



AN703: Using the Simulated EEPROM for the EM35x and Mighty Gecko SoC Platforms

This document describes how to use the simulated EEPROM in Silicon Labs' EM35x and Mighty Gecko (EFR32MG) platforms, gives an overview of the internal design, and discusses design considerations.

Before reading this document, you should be familiar with the token system in the Ember HAL, inasmuch as primary usage of the simulated EEPROM is driven by the token system. Specifically, an understanding of token definitions, the token API, and the practice of using real EEPROM for non-volatile storage will help you understand this document. For more information on HAL and the token system, see the online HAL API documentation.

KEY FEATURES

- About the Simulated EEPROM
- Differences between Version 1 and Version 2
- Write cycle constraints
- Token definition tips
- Usage overview
- Design constraints

1 Overview

Because EM35x and Mighty Gecko process technology does not offer an internal EEPROM, a simulated EEPROM is implemented to use a section of internal flash memory for stack and application token storage. Parts that use the simulated EEPROM to store non-volatile data have different levels of flash performance with respect to guaranteed write cycles. Recently, version 2 of the simulated EEPROM has been released. For version 1, the EM35x and Mighty Gecko utilize either 4 kB or 8 kB of upper flash memory to store the simulated EEPROM. For version 2, the simulated EEPROM requires 36 kB of upper flash storage. EM35x-I flash cells are qualified for a guaranteed 2,000 write cycles across voltage and temperature; other EM35x flash cells are qualified for a guaranteed 20,000 write cycles, and Mighty Gecko flash cells are qualified for a guaranteed 10,000 write cycles. Due to the limited write cycles, the simulated EEPROM implements a wear-leveling algorithm that effectively extends the number of write cycles for individual tokens. This document addresses simulated EEPROM usage and design considerations from the perspective of the wear-leveling algorithm and its operation.

The simulated EEPROM is designed to operate below the token module as transparently as possible. The application is only required to implement one callback and periodically call one utility function. In addition, a status function is available to provide the application with two basic statistics about simulated EEPROM usage.

2 What is the Simulated EEPROM?

The simulated EEPROM is designed to operate below the token system, so the majority of its use—initialization, getting and setting data, and repairs—is driven by the token system and is otherwise transparent to users. (For information on the token system, see the online HAL API documentation.) The simulated EEPROM is based on dynamic placement of data to maximize wear-leveling effectiveness and to increase system write cycles. The wear-leveling algorithm primarily aims to minimize the number of writes; it does so by only writing fresh data.

The simulated EEPROM requires flash pages to be erased as data moves around. Erasure of flash pages is under the application's control because erasing a page will prevent interrupts from being serviced for 21 ms. Ideally the application will regularly call `halSimEepromErasePage()` when it can. Calling `halSimEepromErasePage()` will return immediately if there are no pages to be erased.

Version 1 is the default choice. Using version 2 requires a special key from Silicon Labs. The only way to downgrade requires full data loss and upgrading might not retain every token.

2.1 Fundamental Differences between Version 1 and Version 2

Version 1 and version 2 share the same fundamental limits on maximum number of tokens, maximum token or element size, and maximum number of elements in an indexed token. Maximum sum total of all token sizes is 2 kB in version 1 and 8 kB in version 2. Version 2 is designed to reduce the maximum time needed when an application writes a token, at the expense of consuming more flash.

The fundamental design difference in version 2 is that not only can there be more data, but “live data” can exist across different virtual pages. This concept means that moving data and erasing hardware pages are accomplished in smaller groupings which take less time. In version 1, as more tokens are stored, token write time increases because an entire virtual page might need to be affected at once.

The rest of Section 2 provides more details explaining how the different versions work.

2.2 Version 1

Simulated EEPROM version 1 will check for existence of Simulated EEPROM version 2 data in the chip. If version 2 data is found, `sim-eeprom.c` will assert. The only way to return a device to Simulated EEPROM version 1 is to explicitly erase Simulated EEPROM data in flash or manually edit `sim-eeprom.c` to remove the asserts.

2.2.1 Virtual Page Memory Management

The simulated EEPROM, version 1, is composed of two virtual pages, and a virtual page is composed of physical hardware pages in flash. For the 8 kB simulated EEPROM, a flash page is 2 kB, a virtual page is 4 kB, and the entire simulated EEPROM uses 8 kB. For the 4 kB simulated EEPROM, a flash page is 2 kB, a virtual page is 2 kB, and the entire simulated EEPROM uses 4 kB. To better understand the simulated EEPROM design, refer to Figure 1 and Figure 2.

Inside a virtual page, the simulated EEPROM maintains a small block of management information that describes what tokens are in the simulated EEPROM. Above this block of management information is the base token storage, which stores a single copy of every token. The rest of the unused flash in the virtual page is available for storing copies of tokens.

To keep memory use efficient, each copy of a token requires just a single 16-bit word of data space as a simple tag to identify the copy. When the set token function is called, only the fresh data that is passed into the function is written into the simulated EEPROM, and only one extra 16-bit word is consumed to tag this fresh data.

Storing copies of token data is the primary reason why write cycle calculations are so difficult. The total life of the simulated EEPROM depends on which tokens are written and how often. A large token can only be written a few times, while a small token can be written many more times in the same available space.

2.2.2 Writing and Erasing

The simulated EEPROM version 1's main operation consists of moving between its two virtual pages, allowing writes to one while erasing the other, as shown in the following figure. A virtual page fills up by writing copies of tokens using the token system API. When the second virtual page is erased and the first is full of data, the simulated EEPROM extracts the management information and the freshest data for all tokens and shifts to the second (erased) virtual page. This shift consists of two operations:

- Writing the management information to the bottom of the virtual page.
- Reconstructing the base token storage using the freshest data for all of the tokens.

