



# Using the Simulated EEPROM

## For the EM250 and EM35x SoC Platforms

This document describes how to use the Simulated EEPROM in the EM250 and EM35x, gives an overview of the internal design, and discusses design considerations.

Before reading this document, you should be familiar with the token system in EmberZNet HAL 3.0 or later, 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 3.0 and the token system, see the online HAL API documentation.

### Contents

Overview .....	2
What Is Simulated EEPROM? .....	2
Write Cycle Constraints .....	4
Token Definition Tips .....	7
Usage Overview .....	9
Design Constraints .....	13
Erase Timing and Callback Implementation .....	14



## Overview

Because EM250 and EM35x 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. The EM250 utilizes 8kB of upper flash to store this non-volatile data, and the EM250 flash cells are qualified for a guaranteed 1,000 write cycles across voltage and temperature. The EM250 does not offer a 4kB Simulated EEPROM. The EM35x utilizes either 8kB or 4kB of upper flash to store this non-volatile data, and the EM35x flash cells are qualified for a guaranteed 20,000 write cycles across voltage and temperature. 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.

## What Is 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. To better understand the Simulated EEPROM design, refer to Figure 1 and Figure 2.

### Virtual page memory management

The Simulated EEPROM is comprised of two virtual pages, and a virtual page is comprised of physical hardware pages in flash. Due to the different flash architecture, the EM250 8kB Simulated EEPROM virtual pages differ from the EM35x 8kB and 4kB Simulated EEPROM virtual pages. For the EM250 8kB Simulated EEPROM, a flash page is 1kB, a virtual page is 4kB, and the entire Simulated EEPROM uses 8kB. For the EM35x 8kB Simulated EEPROM, a flash page is 2kB, a virtual page is 4kB, and the entire Simulated EEPROM uses 8kB. For the EM35x 4kB Simulated EEPROM, a flash page is 2kB, a virtual page is 2kB, and the entire Simulated EEPROM uses 4kB.

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 word of data as a simple tag to identify the data. 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 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, because 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.

### Simultaneous writing and erasing

The Simulated EEPROM's main operation consists of moving between its two virtual pages, allowing writes to one while erasing the other (see Figure 1). 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.

When the shift is complete, the second virtual page begins to fill up with copies of data while the first virtual page is erased (see Figure 2).

Figure 1. Operation inside a page

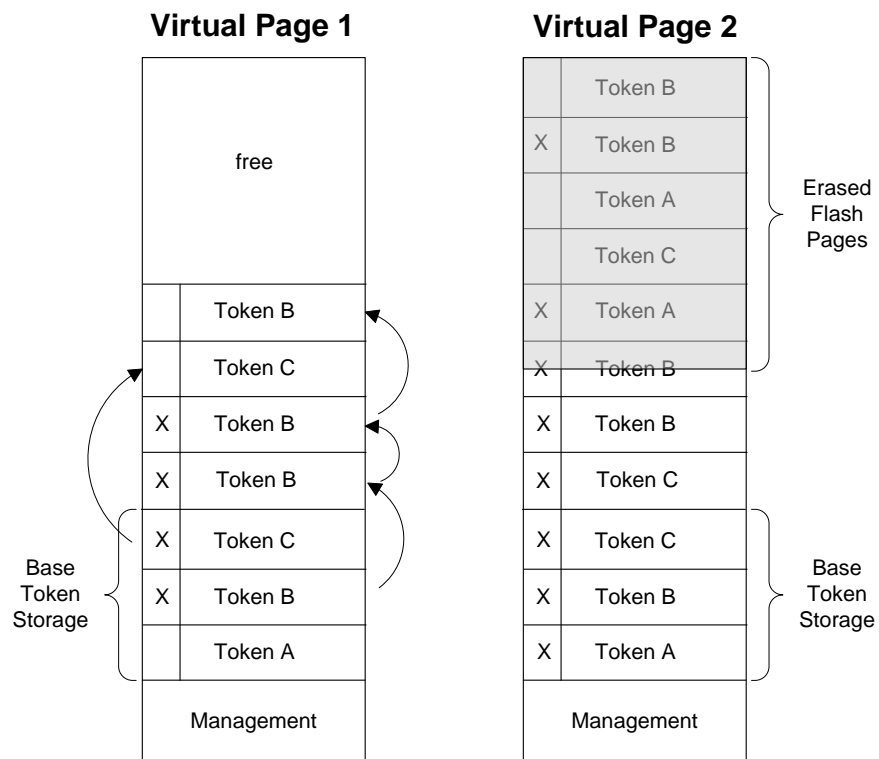
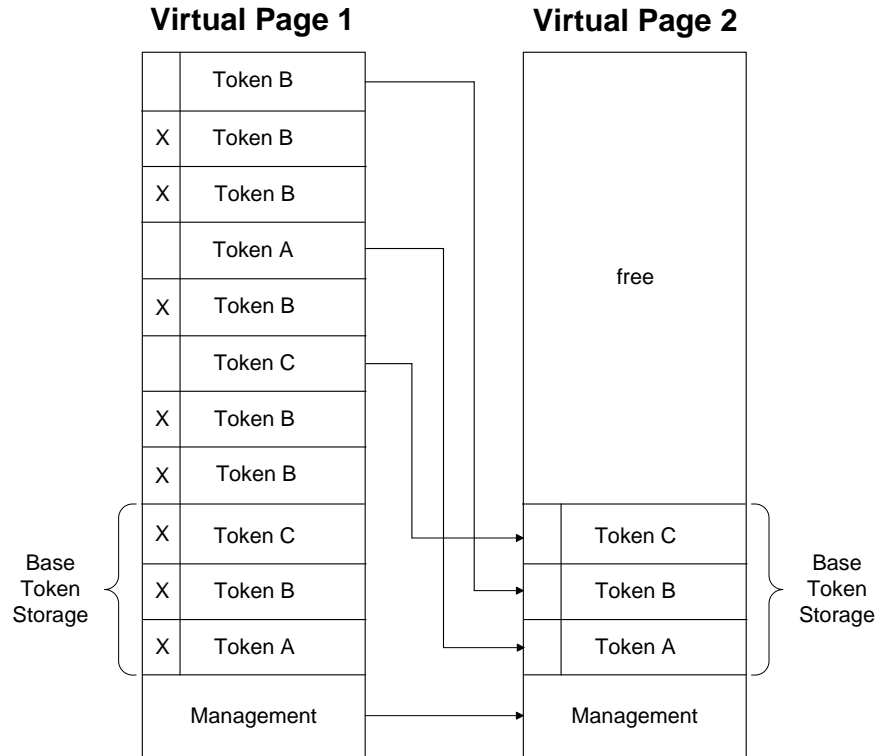


