic RAM function identifier to a compiler specific one.

#define NO_OPERATION ()
Define a generic no operation identifier to a compiler specific one.

#define SET_REG_FIELD (reg, field, value)
A convenience macro that makes it easy to change the field of a register to any unsigned value.

#define SET_CMSIS_REG (reg, mask, value)
A convenience macro that makes it easy to change a register using the provided mask(s) and value(s). Example: SET_CMSIS_REG(GPIO->P[1].CFGH, (_GPIO_P_CFGH_Px5_MASK | _GPIO_P_CFGH_Px6_MASK), (GPIO_P_CFGH_Px5_OUT | GPIO_P_CFGH_Px6_OUT));.

#define SET_CMSIS_REG_FIELD (reg, field, value)
A convenience macro that makes it easy to change the field of a register, as defined in CMSIS Device headers, to any unsigned value. Example using EM35xx: SET_CMSIS_REG_FIELD(GPIO->P[0].CFGL, GPIO_P_CFGL_Px0, _GPIO_P_CFGL_Px0_OUT);.

Copyright © 2025 Silicon Laboratories. All rights reserved.

#define simulatedTimePasses ()
Stub for code not running in simulation.

#define simulatedTimePassesMs (x)
Stub for code not running in simulation.

#define simulatedSerialTimePasses ()
Stub for code not running in simulation.

#define _HAL_USE_COMMON_DIVMOD_
Use the Divide and Modulus Operations from platform-common.h.

#define VAR_AT_SEGMENT (__variableDeclaration, __segmentName)
Provide a portable way to specify the segment where a variable lives.

#define STRINGIZE (X)
Convinience macro for turning a token into a string.

#define ALIGNMENT (X)
Provide a portable way to align data.

#define WEAK (__symbol)
Provide a portable way to specify a symbol as weak.

#define NO_INIT (__symbol)
Provide a portable way to specify a non initialized symbol.

#define STATIC_ASSERT (__condition, __errorstr)
Provide a portable way to specify a compile time assert.

External Declarations

If the line below is uncommented we will use Ember memory APIs, otherwise, we will use the C Standard library (memset,memcpy,memmove) APIs.These are routines that are defined in certain header files that we don't want to include, e.g. stdlib.h

int abs(int I)
Returns the absolute value of I (also called the magnitude of I). That is, if I is negative, the result is the opposite of I, but if I is nonnegative the result is I.

#define PLATCOMMONOKTOINCLUDE
Include platform-common.h last to pick up defaults and common definitions.

#define MAIN_FUNCTION_PARAMETERS void
The kind of arguments the main function takes.

#define MAIN_FUNCTION_ARGUMENTS
Include platform-common.h last to pick up defaults and common definitions.

Portable segment names

#define __NO_INIT__ ".noinitlegacy"
Portable segment names.

#define __DEBUG_CHANNEL__ "DEBUG_CHANNEL"
Portable segment names.

#define __INTVEC__ ".intvec"
Portable segment names.

Copyright © 2025 Silicon Laboratories. All rights reserved.

#define __CSTACK__ "CSTACK"
Portable segment names.

#define __RESETINFO__ "RESETINFO"
Portable segment names.

#define __DATA_INIT__ ".data_init"
Portable segment names.

#define __DATA__ ".data"
Portable segment names.

#define __BSS__ ".bss"
Portable segment names.

#define __CONST__ ".rodata"
Portable segment names.

#define __TEXT__ ".text"
Portable segment names.

#define __TEXTRW_INIT__ ".textrw_init"
Portable segment names.

#define __TEXTRW__ ".textrw"
Portable segment names.

#define __AAT__ "AAT"
Portable segment names.

#define __BAT__ "BAT"
Portable segment names.

#define __BAT_INIT__ "BAT"
Portable segment names.

#define __FAT__ "FAT"
Portable segment names.

#define __RAT__ "RAT"
Portable segment names.

#define __SIMEE__ "SIMEE"
Portable segment names.

#define __PSSTORE__ "PSSTORE"
Portable segment names.

#define __LONGTOKEN__ "LONGTOKEN"
Portable segment names.

