AN1412: Zigbee Security Manager

This document describes the Zigbee Security Manager group of components that have been used by Silicon Labs' Zigbee stack for its security operations since EmberZNet 7.2. The components add several new APIs for key access and cryptographic operations, introduced to provide support for the use of secure key storage (see *AN1271: Secure Key Storage*) on supported parts.

This document is mainly focused on describing the components and APIs used to interact with security functionality since Zigbee EmberZNet SDK 7.2. For a general view of the security concepts used within Zigbee and Silicon Labs' Zigbee stack, see *AN1233: Zigbee Security*. For more information on how secure key storage is implemented on supported parts, see *AN1271: Secure Key Storage*.

KEY POINTS

- The Zigbee Security Manager components are now used by the Zigbee stack to manage access to security keys and operations performed with them.
- On parts with Secure Vault-High support, the underlying implementation of the APIs is provided by the PSA Crypto implementation in Secure Vault.
- The new APIs replace legacy Zigbee security API functionality and separate fetching of key data from key metadata.

 Copyright © 2022 by Silicon Laboratories

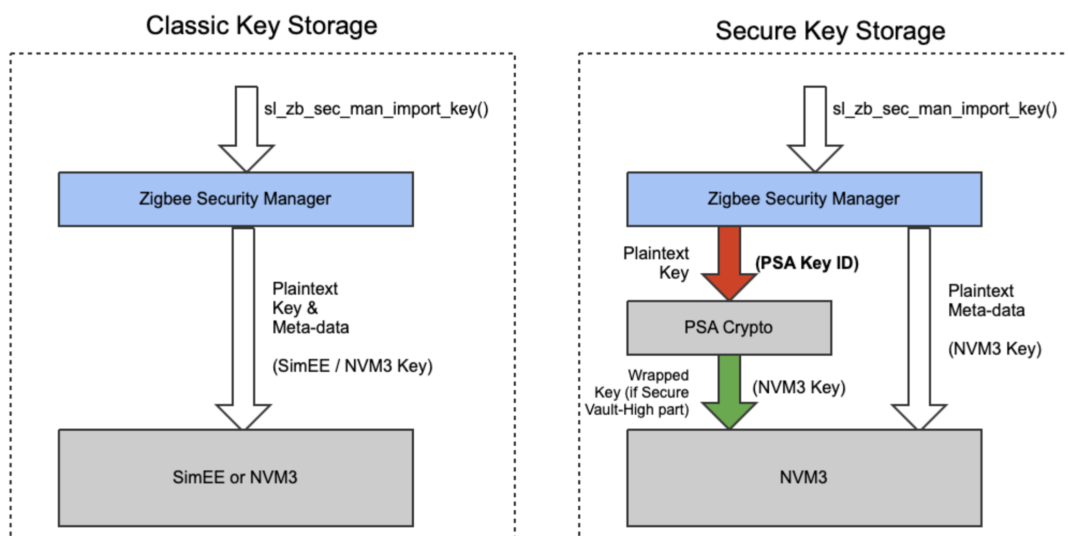
1 Introduction

Versions of EmberZNet since 7.2 have offered the ability to use *AN1271: Secure Key Storage* on supported parts, with the PSA Crypto APIs in Secure Engine being used to implement this. As not all parts support key wrapping, the existing methods of key storage in plaintext NVM3 tokens continue to be maintained. Such builds do not call the PSA Crypto APIs for key storage. The Zigbee Security Manager components were therefore introduced to have a unified set of APIs for the stack to use that more closely aligned with the setup of the PSA Crypto APIs.

This document describes some of the changes involved in using these new APIs to manage key storage and cryptographic operations. Previous APIs are callable in versions through 7.3 with their implementations now routing through Zigbee Security Manager APIs but are scheduled for removal in a future release.

2 Design Concepts

The Zigbee Security Manager components were introduced to facilitate support for storing Zigbee keys inside the secure key storage provided by Secure Vault-High devices (and interacted with via PSA Crypto APIs); versions of EmberZNet prior to 7.2 only stored keys as plaintext in NVM3 tokens. As many currently supported parts do not have Secure Vault-High capabilities, this support has been obfuscated behind Zigbee Security Manager, which connects to the existing method of key storage by default on parts without secure key storage and connects to secure key storage for Secure Vault-High devices. Classic key storage ties in the storage of a key's value and metadata about the key into the same storage location, while secure key storage separates the two, relying on PSA Crypto APIs to store the key value (which allows encrypting it in the process). Definitions for the existing NVM3 tokens remain the same across both versions, with secure key storage leaving the classic token bytes intended for key values unused, done to allow an already deployed device to begin using secure key storage. The following figure shows how keys are stored when using the secure key storage implementation of Zigbee Security Manager, compared to the traditional method of key storage.