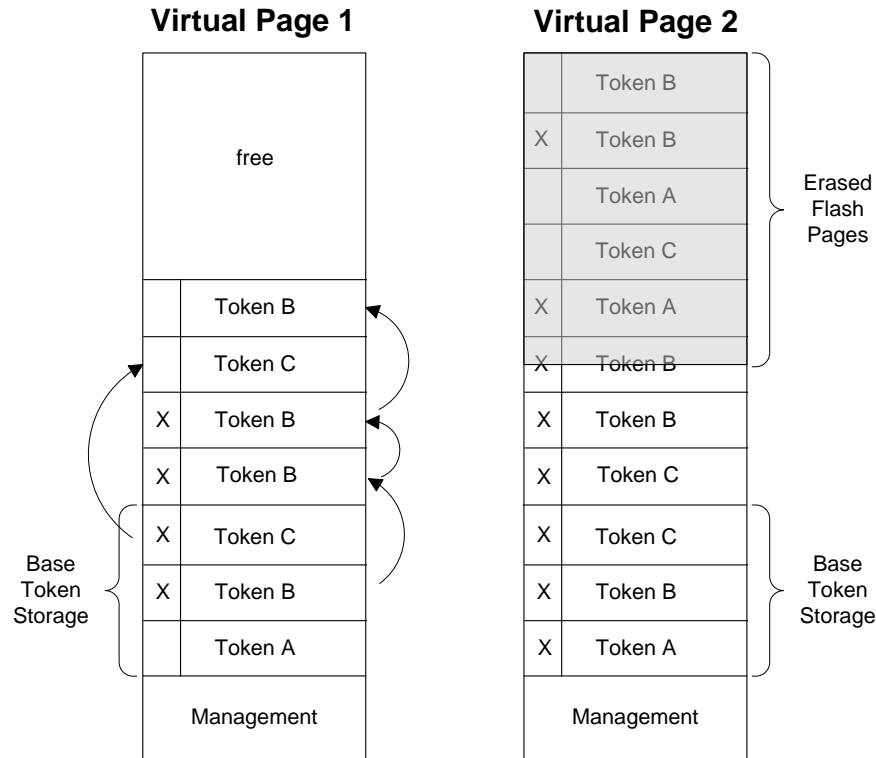


Figure 1. Operation Inside a Page

When the shift is complete, the second virtual page begins to fill up with copies of data while the first virtual page is erased, as shown in the following figure.

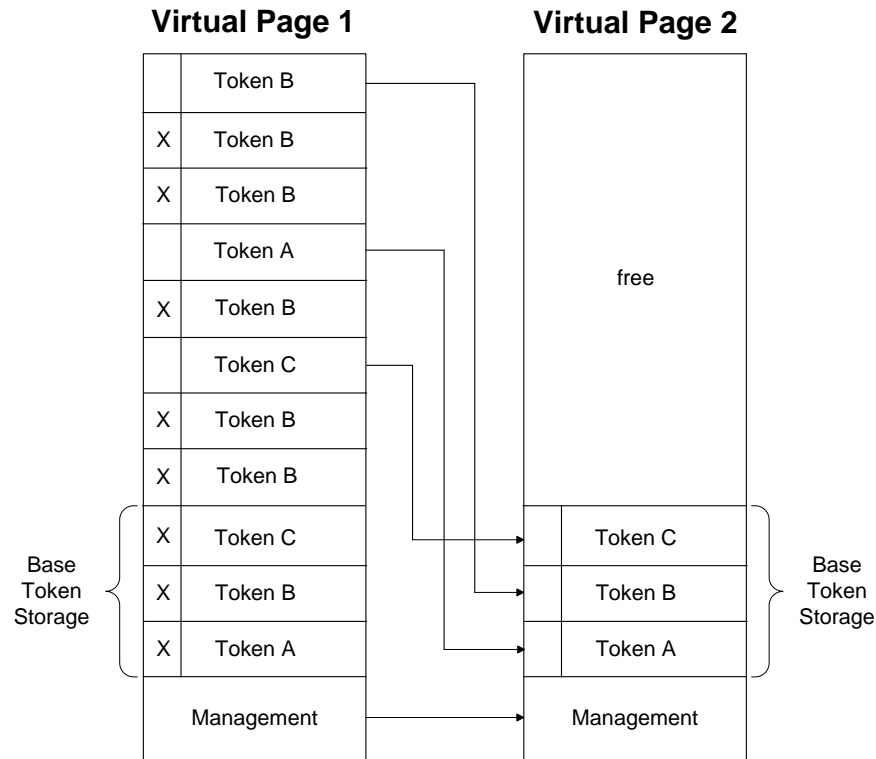


Figure 2. Transitioning Between Pages

2.3 Version 2

2.3.1 Virtual Page Memory Management

The simulated EEPROM, version 2, is composed of three virtual pages, and a virtual page is composed of physical hardware pages in flash. A flash page is 2 kB, a virtual page is 12 kB, and the entire simulated EEPROM uses 36 kB. Because live data is split across two virtual pages, a third virtual page is required to recover and repair a possible failure. While data is rotated among all three virtual pages, one virtual page is always kept erased to recover into.

To better understand the simulated EEPROM design, refer to Figure 3 through Figure 6 in this section. Note that the third page is not shown in the figures, as it is only kept as a repair page.