#define __EMHEAP__ "EMHEAP"
Portable segment names.

#define __GUARD_REGION__ "GUARD_REGION"
Portable segment names.

#define __DLIB_PERTHREAD_INIT__ "__DLIB_PERTHREAD_init"
Portable segment names.

#define __DLIB_PERTHREAD_INITIALIZED_DATA__ "DLIB_PERTHREAD_INITIALIZED_DATA"
Portable segment names.

#define __DLIB_PERTHREAD_ZERO_DATA__ "DLIB_PERTHREAD_ZERO_DATA"
Portable segment names.

Copyright © 2025 Silicon Laboratories. All rights reserved.

#define __INTERNAL_STORAGE__ "INTERNAL_STORAGE"
Portable segment names.

#define __LOCKBITS_IN_MAINFLASH__ "LOCKBITS_IN_MAINFLASH"
Portable segment names.

#define __UNRETAINED_RAM__ "UNRETAINED_RAM"
Portable segment names.

#define _NO_INIT_SEGMENT_BEGIN __segment_begin(__NO_INIT__)
Portable segment names.

#define _DEBUG_CHANNEL_SEGMENT_BEGIN __segment_begin(__DEBUG_CHANNEL__)
Portable segment names.

#define _INTVEC_SEGMENT_BEGIN __segment_begin(__INTVEC__)
Portable segment names.

#define _CSTACK_SEGMENT_BEGIN __segment_begin(__CSTACK__)
Portable segment names.

#define _RESETINFO_SEGMENT_BEGIN __segment_begin(__RESETINFO__)
Portable segment names.

#define _DATA_INIT_SEGMENT_BEGIN __segment_begin(__DATA_INIT__)
Portable segment names.

#define _DATA_SEGMENT_BEGIN __segment_begin(__DATA__)
Portable segment names.

#define _BSS_SEGMENT_BEGIN __segment_begin(__BSS__)
Portable segment names.

#define _CONST_SEGMENT_BEGIN __segment_begin(__CONST__)
Portable segment names.

#define _TEXT_SEGMENT_BEGIN __segment_begin(__TEXT__)
Portable segment names.

#define _TEXTRW_INIT_SEGMENT_BEGIN __segment_begin(__TEXTRW_INIT__)
Portable segment names.

#define _TEXTRW_SEGMENT_BEGIN __segment_begin(__TEXTRW__)
Portable segment names.

#define _AAT_SEGMENT_BEGIN __segment_begin(__AAT__)
Portable segment names.

#define _BAT_SEGMENT_BEGIN __segment_begin(__BAT__)
Portable segment names.

#define _BAT_INIT_SEGMENT_BEGIN __segment_begin(__BAT_INIT__)
Portable segment names.

#define _FAT_SEGMENT_BEGIN __segment_begin(__FAT__)
Portable segment names.

#define _RAT_SEGMENT_BEGIN __segment_begin(__RAT__)
Portable segment names.

#define _SIMEE_SEGMENT_BEGIN __segment_begin(__SIMEE__)
Portable segment names.

Copyright © 2025 Silicon Laboratories. All rights reserved.

#define _PSSTORE_SEGMENT_BEGIN __segment_begin(__PSSTORE__)
Portable segment names.

#define _LONGTOKEN_SEGMENT_BEGIN __segment_begin(__LONGTOKEN__)
Portable segment names.

#define _EMHEAP_SEGMENT_BEGIN __segment_begin(__EMHEAP__)
Portable segment names.

#define _GUARD_REGION_SEGMENT_BEGIN __segment_begin(__GUARD_REGION__)
Portable segment names.

#define _DLIB_PERTHREAD_INIT_SEGMENT_BEGIN __segment_begin(__DLIB_PERTHREAD_INIT__)
Portable segment names.

#define _DLIB_PERTHREAD_INITIALIZED_DATA_SEGMENT_BEGIN
__segment_begin(__DLIB_PERTHREAD_INITIALIZED_DATA__)
Portable segment names.

#define _DLIB_PERTHREAD_ZERO_DATA_SEGMENT_BEGIN __segment_begin(__DLIB_PERTHREAD_ZERO_DATA__)
Portable segment names.