2.1 Component List

- Zigbee Security Manager: The base component provides declarations for the relevant APIs, along with implementations for APIs that are common to both the classic and secure key storage variations. This component does not provide a full implementation requiring one of the variation-specific components to complete the implementation.
- Zigbee Security Manager (Host): Contains implementations of the Zigbee Security Manager APIs for host applications in a host-NCP setup. Access to most keys through these APIs is provided as a wrapper around EZSP frames (with the keys being stored on the NCP), while calls to cryptographic operations via this component use a software-based implementation and operate on the host platform. Applications should generally be able to use the same APIs regardless of whether they are running on a host-NCP or SoC setup.
- Zigbee Classic Key Storage: Implementation of Zigbee Security Manager that stores keys in plaintext NVM3, continuing the method that EmberZNet 7.1 and earlier used to store keys. This component is included by default during project creation for any part that does not have Secure Vault-High capabilities.
- Zigbee Secure Key Storage: Implementation of Zigbee Security Manager that uses PSA Crypto APIs for storing keys along with implementing some cryptographic operations used by Zigbee. This component is included by default during project creation for Secure Vault-High devices, although it can be used instead of Classic Key Storage for other parts. On parts without Secure Vault-High, keys stored through the PSA Crypto APIs may not be encrypted.
- Zigbee Secure Key Storage Upgrade: This component allows the migration of keys from classic key storage to secure key storage. If an image with secure key storage has this component present too, a startup function will check classic key storage for stored keys and move them over to secure key storage, at which point the device will act like any other device using secure key storage. This migration process is one-way.

2.2 API Design

The implementation of secure key storage uses PSA Crypto APIs, so Zigbee Security Manager mirrors much of those APIs' setup. Operations involving a key are called by referencing a key identifier, rather than by using a key's value as a parameter. At the Zigbee Security Manager level, these identifiers are set up by a key's type (network key, preconfigured/trust center link key, application link key,

etc.), along with any other metadata needed to unambiguously identify the requested key (typically either the key's index, or the EUI64 address of the device the key is being used to communicate with). A context type, sl_zb_sec_man_context_t, is defined to aggregate this information, and is used for most operations involving Zigbee Security Manager. The PSA Crypto APIs themselves use a 32-bit value to identify keys passed into them; the Zigbee stack uses a range of IDs between 0x00030000 and 0x0003FFFF, but applications using Zigbee Security Manager do not need to keep track of those values for applicable keys. The range between 0x30000000 and 0x3FFFFFFF is reserved for application-managed keys.

Importing a key into Zigbee Security Manager will create or change the key to have the specified value, allowing later operations to use the key by reference to the type it was imported as. For existing installations using classic key storage, all keys present on the device should already be considered as imported and available. This also applies to installations migrating to secure key storage if the upgrade component is present (as the migration process is done automatically).

Exporting a key returns the plaintext value of the key to the caller. The key value remains available in storage for use (exporting a key is not a pop operation - removing a key is done through deleting it, which does not return the deleted key's value). The PSA Crypto APIs do allow users to set whether a key is allowed to be exported from Secure Vault. As for EmberZNet 7.3, Zigbee keys are always set as exportable to handle certain use cases not implemented inside Secure Vault (like the AES-MMO algorithm Zigbee uses for hashing). The design of Zigbee Security Manager has been set up to minimize the need to export keys unnecessarily.

Using a key for a cryptographic operation requires first loading the key, which informs Zigbee Security Manager that the referenced key is the one to start using for operations going forward. This design aspect differs from direct calls to the PSA Crypto APIs, which take in their own key ID type as a parameter; this is done for compatibility with some older APIs (where the load operation directly places the key into an AES encryption engine), and the load function also returns a status to confirm whether a context points to a valid key. After a key is loaded, later calls to operations like AES-CCM encryption will use that key until a different key is loaded or the device is rebooted.