Inside a virtual page, the simulated EEPROM maintains a small block of management information that describes what tokens are in the simulated EEPROM, including integrity checks. Above this block of management information is the base token storage, which stores a single copy of every token. The rest of the unused flash in the virtual page is available for storing copies of tokens.

Each copy of a token requires 48-bits of data space as a tag. This tag is to identify the data, including an integrity check of the tag itself. When the set token function is called, only the fresh data that is passed into the function is written into the simulated EEPROM, with the tag.

Storing copies of token data is the primary reason why write cycle calculations are so difficult. The total life of the simulated EEPROM depends on which tokens are written and how often. A large token can only be written a few times, while a small token can be written many more times in the same available space.

2.3.2 Writing and Erasing

The main operation of simulated EEPROM version 2 consists of moving between its virtual pages, allowing writes to one while erasing the other.

- A virtual page fills up by writing copies of tokens using the token system API, as shown in the following figure.

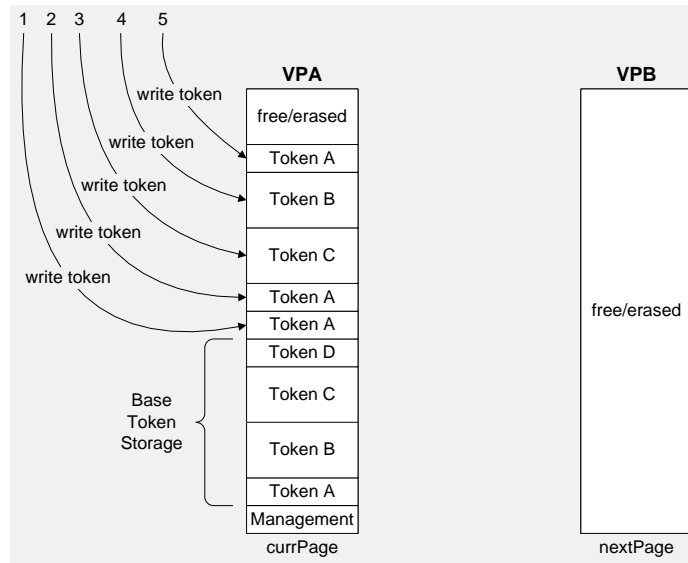


Figure 3. Writes to a Fresh Page

- When a token write doesn't fit in the current virtual page, the management data is copied over to the next page while writing the new data in the next page. In addition to copying over the management data, the latest versions of existing tokens are copied over, as shown in the following figure.

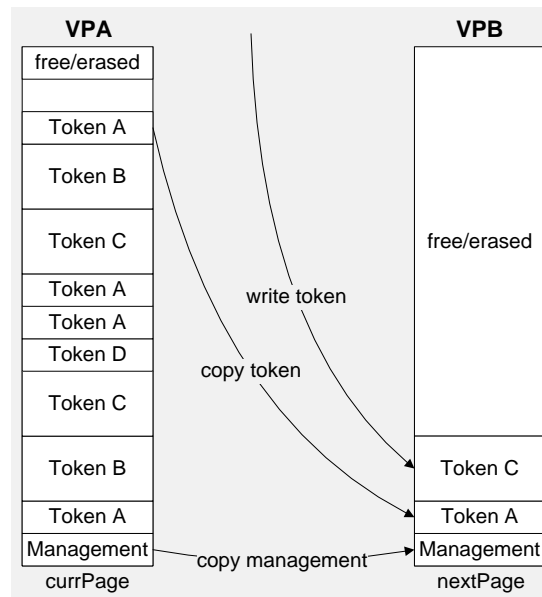


Figure 4. Write of Token C to Available Space with Copy of Management and a Token

- Once the management data has been copied, the algorithm will always copy as much token data as the size of the new token write, as shown in the following figure.

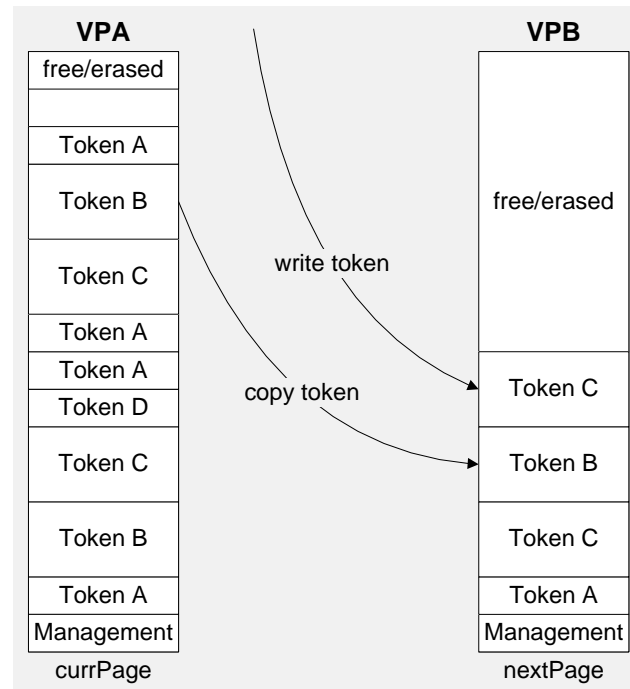


Figure 5. Second Write of Token C with a Copy of a Token

- By copying an existing token(s) alongside a new token write, the freshest token data will be split across multiple virtual pages, keeping the maximum write time capped.
- Once the freshest copy of all tokens exists on the next page, the virtual pages swap current page designation and next page designation.
- With all the tokens existing on the current page, the page containing all the old data will begin to be erased, as shown in the following figure.