#define _INTERNAL_STORAGE_SEGMENT_BEGIN __segment_begin(__INTERNAL_STORAGE__)
Portable segment names.

#define _LOCKBITS_IN_MAINFLASH_SEGMENT_BEGIN __segment_begin(__LOCKBITS_IN_MAINFLASH__)
Portable segment names.

#define _UNRETAINED_RAM_SEGMENT_BEGIN __segment_begin(__UNRETAINED_RAM__)
Portable segment names.

#define _NO_INIT_SEGMENT_END __segment_end(__NO_INIT__)
Portable segment names.

#define _DEBUG_CHANNEL_SEGMENT_END __segment_end(__DEBUG_CHANNEL__)
Portable segment names.

#define _INTVEC_SEGMENT_END __segment_end(__INTVEC__)
Portable segment names.

#define _CSTACK_SEGMENT_END __segment_end(__CSTACK__)
Portable segment names.

#define _RESETINFO_SEGMENT_END __segment_end(__RESETINFO__)
Portable segment names.

#define _DATA_INIT_SEGMENT_END __segment_end(__DATA_INIT__)
Portable segment names.

#define _DATA_SEGMENT_END __segment_end(__DATA__)
Portable segment names.

#define _BSS_SEGMENT_END __segment_end(__BSS__)
Portable segment names.

#define _CONST_SEGMENT_END __segment_end(__CONST__)
Portable segment names.

#define _TEXT_SEGMENT_END __segment_end(__TEXT__)
Portable segment names.

#define _TEXTRW_INIT_SEGMENT_END __segment_end(__TEXTRW_INIT__)
Portable segment names.

Copyright © 2025 Silicon Laboratories. All rights reserved.

#define _TEXTRW_SEGMENT_END __segment_end(__TEXTRW__)
Portable segment names.

#define _AAT_SEGMENT_END __segment_end(__AAT__)
Portable segment names.

#define _BAT_SEGMENT_END __segment_end(__BAT__)
Portable segment names.

#define _BAT_INIT_SEGMENT_END __segment_end(__BAT_INIT__)
Portable segment names.

#define _FAT_SEGMENT_END __segment_end(__FAT__)
Portable segment names.

#define _RAT_SEGMENT_END __segment_end(__RAT__)
Portable segment names.

#define _SIMEE_SEGMENT_END __segment_end(__SIMEE__)
Portable segment names.

#define _PSSTORE_SEGMENT_END __segment_end(__PSSTORE__)
Portable segment names.

#define _LONGTOKEN_SEGMENT_END __segment_end(__LONGTOKEN__)
Portable segment names.

#define _EMHEAP_SEGMENT_END __segment_end(__EMHEAP__)
Portable segment names.

#define _GUARD_REGION_SEGMENT_END __segment_end(__GUARD_REGION__)
Portable segment names.

#define _DLIB_PERTHREAD_INIT_SEGMENT_END __segment_end(__DLIB_PERTHREAD_INIT__)
Portable segment names.

#define _DLIB_PERTHREAD_INITIALIZED_DATA_SEGMENT_END
__segment_end(__DLIB_PERTHREAD_INITIALIZED_DATA__)
Portable segment names.

#define _DLIB_PERTHREAD_ZERO_DATA_SEGMENT_END __segment_end(__DLIB_PERTHREAD_ZERO_DATA__)
Portable segment names.

#define _INTERNAL_STORAGE_SEGMENT_END __segment_end(__INTERNAL_STORAGE__)
Portable segment names.

#define _LOCKBITS_IN_MAINFLASH_SEGMENT_END __segment_end(__LOCKBITS_IN_MAINFLASH__)
Portable segment names.

#define _UNRETAINED_RAM_SEGMENT_END __segment_end(__UNRETAINED_RAM__)
Portable segment names.

#define _NO_INIT_SEGMENT_SIZE __segment_size(__NO_INIT__)
Portable segment names.

#define _DEBUG_CHANNEL_SEGMENT_SIZE __segment_size(__DEBUG_CHANNEL__)
Portable segment names.