3 Key Operations

A list of all the APIs associated with the Zigbee Security Manager components is located at the Silicon Labs website, under the documentation section https://docs.silabs.com/zigbee/7.3.1/zigbee-stack-api/zigbee-security-manager; this App Note is focused more on documenting the general method of interacting with them, and it differs from what the legacy security APIs use.

3.1 Importing a Key

In this code example, a link key is being imported into Zigbee Security Manager using the generic key import API.

```
//Declare a context struct that will be used to pass in information to the Zigbee Security Manager
API, in this case which type of key is being imported.
sl_zb_sec_man_context_t context;

//Initialize the context to ensure each member of it is set to its default value. In most cases this
is 0, although secure key storage builds set a key's default algorithm permissions (used by PSA Crypto
APIs) to the value necessary to permit AES-CCM encryption with a 4-byte MIC.
sl_zb_sec_man_init_context(&context);

//Set the key type: here it is a persistent link key stored in the table.
context.core_key_type = SL_ZB_SEC_MAN_KEY_TYPE_APP_LINK;

//Set the EUI64 associated with this key.
EmberEUI64 eui = {0x01, 0x23, 0x45, 0x67, 0x89, 0xAB, 0xCD, 0xEF}
MEMMOVE(context.eui64, eui, sizeof(EmberEUI64));

//Import the key.  key is of type sl_zb_sec_man_key_t (containing 16-byte key data as its only field)
sl_zb_sec_man_key_t key = {0xF0, 0xE1, 0xD2, 0xC3, 0xB4, 0xA5, 0x96, 0x87, 0x78, 0x69, 0x5A, \
0x4B, 0x3C, 0x2D, 0x1E, 0x0F};
sl_zb_sec_man_import_key(&context, &key);
```

For link keys of both persistent (stored in the table, not the trust center/preconfigured key) and transient types, shortcut APIs also exist that will manage context setup themselves rather than relying on the caller. In the example below, the same key is imported into index 0, with an API that does not use a context parameter. The EUI64 and key would still have to be initialized, but a context will not be used outside of the API's implementation.

```
sl_zb_sec_man_import_link_key(0, &eui, &key);
```

The context fields that must be set are dependent on the key type; in any case, enough information must be filled out so that the context becomes unambiguous about which stored key it is referencing.

3.2 Exporting a Key

Exporting a key follows a very similar API setup to importing one, although with some differences. The following example illustrates exporting the transient link key that is being used. for temporary communications with a device whose EUI is 0x0123456789ABCDEF. Link keys in persistent and transient tables can also be exported through shortcut APIs. There are separate APIs to export by index and by EUI64.

```
//Declare the context struct.
sl_zb_sec_man_context_t context;

//Initialize the context.
sl_zb_sec_man_init_context(&context);

//Set the key type: here it is a transient key.
context.core_key_type = SL_ZB_SEC_MAN_KEY_TYPE_TC_LINK_WITH_TIMEOUT;

//Set the EUI64 associated with this key.
EmberEUI64 eui = {0x01, 0x23, 0x45, 0x67, 0x89, 0xAB, 0xCD, 0xEF}
MEMMOVE(context.eui64, eui, sizeof(EmberEUI64));

//Mark this context as having its EUI64 field set (so the export operation is aware that it should
```

```
search for this key by its EUI64 value).  This is necessary so an unset key index value doesn't get
treated as a reference to the key at index 0, for example.
context.flags |= ZB_SEC_MAN_FLAG_EUI_IS_VALID;

//Export the key.
sl_zb_sec_man_key_t key;
sl_zb_sec_man_export_key(&context, &key);
```

3.3 Retrieving Key Metadata

Zigbee Security Manager APIs also allow the retrieval of a key's metadata separately from its value, as part of its goal to untangle the two; with only the key's actual value being managed through the PSA Crypto APIs on supported builds.

The shortcut APIs for exporting application and transient link keys also return this information which is stored in sl_zb_sec_man_aps_key_metadata_t for APS key types.