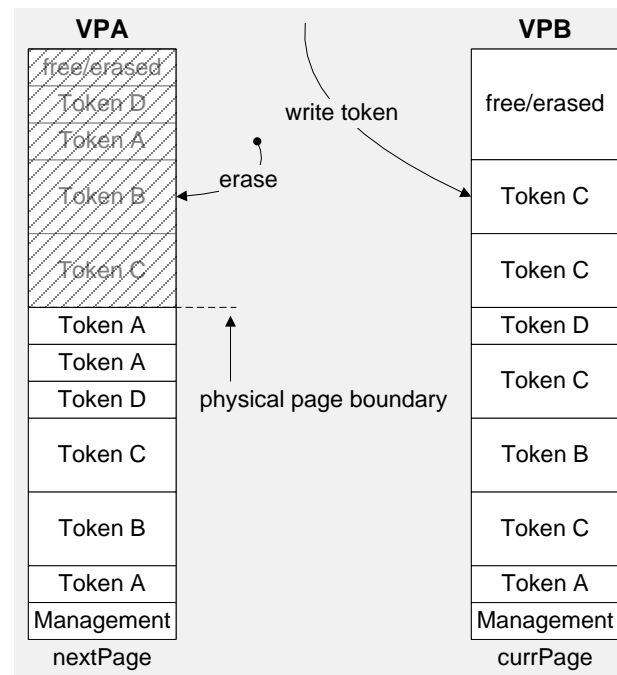


Figure 6. Token C Write and Physical Page Erase

- Token erasing is still under the control of the application. The simulated EEPROM callback will request that the application erase pages.

2.3.3 Enabling Simulated EEPROM, Version 2

Enabling simulated EEPROM, version 2, on SoC platforms requires EmberZNet PRO 5.4.0 or later or Silicon Labs Thread, along with the corresponding version of the Application Builder tool. This combination of software provides an Application Framework plugin called “Simulated EEPROM version 2 Library” that can be enabled with Application Builder to leverage the binary library file provided as `sim-eeprom2-library`. To create a new application configuration in Application Builder that will utilize simulated EEPROM, version 2, follow these steps:

1. In Application Builder's **Plugins** tab, be sure the **Simulated EEPROM Version 1 Library** plugin is disabled.
2. Enable the **Simulated EEPROM Version 2 Library** plugin.
3. In the Password field of the **Simulated EEPROM Version 2 Library** plugin, enter the following string exactly as shown (case-sensitive): `!spoon`
4. If the chip currently has data from Simulated EEPROM version 1 and you want to preserve the data, link in the upgrade library (see section [Upgrade Simulated EEPROM Version 1 to Version 2](#), for more instructions). If the chip does not have any version 1 data, or if you don't care about preserving any, then link in the upgrade stub library instead of the upgrade library.

2.3.4 Upgrade Simulated EEPROM Version 1 to Version 2

It is possible to upgrade simulated EEPROM version 1 to version 2 using the **Simulated EEPROM Version 1 to Version 2 Upgrade Library** found in Application Builder. This upgrade library can only be linked into applications that are built with the **Simulated EEPROM Version 2 Library**. The upgrade library is a companion to the main **Simulated EEPROM Version 2 Library** and cannot be run on its own.

Like every type and instance of the simulated EEPROM, the tokens defined in the application must match the tokens already stored in flash. Only the tokens defined in the application matching the tokens that exist in flash can be upgraded to version 2. Any new tokens and tokens that do not match will be installed with their default definitions. Tokens in flash that do not have a matching definition in the application will be ignored and eventually erased.

The upgrade library runs a simplified copy of the simulated EEPROM version 1 to find existing tokens. Simulated EEPROM version 2 then takes over and performs a standard repair process to reformat version 1 tokens into version 2 formatting.

The upgrade is a one-way process that only has to be run once per device to move the tokens in flash into the proper format and locations. Once an upgrade has been performed on a device, any further reboots of a device will check to see if simulated EEPROM version 1 tokens exist and not take any further upgrade action. After an upgrade has been performed on a device, the application can be recompiled using the upgrade stub library to save space.

2.3.5 Downgrade Simulated EEPROM Version 2 to Version 1

Downgrading requires full data loss. There are four means of downgrading:

1. In Application Builder's **Plugins** tab, **Simulated EEPROM Version 1 Library** plugin has a checkbox option for Destructive SimEE version 2 to version 1 downgrade
2. Manually add the preprocessor define `DESTRUCTIVE_SIMEE2TO1_DOWNGRADE` to your build
3. Manually edit `sim-eeprom.c` to remove the asserts that check for version 2
4. Manually and explicitly erase simulated EEPROM data in flash

All data will be lost and recreated from default values in the application.

3 Write Cycle Constraints

Because flash cells are qualified for a guaranteed 2,000, 10,000, or 20,000 write cycles, the simulated EEPROM implements a wear-leveling algorithm that effectively extends the number of write cycles for individual tokens. The number of set token operations is finite due to flash write-cycle limitations. It is impossible to guarantee an exact number of set token operations because the life of the simulated EEPROM depends on which tokens are written and how often.

Estimating write cycles can be very difficult and inaccurate because every application, token setup, and chip can be different along with the variability of the larger environment after deployment into the field. In general, the number of token writes is on the order of hundreds of thousands and up to millions. Estimating write cycles gives a false sense of certainty.