#define _INTVEC_SEGMENT_SIZE __segment_size(__INTVEC__)
Portable segment names.

#define _CSTACK_SEGMENT_SIZE __segment_size(__CSTACK__)
Portable segment names.

Copyright © 2025 Silicon Laboratories. All rights reserved.

#define _RESETINFO_SEGMENT_SIZE __segment_size(__RESETINFO__)
Portable segment names.

#define _DATA_INIT_SEGMENT_SIZE __segment_size(__DATA_INIT__)
Portable segment names.

#define _DATA_SEGMENT_SIZE __segment_size(__DATA__)
Portable segment names.

#define _BSS_SEGMENT_SIZE __segment_size(__BSS__)
Portable segment names.

#define _CONST_SEGMENT_SIZE __segment_size(__CONST__)
Portable segment names.

#define _TEXT_SEGMENT_SIZE __segment_size(__TEXT__)
Portable segment names.

#define _TEXTRW_INIT_SEGMENT_SIZE __segment_size(__TEXTRW_INIT__)
Portable segment names.

#define _TEXTRW_SEGMENT_SIZE __segment_size(__TEXTRW__)
Portable segment names.

#define _AAT_SEGMENT_SIZE __segment_size(__AAT__)
Portable segment names.

#define _BAT_SEGMENT_SIZE __segment_size(__BAT__)
Portable segment names.

#define _BAT_INIT_SEGMENT_SIZE __segment_size(__BAT_INIT__)
Portable segment names.

#define _FAT_SEGMENT_SIZE __segment_size(__FAT__)
Portable segment names.

#define _RAT_SEGMENT_SIZE __segment_size(__RAT__)
Portable segment names.

#define _SIMEE_SEGMENT_SIZE __segment_size(__SIMEE__)
Portable segment names.

#define _PSSTORE_SEGMENT_SIZE __segment_size(__PSSTORE__)
Portable segment names.

#define _LONGTOKEN_SEGMENT_SIZE __segment_size(__LONGTOKEN__)
Portable segment names.

#define _EMHEAP_SEGMENT_SIZE __segment_size(__EMHEAP__)
Portable segment names.

#define _GUARD_REGION_SEGMENT_SIZE __segment_size(__GUARD_REGION__)
Portable segment names.

#define _DLIB_PERTHREAD_INIT_SEGMENT_SIZE __segment_size(__DLIB_PERTHREAD_INIT__)
Portable segment names.

#define _DLIB_PERTHREAD_INITIALIZED_DATA_SEGMENT_SIZE
__segment_size(__DLIB_PERTHREAD_INITIALIZED_DATA__)
Portable segment names.

#define _DLIB_PERTHREAD_ZERO_DATA_SEGMENT_SIZE __segment_size(__DLIB_PERTHREAD_ZERO_DATA__)
Portable segment names.

Copyright © 2025 Silicon Laboratories. All rights reserved.

#define _INTERNAL_STORAGE_SEGMENT_SIZE __segment_size(__INTERNAL_STORAGE__)
Portable segment names.

#define _LOCKBITS_IN_MAINFLASH_SEGMENT_SIZE __segment_size(__LOCKBITS_IN_MAINFLASH__)
Portable segment names.

#define _UNRETAINED_RAM_SEGMENT_SIZE __segment_size(__UNRETAINED_RAM__)
Portable segment names.

Functions

void _executeBarrierInstructions(void)

Master Variable Types Documentation

boolean

```
typedef bool boolean
```

A typedef to make the size of the variable explicitly known.

int8u

```
typedef unsigned char int8u
```

Denotes that this platform supports 64-bit data-types.

int8s

```
typedef signed char int8s
```

Denotes that this platform supports 64-bit data-types.

int16u

```
typedef unsigned short int16u
```

Denotes that this platform supports 64-bit data-types.

int16s

```
typedef signed short int16s
```

Denotes that this platform supports 64-bit data-types.

int32u

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
typedef unsigned int int32u
```

Denotes that this platform supports 64-bit data-types.

int32s

```
typedef signed int int32s
```

Denotes that this platform supports 64-bit data-types.

int64u