In this example, retrieve the metadata of the link key at index 3 in the table.

```
//Declare context, initialize it, set key type and index.
sl_zb_sec_man_context_t context;
sl_zb_sec_man_init_context(&context);
context.core_key_type = SL_ZB_SEC_MAN_KEY_TYPE_APP_LINK;
context.key_index = 3;

//Mark context's key index field as valid to tell the operation to find the key by index.
context.flags |= ZB_SEC_MAN_FLAG_KEY_INDEX_IS_VALID;

//Retrieve the metadata associated with this key.
sl_zb_sec_man_aps_key_metadata_t metadata;
sl_zb_sec_man_get_aps_key_info(&context, &metadata);
```

3.3.1 Metadata Retrievable

Metadata structures may change in the future, with fields being added or removed. Such changes will be communicated in the release notes.

- For network keys:
  - Whether the key is set/has valid data (both for currently used network key and for the alternate/next one)
  - Sequence numbers for both the current and next keys
  - Frame counter for the current network key
- For link/APS keys:
  - Outgoing and incoming frame counters (when valid/applicable)
  - Remaining time for transient keys
  - Bitmask of other key metadata (using the EmberKeyStructBitmask type)

3.4 Deleting a Key

Deleting a key can be done in a similar method to importing and exporting keys. Indexed link keys, application, and transient types, have their own dedicated APIs. A general sl_zb_sec_man_delete_key exists and will route to these based on the key type from the passed-in context. As with imports and exports, the context will need to be filled with enough information to be unambiguous about the key to be deleted; for indexed keys, a flag may also need to be set to specify which piece of data would identify the key.

3.5 Primary Key Types (as of EmberZNet 7.3)

Key Type	Name in key type enum	Description
Network Key	SL_ZB_SEC_MAN_KEY_TYPE_NETWORK	Used for network-layer encryption. Both the current and next network keys are of this type.
Trust Center Link Key	SL_ZB_SEC_MAN_KEY_TYPE_TC_LINK	Preconfigured link key used for communication with the Trust Center. On a Trust Center node, this will be the device's master key.
Application Link Key	SL_ZB_SEC_MAN_KEY_TYPE_APP_LINK	Indexed link key, stored in persistent table. Used for communication between two non-Trust Center devices, as well as by the Trust Center for the link key each device has with it.
Transient Link Key	SL_ZB_SEC_MAN_KEY_TYPE_TC_LINK_WITH_TIMEOUT	Temporary indexed link key used during the joining process.
Generic / single-use key type	SL_ZB_SEC_MAN_KEY_TYPE_INTERNAL	Temporary key type that allows an arbitrary key value to be placed inside Zigbee Security Manager for access to its cryptographic operations. It is used by the stack for some purposes and should be imported any time it is being used).

There are some other key types recognized by Zigbee Security Manager as well, mainly relating to key types specific to Zigbee Light Link (ZLL) and Green Power, but these types are not mentioned here as they have not been accessible through application APIs.

For reference to versions of EmberZNet before 7.2, the network key and preconfigured link key were traditionally accessed through emberGetKey().

4 Using Zigbee Security Manager for Cryptographic Operations

APIs to carry out cryptographic operations are also present in Zigbee Security Manager; these APIs will use keys that have already been pre-loaded by reference.

4.1 Using AES-CCM to Encrypt some Data

In this example, the key used will be the key that is being stored at index 3.

The algorithm being called in this example uses AES-CCM encryption with a 4-byte MIC appended to the result, as this is the algorithm most used by Zigbee for encryption. It is possible to use a different MIC length with another API, although when using secure key storage this requires that the key to encrypt with was originally imported with permission to use the algorithm of that length.

The example assumes that this key has already been imported. If the goal is to encrypt something with a new key, it will need to be imported before it can be loaded. The context can be reused if both steps are called from the same function.

```
//Declare context, initialize it, set key type and index.
sl_zb_sec_man_context_t context;
sl_zb_sec_man_init_context(&context);
context.core_key_type = SL_ZB_SEC_MAN_KEY_TYPE_APP_LINK;
context.key_index = 3;

//Mark context's key index field as valid to tell the operation to find the key by index.
context.flags |= ZB_SEC_MAN_FLAG_KEY_INDEX_IS_VALID;

//Load the key that this context points to.  The implementation behind loading can vary; classic key
storage copies key bytes into a location where encryption algorithms have expected it, while secure
key storage just saves the PSA ID of the loaded key.
sl_zb_sec_man_load_key_context(&context);

/*Encrypt the desired packet with the key that has been loaded.  The packet has a total length of
packetLength, with its first authDataLength bytes being the associated data used by CCM for authen-
tication, and the remainder being the data to be encrypted. tempBuffer needs to have enough space to
store the result of (packetLength + 4) bytes. */
sl_zb_sec_man_aes_ccm(nonce, true, packet, authDataLength, packetLength, tempBuffer);
```

4.2 Using HMAC to Perform a Keyed Hash AES-MMO

Hashing data follows a similar process to encryption. This example assumes the key to be used in the keyed hash is imported but has not already been loaded; the context setup is only to ensure that sl_zb_sec_man_load_key_context can find the correct key and load it, with multiple operations in the same scope not needing any sort of re-load.