4 Token Definition Tips

There are two basic types of tokens:

- Non-indexed tokens, also called scalar tokens, can be thought of as a simple char variable type.
- Indexed tokens can be considered an 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.

The token system provides separate API functions for non-indexed and indexed tokens, which are not interchangeable.

The following sections provide recommendations on token definitions:

- Structured types and byte alignment
- When to define a counter token
- Array tokens versus indexed tokens
- Unused and zero-sized tokens

4.1 Structured Types and Byte Alignment

To maximize the write cycles and efficiency of the simulated EEPROM, avoid using extraneous memory wherever possible. The following is a subtle example from EmberZNet PRO 5.7 of avoiding extraneous memory by defining a structured token type.

```
typedef struct {  
    uint8_t dataA;  
    uint16_t dataB;  
    uint8_t dataC;  
} tokTypeData
```

In this example, the size of the struct is only 4 bytes, and the simulated EEPROM stores data as 16-bit quantities, so each copy of this data in the flash should only use two 16-bit words. However, this is inefficient and costs an extra word of storage because the compiler's alignment of data in memory performs a direct translation of this structure. The compiler places a padding byte after `dataA`, so `dataB` is word aligned. This forces `dataC` into a third word, where another padding byte is placed after `dataC`. Altogether, this structure consumes three 16-bit words of memory.

If `dataB` precedes `dataA`, the compiler places `dataB` as the first word and packs `dataA` and `dataC` into a single word following `dataB`. The following version uses only two words, 4 bytes of storage:

```
typedef struct {  
    uint16_t dataB;  
    uint8_t dataA;  
    uint8_t dataC;  
} tokTypeData
```

4.2 When to Define a Counter Token

A counter token is a non-indexed token that is meant to store a number. This number is most often incremented as opposed to explicitly set. A counter token can only be non-indexed, and the token API function `halCommonIncrementCounterToken()` can only operate on counter tokens. Counter tokens must be 32-bits.

A token should only be declared as a counter when the function `halCommonIncrementCounterToken()` is used on the token. A counter token receives special memory storage. To increase the density of stored information in the simulated EEPROM, counter tokens consume an extra 50 bytes for each instance. These extra 50 bytes of storage are meant to store +1 markers. If an application declares a 32-bit counter token as a normal token and wants to increment this counter 50 times, 200 bytes are consumed. However, if the application declares this token as a counter and then increments the counter 50 times, only 50 bytes are consumed.

When a token is declared as a counter and incremented significantly more than it is set, the write-cycle density is greatly increased due to the extra efficiency in storing the number. Conversely, if a token is declared as a counter and never incremented but only set, the write-cycle density decreases dramatically due to the extra 50 bytes that are consumed for each token instance.

4.3 Array Tokens Versus Indexed Tokens

The difference between an array token (a scalar token with a type that is an array) and an indexed token is subtle but important.

Array Token

The following example defines an array token:

```
typedef uint8_t tokTypeAdata[10];
DEFINE_BASIC_TOKEN(ADATA, tokTypeAdata, {0,})
```

In this case, the fifth byte of token ADATA is read as follows:

```
halCommonGetToken(&data, TOKEN_ADATA);
```

Byte 5 is accessed with a local variable such as `data[5]`.

Index Token

In the following example, the same data is defined an indexed token:

```
typedef uint8_t tokTypeIdata;
DEFINE_INDEXED_TOKEN(IDATA, tokTypeIdata, 10, {0,})
```

In this case, the fifth byte of token IDATA is read as follows:

```
halCommonGetIndexedToken(&data, TOKEN_IDATA, 5);
```

The local variable already holds byte 5 and only byte 5.

Differences in EEPROM Storage

The simulated EEPROM stores these two tokens with significant differences:

- A BASIC token is stored as a chunk of data with a management word attached to it.
- An INDEXED token is broken up and each element of the token is stored internally as a basic token.

Both methods allow grouping similar data together under a single token name. In general, grouped data should be stored as a BASIC token only when all of the grouped data will commonly change in unison. Alternatively, if each piece of the grouped data needs to change independently of the others, the token should be declared as INDEXED.

In the default set of stack tokens there are good examples of BASIC versus INDEXED tokens:

- `TOKEN_STACK_NODE_DATA` contains six different pieces of data, but all six are always changed together. Therefore, the token is declared as BASIC and the token's type is a structure that contains each piece of data.
- `TOKEN_STACK_BINDING_TABLE` is an INDEXED token because it contains multiple entries that change independently of each other.

4.4 Unused and Zero-Sized Tokens

You should remove declarations for unused tokens from the system. Otherwise, their default values remain set and they simply consume flash storage and reduce available write cycles.

Indexed tokens with an array size of zero can be allowed to stay; however, they should be removed if they are always zero-sized. A zero-sized token does not consume any storage for its data; however, in version 1 it consumes two 16-bit words for management data and in version 2 it consumes 48-bits for management data. Maintaining management data is necessary since the simulated EEPROM must always know the token exists, whether empty or not.

5 Usage Overview

Only three simulated EEPROM functions are exposed to the application:

- `halSimEepromCallback()`
- `halSimEepromErasePage()`
- `halSimEepromStatus()`

Prototypes and in-code descriptions for these functions can be found in `hal/micro/sim-eeprom.h`.

`halSimEepromCallback()` is critical because the application is responsible for periodically erasing flash pages by calling `halSimEepromErasePage()` to allow the wear-leveling algorithm to continue to operate. The application must control when to erase pages, because an erase operation renders EM35x completely unresponsive for 21 milliseconds while the flash is busy.