```
typedef unsigned long long int64u
```

Denotes that this platform supports 64-bit data-types.

int64s

```
typedef signed long long int64s
```

Denotes that this platform supports 64-bit data-types.

PointerType

```
typedef unsigned int PointerType
```

Denotes that this platform supports 64-bit data-types.

Miscellaneous Macros Documentation

halInternalAssertFailed

```
void halInternalAssertFailed (const char * filename, int linenumber)
```

A prototype definition for use by the assert macro. (see hal/micro/micro.h)

Parameters

Type	Direction	Argument Name	Description
const char *	N/A	filename	
int	N/A	linenumber	

halInternalResetWatchDog

Copyright © 2025 Silicon Laboratories. All rights reserved.

> void halInternalResetWatchDog (void)

Macro to reset the watchdog timer. Note: be very very careful when using this as you can easily get into an infinite loop if you are not careful.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

External Declarations Documentation

abs

> int abs (int I)

Returns the absolute value of I (also called the magnitude of I). That is, if I is negative, the result is the opposite of I, but if I is nonnegative the result is I.

Parameters

Type	Direction	Argument Name	Description
int	N/A	I	An integer.

Returns

- A nonnegative integer.

Portable segment names Documentation

Function Documentation

_executeBarrierInstructions

> void _executeBarrierInstructions (void)

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Copyright © 2025 Silicon Laboratories. All rights reserved.

Common PLATFORM_HEADER Configuration

Common PLATFORM_HEADER Configuration

Compiler and Platform specific definitions and typedefs common to all platforms.

platform-common.h provides PLATFORM_HEADER defaults and common definitions. This head should never be included directly, it should only be included by the specific PLATFORM_HEADER used by your platform.

See platform-common.h for source code.

Generic Types

#define TRUE 1
An alias for one, used for clarity.

#define FALSE 0
An alias for zero, used for clarity.

#define NULL ((void *)0)
The null pointer.

Bit Manipulation Macros

#define BIT (x)
Useful to reference a single bit of a byte.

#define BIT32 (x)
Useful to reference a single bit of an uint32_t type.

#define SETBIT (reg, bit)
Sets `bit` in the `reg` register or byte.

#define SETBITS (reg, bits)
Sets the bits in the `reg` register or the byte as specified in the bitmask `bits`.

#define CLEARBIT (reg, bit)
Clears a bit in the `reg` register or byte.

#define CLEARBITS (reg, bits)
Clears the bits in the `reg` register or byte as specified in the bitmask `bits`.

#define READBIT (reg, bit)
Returns the value of `bit` within the register or byte `reg`.

#define READBITS (reg, bits)
Returns the value of the bitmask `bits` within the register or byte `reg`.

Byte Manipulation Macros

#define LOW_BYTE (n)
Returns the low byte of the 16-bit value `n` as an `uint8_t`.

#define HIGH_BYTE (n)
Returns the high byte of the 16-bit value `n` as an `uint8_t`.

Copyright © 2025 Silicon Laboratories. All rights reserved.

#define HIGH_LOW_TO_INT (high, low)
Returns the value built from the two `uint8_t` values `high` and `low`.

#define INT8U_TO_INT32U (byte3, byte2, byte1, byte0)
Returns the value built from the four `uint8_t` as an `uint32_t`.

#define BYTE_0 (n)
Returns the low byte of the 32-bit value `n` as an `uint8_t`.

#define BYTE_1 (n)
Returns the second byte of the 32-bit value `n` as an `uint8_t`.

#define BYTE_2 (n)
Returns the third byte of the 32-bit value `n` as an `uint8_t`.

#define BYTE_3 (n)
Returns the high byte of the 32-bit value `n` as an `uint8_t`.

#define BYTE_4 (n)
Returns the fifth byte of the 64-bit value `n` as an `uint8_t`.

#define BYTE_5 (n)
Returns the sixth byte of the 64-bit value `n` as an `uint8_t`.