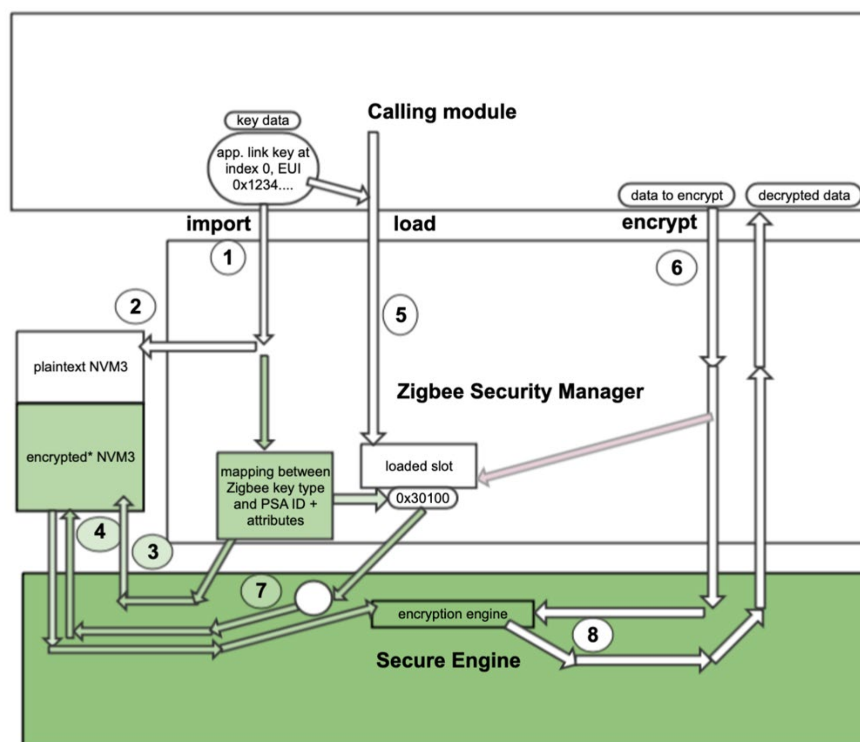
```
//Declare context, initialize it, set key type and index.
sl_zb_sec_man_context_t context;
sl_zb_sec_man_init_context(&context);
context.core_key_type = SL_ZB_SEC_MAN_KEY_TYPE_APP_LINK;
context.key_index = 3;

//Mark context's key index field as valid to tell the operation to find the key by index.
context.flags |= ZB_SEC_MAN_FLAG_KEY_INDEX_IS_VALID;

//Load the key that this context points to.
sl_zb_sec_man_load_key_context(&context);

/*Use HMAC with the loaded key to hash inputData, placing it in result.*/
sl_zb_sec_man_hmac_aes_mmo(inputData, dataLength, result);
```

A visual representation and description of the data flow between modules when calling some Zigbee Security Manager APIs is provided below.



Note: Only encrypted on Secure Vault High parts, but it is stored separately from metadata in any build using Zigbee Secure Key Storage component.