In summary, when a token is set, the simulated EEPROM determines if a page needs to be erased, and if so, the callback is triggered by one of the following `EmberStatus` codes:

- `EMBER_SIM_EEPROM_ERASE_PAGE_GREEN` indicates that there is still room available to the wear-leveling algorithm.
- `EMBER_SIM_EEPROM_ERASE_PAGE_RED` indicates that the simulated EEPROM is nearly out of room for the wear-leveling algorithm to continue, and that data loss due to a full simulated EEPROM is possible.

The application should always call `halSimEepromErasePage()` when one of these two status codes return, in order to maintain ample room for the simulated EEPROM.

The application should take care only to call `halSimEepromErasePage()` while it is in a state where it can be unresponsive for an extended period of time. `halSimEepromErasePage()` is typically called from the callback as needed. If the application is concerned about being unresponsive at inopportune times, Silicon Labs recommends two methods:

- The application calls `halSimEepromErasePage()` whenever it can and as often as it can, inasmuch as the function only executes when necessary.
- The callback sets a simple flag whenever it requests a page be erased. The application can then check this flag when it is safe to do so, perform the erase, and clear the flag when the erase operation is complete. For example, the application can safely proceed with an erase during sleeping or waking sequences when the network is inactive.

The following sections provide more detail on the three functions.

5.1 `halSimEepromCallback`

```
void halSimEepromCallback(EmberStatus status)
```

The simulated EEPROM callback function must be implemented by the application. Because the majority of implementations follow the same basic pattern, a default instance is provided in `hal/ember-configuration.c`. A page erase operation causes the processor to ignore interrupts for 21 milliseconds while the flash is busy.

If the application has specific timing requirements and must tightly control when the simulated EEPROM performs a page erasure, the application can implement a custom callback handler and override the default implementation by defining the macro `EMBER_APPLICATION_HAS_CUSTOM_SIM_EEPROM_CALLBACK`. The primary purpose of the callback is to inform the application about the status of the simulated EEPROM and the need for erasing flash pages. The callback always reports one of the following five possible `EmberStatus` codes:

- `EMBER_SIM_EEPROM_ERASE_PAGE_GREEN`
- `EMBER_SIM_EEPROM_ERASE_PAGE_RED`
- `EMBER_SIM_EEPROM_FULL`
- `EMBER_ERR_FLASH_WRITE_INHIBITED`
- `EMBER_ERR_FLASH_VERIFY_FAILED`

This callback is critical because the application is responsible for periodically erasing flash pages by calling `halSimEepromErasePage()`, so the wear-leveling algorithm can continue to operate.

It is best for all applications to regularly call `halSimEepromErasePage()` when they can. Calling `halSimEepromErasePage()` will return immediately if there are no pages to be erased.

EMBER_SIM_EEPROM_ERASE_PAGE_GREEN

EMBER_SIM_EEPROM_ERASE_PAGE_GREEN indicates a page needs to be erased, but the simulated EEPROM has enough space available for the wear-leveling algorithm to continue. Ideally the application will always erase a page on this indication, if not before. The application can safely defer or re-schedule the lengthy page erase operation until it is convenient. The application has control of erasing pages because an erase operation causes EM35x to ignore interrupts for 21 milliseconds while the flash is busy.

EMBER_SIM_EEPROM_ERASE_PAGE_RED

EMBER_SIM_EEPROM_ERASE_PAGE_RED indicates that data loss is imminent due to a full simulated EEPROM. Pages must be erased immediately. Call `halSimEepromErasePage()` to prevent data loss.

Because calling `halSimEepromErasePage()` will return immediately if there is nothing to be erased, it is safe to call this function until it returns 0, meaning no more pages to be erased. For more information, see section [halSimEepromErasePage](#).

Note: An application that erases pages regularly should never see a status of EMBER_SIM_EEPROM_RED.

EMBER_SIM_EEPROM_FULL

When the simulated EEPROM becomes full and the application or stack attempts to set another token, the callback is triggered with an `EmberStatus` of EMBER_SIM_EEPROM_FULL. Because the simulated EEPROM is full, nothing can be done for the current set token call and the data being set is dropped.

If this status is passed to the callback, the application should immediately call `halSimEepromErasePage()` for all the physical pages that comprise the virtual page. Doing so enables the simulated EEPROM to continue operating.

Because calling `halSimEepromErasePage()` will return immediately if there is nothing to be erased, it is safe to call this function until it returns 0, meaning no more pages to be erased. For more information, see section [halSimEepromErasePage](#).

Note: An application that erases pages regularly should never see a status of EMBER_SIM_EEPROM_FULL.

EMBER_ERR_FLASH_WRITE_INHIBITED

This `EmberStatus` code indicates that the flash library inhibited the write attempt due to data already existing at the desired address. This error code generally indicates that there is stale data in a portion of the simulated EEPROM that is supposed to be empty and unused. This indicates the simulated EEPROM was fatally interrupted during an earlier write attempt and that it must now be repaired to recover from this error. The callback must now call the function `halInternalSimEeRepair(FALSE)` to resolve this error. To prevent possible reentrance of the repair function, wrap the call to `halInternalSimEeRepair(FALSE)` with a static or global flag. After the repair, the callback must now reset the micro with the HAL function call `halInternalSysReset(RESET_FLASH_INHIBIT)` due to the lost token data as well as to trigger proper initialization.