#define BYTE_6 (n)
Returns the seventh byte of the 64-bit value `n` as an `uint8_t`.

#define BYTE_7 (n)
Returns the high byte of the 64-bit value `n` as an `uint8_t`.

#define COUNTOF (a)
Returns the number of entries in an array.

Time Manipulation Macros

#define elapsedTimeInt8u (oldTime, newTime)
Returns the elapsed time between two 8 bit values. Result may not be valid if the time samples differ by more than 127.

#define elapsedTimeInt16u (oldTime, newTime)
Returns the elapsed time between two 16 bit values. Result may not be valid if the time samples differ by more than 32767.

#define elapsedTimeInt32u (oldTime, newTime)
Returns the elapsed time between two 32 bit values. Result may not be valid if the time samples differ by more than 2147483647.

#define MAX_INT8U_VALUE (0xFF)
Returns true if t1 is greater than t2. Can only account for 1 wrap around of the variable before it is wrong.

#define HALF_MAX_INT8U_VALUE (0×80)
Returns the elapsed time between two 8 bit values. Result may not be valid if the time samples differ by more than 127.

#define timeGTorEqualInt8u (t1, t2)
Returns the elapsed time between two 8 bit values. Result may not be valid if the time samples differ by more than 127.

#define MAX_INT16U_VALUE (0xFFFF)
Returns true if t1 is greater than t2. Can only account for 1 wrap around of the variable before it is wrong.

#define HALF_MAX_INT16U_VALUE (0×8000)
Returns the elapsed time between two 8 bit values. Result may not be valid if the time samples differ by more than 127.

Copyright © 2025 Silicon Laboratories. All rights reserved.

#define timeGTorEqualInt16u (t1, t2)
Returns the elapsed time between two 8 bit values. Result may not be valid if the time samples differ by more than 127.

#define MAX_INT32U_VALUE (0xFFFFFFFFUL)
Returns true if t1 is greater than t2. Can only account for 1 wrap around of the variable before it is wrong.

#define HALF_MAX_INT32U_VALUE (0x80000000UL)
Returns the elapsed time between two 8 bit values. Result may not be valid if the time samples differ by more than 127.

#define timeGTorEqualInt32u (t1, t2)
Returns the elapsed time between two 8 bit values. Result may not be valid if the time samples differ by more than 127.

Miscellaneous Macros

#define UNUSED_VAR (x)

#define DEBUG_LEVEL BASIC_DEBUG
Set debug level based on whether DEBUG or DEBUG_STRIPPED are defined.

#define STATIC_ASSERT (__condition, __errorstr)
Disable static assertions on compilers that don't support them.

Generic Types Documentation

Bit Manipulation Macros Documentation

Byte Manipulation Macros Documentation

Time Manipulation Macros Documentation

Miscellaneous Macros Documentation

Copyright © 2025 Silicon Laboratories. All rights reserved.

NVIC Configuration

NVIC Configuration

Copyright © 2025 Silicon Laboratories. All rights reserved.

Reset Cause Type Definitions

Reset Cause Type Definitions

Copyright © 2025 Silicon Laboratories. All rights reserved.

HAL Utilities

HAL Utilities

Modules

Crash and Watchdog Diagnostics

Cyclic Redundancy Code (CRC)

Random Number Generation

Network to Host Byte Order Conversion

Copyright © 2025 Silicon Laboratories. All rights reserved.

Crash and Watchdog Diagnostics

Crash and Watchdog Diagnostics

Copyright © 2025 Silicon Laboratories. All rights reserved.

Cyclic Redundancy Code (CRC)

Functions that provide access to cyclic redundancy code (CRC) calculation. See crc.h for source code.

Functions

uint16_t halCommonCrc16(uint8_t newByte, uint16_t prevResult)
Calculates 16-bit cyclic redundancy code (CITT CRC 16).

uint32_t halCommonCrc32(uint8_t newByte, uint32_t prevResult)
Calculates 32-bit cyclic redundancy code.

Macros

#define INITIAL_CRC 0xFFFFFFFFL

#define CRC32_START INITIAL_CRC

#define CRC32_END 0xDEBB20E3L

Function Documentation

halCommonCrc16