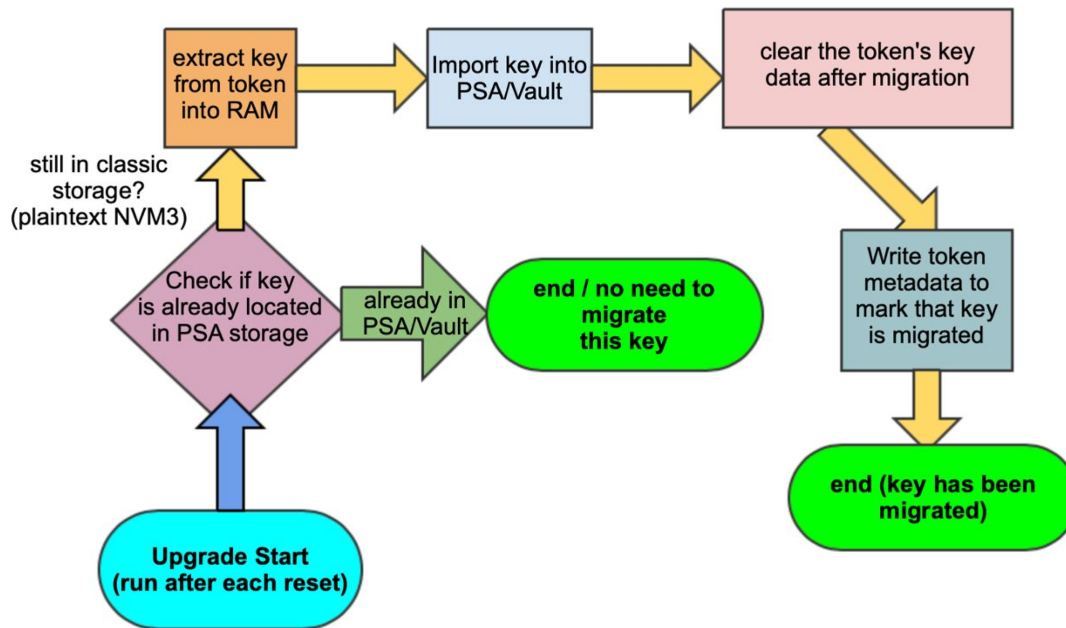
1. Key is imported into Zigbee Security Manager. It is specified as an application link key at index 0; the type and index indicate where this key will be stored when referenced again.
2. Information about the key other than its actual value, like its associated EUI64, is stored in plaintext tokens. Retrieving the metadata stored here is now done through Zigbee Security Manager, but common across both implementations. Classic key storage tied the key's value into the same tokens.
3. Key is brought into secure key storage. The PSA Crypto APIs have their own metadata they expect; Zigbee Security Manager handles that when dealing with key types recognized by Zigbee. In this case, the most important metadata would be the key's ID of 30100 (coming from its storage as an application link key at index 0), and its persistent lifetime. Flags to allow actions like exporting the key back out of the secure engine also get set here.
4. The key data is saved by Secure Engine in an encrypted format (using its root key it will never export). At this point, the application does not need to look at the actual key data again unless it wants to print it for debugging purposes, etc. As of EmberZNet 7.3, some operations still require Zigbee Security Manager to access the key value.
5. Calling the load operation with the same/similar context state that was used to import this key will load it into Zigbee Security Manager. When secure key storage is present, this will just load in the key's PSA ID so later operations will pass that in (as the encryption engine is not interacted with outside of Secure Engine). In contrast, classic key storage has the AES encryption engine synchronized with the loaded slot's key.
6. A call is made to encrypt some piece of data. The key that will be used is the one pointed to by the loaded slot.
7. As part of the encryption call, the key currently pointed to by the loading slot will be read from the wrapped NVM3 storage and brought in for the encryption operation.
8. Decrypted data is sent back to the caller.

5 Upgrading from Classic to Secure Key Storage

From EmberZNet version 7.3 and higher, a method exists to migrate existing keys from classic plaintext storage into secure key storage for existing devices.

To enable key migration, add the Zigbee Secure Key Storage Upgrade component to a project. It is not included by default as the code involved only needs to run once, so it doesn't need to take up space on devices that do not currently require it.

The upgrade process happens upon startup. Each key recognized by Zigbee Security Manager is checked to see whether it is recognized by Secure Vault's PSA Crypto APIs. If the key is already there, then the process moves on to the next key. If not, then the key is extracted out of classic key storage via direct token access, imported into Vault through Zigbee Security Manager; and then the classic token storage is cleared following a successful migration. If the migration process is stopped in an unfinished state for some reason, it continues where it left off upon reset due to its behavior of checking each key independently. The process followed for each key being migrated to PSA storage during the upgrade is provided below.

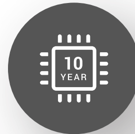


Simplicity Studio

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

IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity



Quality
www.silabs.com/quality

Support & Community
www.silabs.com/community

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

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

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

www.silabs.com