EMBER_ERR_FLASH_VERIFY_FAILED

If the simulated EEPROM ever fails to write to flash—which eventually happens when the write cycles of the flash are exceeded—the simulated EEPROM reports this error to the callback.

Because the wear-leveling algorithm of the simulated EEPROM evenly spreads flash usage, the algorithm can operate cleanly for a long time. Eventually, however, the write-cycle limit is exceeded. Assume every address inside the simulated EEPROM will fail at nearly the same time. This characteristic of the simulated EEPROM's end of life means that the simulated EEPROM cannot recover if it fails to write to flash. There is no means or method of working around failed addresses.

While the callback can still first call `halInternalSimEeRepair(FALSE)` and then or the HAL function `halInternalSysReset(RESET_FLASH_VERIFY)` in an attempt to keep using non-volatile storage, it can no longer guarantee that the data is safe. If this error code is ever seen, the application should cease all operations involving tokens (including the use of the networking stack) and default into a safe mode.

5.2 halSimEepromErasePage

```
uint8_t halSimEepromErasePage(void)
```

Returns a count of how many pages are left to be erased. This value allows for calling code to easily loop until all pages due for erasure are erased.

This function instructs the simulated EEPROM to erase a page. The simulated EEPROM does not erase a page unless explicitly told to do this by this function call, even if failure to do so might cause data loss. Because of the length of time required to erase a page, this decision is left to the discretion of the application.

This function is typically called from within `halSimEepromCallback()`, but it can and should be called from anywhere at any time convenient for the application. When this function is called, the simulated EEPROM only performs a page erasure if needed, and each call to this function only erases a single page.

Because calling `halSimEepromErasePage()` will return immediately if there is nothing to be erased, it is safe to call this function anytime.

Note: Erasing a page of flash takes 21 milliseconds, during which the processor ignores interrupts. The application should take care to call this function only when it can afford to be unresponsive for a long period—for example, during sleeping or wakeup sequences when the network is not active.

5.3 `halSimEepromStatus`

```
void halSimEepromStatus
(uint16_t * freeWordsInCurrPage, uint16_t * totalPageUseCount)
```

This function returns two basic metrics about the simulated EEPROM's current state:

- `freeWordsInCurrPage` returns the number of free words on the current page.
- `totalPageUseCountTotal` returns the total number of pages used.

The simulated EEPROM moves data between two virtual pages: while one virtual page fills with set token calls, the other virtual page is erased by `halSimEepromErasePage()` (see section [What is the Simulated EEPROM?](#) for more information). The two metrics returned by this function provide insight into movement between the virtual pages.

`freeWordsInCurrPage`

The variable `freeWordsInCurrPage` equals the difference between the last location of written data and the end of the virtual page - that is, the number of unused words (16-bits) in the virtual page that are being written. This metric can be used to determine how many set token calls are still available in the current virtual page. By dividing this variable by the size of tokens expected to be written, you can estimate how many set token calls remain for this page.

`totalPageUseCount`

The variable `totalPageUseCount` indicates how many times the simulated EEPROM switched virtual pages to continue setting additional tokens. This value lets you obtain a very rough approximation of write cycles and to allow for time-to-failure calculations. The flash cells are qualified for up to 2,000, 10,000, or 20,000 write cycles, so under ideal conditions this variable is read at least 4,000, 20,000 or 40,000 times before the flash cells fail, as two virtual pages are used.

6 Design Constraints

The following sections describe some inherent design constraints of the simulated EEPROM:

- Numerical parameters
- About simulated EEPROM repairs

6.1 Numerical Parameters

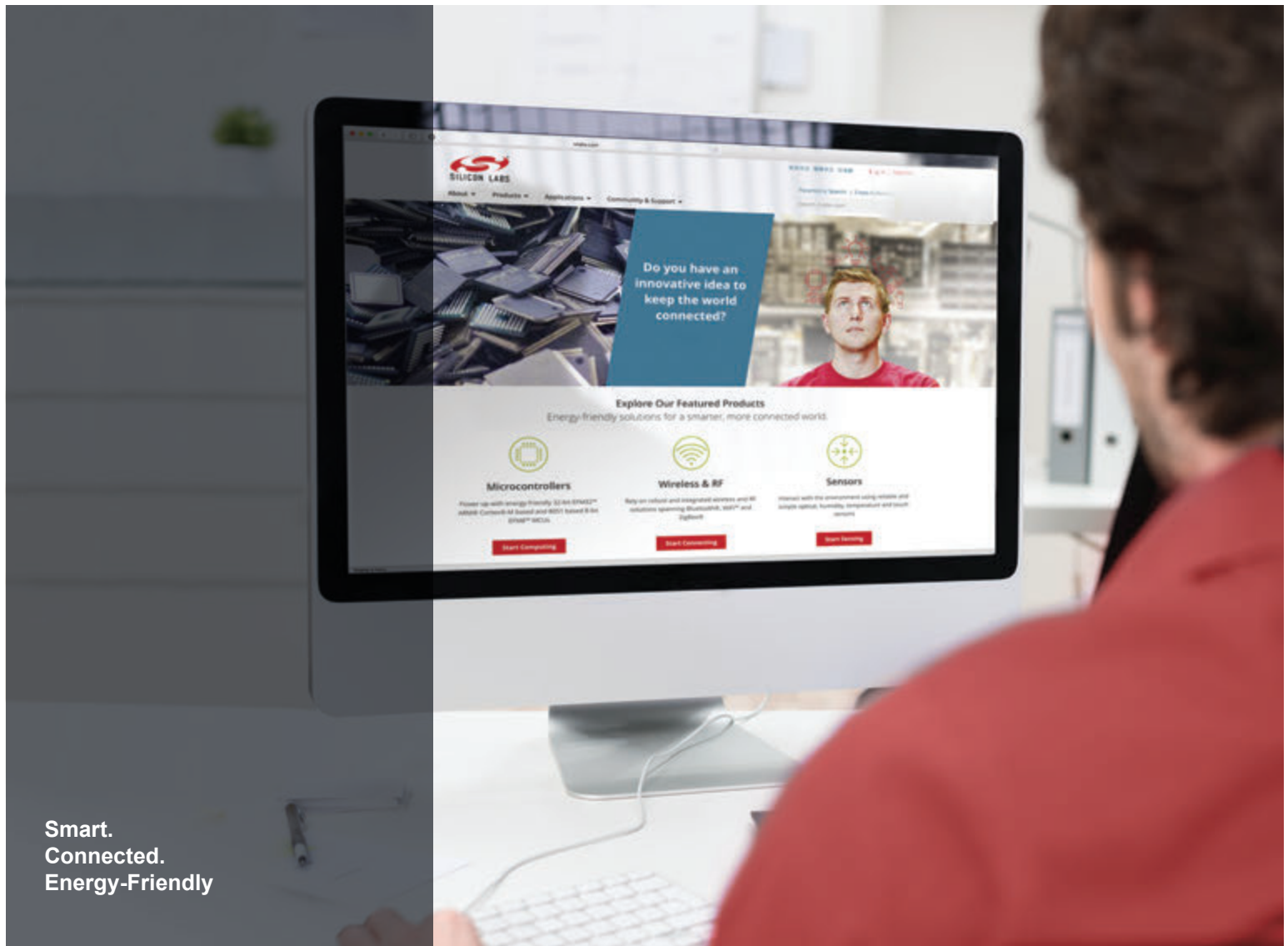
A critical set of numerical parameters define and characterize the simulated EEPROM. The four maximum values shown below are due to internal variable sizes, and the simulated EEPROM is protected from values exceeding these parameters. The timing parameters are given only as a design reference; there is a wide range of operation timing that you should be aware of. Other than write cycles, these parameters do not change due to environmental characteristics.

- Maximum number of tokens: 255
- Maximum number of elements in an indexed token: 126
- Maximum token or element size: 254
- Maximum sum total of all token sizes for version 1: 2 kB
- Maximum sum total of all token sizes for version 2: 8 kB
- Average read time of one 26 byte token/element: 250 microseconds
- Average write time of one 26 byte token/element: 1.5 milliseconds
- Worst-case write time of one token/element: 75 milliseconds
- Erase time of one page: 21 milliseconds
- Best-case simulated EEPROM initialization time: 1.4 milliseconds
- Worst-case simulated EEPROM initialization time without repairing: 30 milliseconds
- Worst-case simulated EEPROM initialization time with repairing: >200 milliseconds
- Write cycles: See section [Write Cycle Constraints](#).

6.2 About Simulated EEPROM Repairs

The worst-case initialization time with repairing is much larger than the worst case initialization time without repairing, due to the extra processing required when repairing. The simulated EEPROM stores token data using a relationally dependent mechanism that allows token data and management information to be packed as closely together as possible. So, if a token is added, a token is deleted, or a token's size is changed, the simulated EEPROM must recalculate the relationship between all of the tokens.

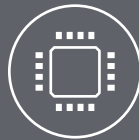
The recalculation is called the Repair function and is automatically performed as needed during the initialization sequence. The Repair function attempts to maintain the integrity of as much data as possible. Deleting a token naturally deletes the token data. Adding or changing a token causes token data to be set to the default value in the token definition.



Smart.
Connected.
Energy-Friendly



Products
www.silabs.com/products



Quality
www.silabs.com/quality



Support and Community
community.silabs.com

Disclaimer

Silicon Laboratories 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 Laboratories 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 Laboratories reserves the right to make changes without further notice and limitation to product information, specifications, and descriptions herein, and does not give warranties as to the accuracy or completeness of the included information. Silicon Laboratories shall have no liability for the consequences of use of the information supplied herein. This document does not imply or express copyright licenses granted hereunder to design or fabricate any integrated circuits. The products are not designed or authorized to be used within any Life Support System without the specific written consent of Silicon Laboratories. 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 Laboratories products are not designed or authorized for military applications. Silicon Laboratories products shall under no circumstances be used in weapons of mass destruction including (but not limited to) nuclear, biological or chemical weapons, or missiles capable of delivering such weapons.

Trademark Information

Silicon Laboratories Inc.®, Silicon Laboratories®, Silicon Labs®, SiLabs® and the Silicon Labs logo®, Bluegiga®, Bluegiga Logo®, Clockbuilder®, CMEEMS®, DSPLL®, EFM®, EFM32®, EFR®, Ember®, Energy Micro, Energy Micro logo and combinations thereof, "the world's most energy friendly microcontrollers", Ember®, EZLink®, EZRadio®, EZRadioPRO®, Gecko®, ISOModem®, Precision32®, ProSLIC®, Simplicity Studio®, SiPHY®, Telegesis, the Telegesis Logo®, USBXpress® and others are trademarks or registered trademarks of Silicon Laboratories Inc. ARM, CORTEX, Cortex-M3 and THUMB are trademarks or registered trademarks of ARM Holdings. Keil is a registered trademark of ARM Limited. All other products or brand names mentioned herein are trademarks of their respective holders.



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

<http://www.silabs.com>