```
uint16_t halCommonCrc16 (uint8_t newByte, uint16_t prevResult)
```

Calculates 16-bit cyclic redundancy code (CITT CRC 16).

Parameters

Type	Direction	Argument Name	Description
uint8_t	N/A	newByte	The new byte to be run through CRC.
uint16_t	N/A	prevResult	The previous CRC result.

Applies the standard CITT CRC 16 polynomial to a single byte. It should support being called first with an initial value, then repeatedly until all data is processed.

Returns

- The new CRC result.

halCommonCrc32

```
uint32_t halCommonCrc32 (uint8_t newByte, uint32_t prevResult)
```

Calculates 32-bit cyclic redundancy code.

Parameters

Copyright © 2025 Silicon Laboratories. All rights reserved.

Type	Direction	Argument Name	Description
uint8_t	N/A	newByte	The new byte to be run through CRC.
uint32_t	N/A	prevResult	The previous CRC result.

Note

- On some radios or micros, the CRC for error detection on packet data is calculated in hardware.

Applies a CRC32 polynomial to a single byte. It should support being called first with an initial value, then repeatedly until all data is processed.

Returns

- The new CRC result.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Random Number Generation

Random Number Generation

Functions that provide access to random numbers.

These functions may be hardware accelerated, though often are not.

See random.h for source code.

Functions

void halStackSeedRandom(uint32_t seed)
Seeds the halCommonGetRandom() pseudorandom number generator.

uint16_t halCommonGetRandom(void)
Runs a standard LFSR to generate pseudorandom numbers.

Function Documentation

halStackSeedRandom

> void halStackSeedRandom (uint32_t seed)

Seeds the halCommonGetRandom() pseudorandom number generator.

Parameters

Type	Direction	Argument Name	Description
uint32_t	N/A	seed	A seed for the pseudorandom number generator.

Called by the stack during initialization with a seed from the radio.

halCommonGetRandom

> uint16_t halCommonGetRandom (void)

Runs a standard LFSR to generate pseudorandom numbers.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Called by the MAC in the stack to choose random backoff slots.

Complicated implementations may improve the MAC's ability to avoid collisions in large networks, but it is **critical** to implement this function to return quickly.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Network to Host Byte Order Conversion

Network to Host Byte Order Conversion

Functions that provide conversions from network to host byte order. Network byte order is big endian, so these APIs are only necessary on platforms which have a natural little endian byte order. On big-endian platforms, the APIs are macro'd away to nothing. See endian.h for source code.

Functions

uint16_t NTOHS(uint16_t val)
Converts a short (16-bit) value from network to host byte order.

uint32_t NTOHL(uint32_t val)
Converts a long (32-bit) value from network to host byte order.

uint32_t SwapEndiannessInt32u(uint32_t val)

Macros

#define HTONL NTOHL

#define HTONS NTOHS

Function Documentation

NTOHS

> uint16_t NTOHS (uint16_t val)

Converts a short (16-bit) value from network to host byte order.

Parameters

Type	Direction	Argument Name	Description
uint16_t	N/A	val	

NTOHL

> uint32_t NTOHL (uint32_t val)

Converts a long (32-bit) value from network to host byte order.

Parameters

Type	Direction	Argument Name	Description
uint32_t	N/A	val	

SwapEndiannessInt32u

Copyright © 2025 Silicon Laboratories. All rights reserved.

uint32_t SwapEndiannessInt32u (uint32_t val)

Parameters

Type	Direction	Argument Name	Description
uint32_t	N/A	val	

Copyright © 2025 Silicon Laboratories. All rights reserved.

Migration Guide

Migration Guide

Connect Stack 4.1.0

Connect Stack 4.1.0 introduces a key change: projects now use `sl_main` instead of `sl_system` for initialization and main loop control. This section will help you update your project quickly and with confidence.

Detailed API Change

Upgrade to sl_main

Migrating to the new `sl_main` APIs is the best way to ensure long-term support and access to new features.

How to Upgrade

1. **Update your project** In Simplicity Studio, switch your project to the `sisdk-2025.6` version.
2. **Replace API calls and headers** Change all `sl_system` API calls and header includes to their `sl_main` equivalents (see below).
3. **Update initialization and main loop** Follow the migration examples to update your main loop and initialization code.

Upgrading keeps your project compatible with future releases and lets you benefit from the latest improvements.

Header File Changes

Update your includes as follows:

Version 4.0.0 (Old)	Version 4.1.0 (New)
`#include "sl_system_init.h"`	`#include "sl_main_init.h"`
`#include "sl_system_kernel.h"` `#include "sl_system_process_action.h"`	`#include "sl_main_kernel.h"` `#include "sl_main_process_action.h"`

Example of include changes::

```
// Version 4.0.0 (Old)
#include "sl_system_init.h"
#if defined(SL_CATALOG_KERNEL_PRESENT)
  #include "sl_system_kernel.h"
#else
  #include "sl_system_process_action.h"
#endif

// Version 4.1.0 (New)
#include "sl_main_init.h"
#if defined(SL_CATALOG_KERNEL_PRESENT)
  #include "sl_main_kernel.h"
#else
  #include "sl_main_process_action.h"
#endif
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Initialization and Main Loop Changes

Update your main loop depending on whether your project uses an RTOS kernel or is bare metal:

Table of API Changes

Version 4.0.0 (sl_system)	Version 4.1.0 (sl_main)
sl_system_init()	sl_main_init() or sl_main_second_stage_init()
sl_system_kernel_start()	(see new kernel loop below)
sl_system_process_action()	sl_main_process_action()

If Kernel is Present

```
// Version 4.0.0 (Old)
sl_system_init();
app_init();
sl_system_kernel_start();

// Version 4.1.0 (New)
sl_main_second_stage_init();
app_init();
while (sl_main_start_task_should_continue()) {
  app_process_action();
}
```

If Kernel is NOT Present

```
// Version 4.0.0 (Old)
sl_system_init();
app_init();
while (1) {
  sl_system_process_action();
  app_process_action();
}

// Version 4.1.0 (New)
sl_main_init();
app_init();
while (1) {
  sl_main_process_action();
  app_process_action();
}
```

Copyright © 2025 Silicon Laboratories. All rights reserved.

Deprecated List

Deprecated List

- Global EMBER_MAX_SECURED_APPLICATION_PAYLOAD_LENGTH
  The maximum length in bytes of the application payload for a secured message. This define has been deprecated, you should use the emberGetMaximumPayloadLength API instead.
- Global EMBER_MAX_UNSECURED_APPLICATION_PAYLOAD_LENGTH
  The maximum length in bytes of the application payload for an unsecured message. This define has been deprecated, you should use the emberGetMaximumPayloadLength API instead.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Training

Training

This series of tutorials demonstrates "the essentials" of building applications based on Silicon Labs Connect. The collection presents an incremental review of key techniques and features that help developers access the powerful convenience available thru Connect. These tutorials supplement the Developer's Guide and the API reference. Note: The API reference is the authoritative Connect resource, and represents the most current documentation at all times. Grant it a priority in any conflicts you may encounter in the following (or any other) Connect guidance.

Prerequisites

We strongly recommend familiarity with the resources above (especially the Developer's Guide) before beginning Connect-based application development, as they provide insight on crucial decisions that impact the design phase of your project.

That said, each tutorial in this series walks you through important Connect concepts using accessible demonstrations and discussions that illuminate how you can make Connect work for you. To extract the most value from this tutorial series, programming experience in embedded C and (at least) a baseline understanding of wireless networking theory is recommended. See Fundamentals.

The Tutorials

This group of tutorials leads you from "square one" (tutorial 1) to performing packet analysis on traffic captured from the firmware you've developed along the way (tutorial 7). A Direct mode Connect-based application serves as the demonstration vehicle. Though the coverage spans a broad range of topics, the tutorials are designed to be completed sequentially. However, Tutorial 6 is largely independent, and is immediately accessible as a strong subject matter reference.

1. Getting Started with Application Development
2. Communication Basics: Send and Receive
3. Command Line Interface
4. Communication Features: Acknowledge and Message Queue
5. Communication Features: Security
6. IEEE 802.15.4 Addressing
7. Traffic Analysis: Addressing, Acknowledgement, and Security
8. Network Management

Copyright © 2025 Silicon Laboratories. All rights reserved.