Figure 2. Transitioning between pages



## Write Cycle Constraints

Because flash cells are qualified for a guaranteed 1,000 (EM250) or 20,000 (EM35x) 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

You can use the following equation to estimate the write cycles of an individual token:

$$W = \left( \left( \frac{P - (4 \times T) - B}{X + 2} \right) + 1 \right) \times (C \times 2)$$

W	The approximate write-cycle count for the token in question
P	The virtual page size in bytes. 4096 for 8kB and 2048 for 4kB SimEE.
T	The total number of tokens in the Simulated EEPROM <sup>1</sup>
B	The bytes size of base token storage <sup>2</sup>
X	The byte size of the token in question
C	Flash cell write cycles. 1,000 for EM250 and 20,000 for EM35x.

<sup>1</sup> The default number of tokens is 12.

<sup>2</sup> The default set of tokens yield a base token storage size of 924 bytes. Base token storage refers to minimum storage size needed to hold a single copy of all token data (this includes the management tags).

**Note:** This equation is only suitable for a single token that is set over the entire life of the flash. For more complex estimates that involve writing to multiple tokens, you must use ratios.

**Note:** You can also estimate write cycles through the WRITECYCLES command, which is available in the EM250 Token Utility application.

**Note:** The EM35x does not yet offer a comparable Token Utility application.

Most stack tokens are only written on network join or leave operations. Because the number of join or leave operations is typically very low, the setting of stack tokens comprises a small percentage of set token operations. The only set token performed by the stack periodically comes from the nonce, which is set once for every 4,096 encrypted messages sent. The nonce is a non-indexed, 4-byte token.

### Examples

**Note:** The following examples were calculated for an EM250 8kB Simulated EEPROM. For an EM35x 8kB Simulated EEPROM, the examples below can be approximated by multiplying the resultant write cycles by 20. For an EM35x 4kB Simulated EEPROM, the examples should be recalculated since the relationship cannot be linearly approximated.

Consider a single application token, called AppTok, that is declared as non-indexed and 8 bytes large. For the sake of simplicity, this discussion only considers set token operations performed on the nonce and the AppTok (which would mean a stable network with no joining or leaving and periodic messages). Because these examples use the default token set plus the AppTok, the T variable in the above equation is 13 and the B variable is 934:

- The T variable is 13 because there are a default set of 12 tokens plus the AppTok.
- The B variable is 934 because the default set of tokens yield a base token storage size of 924 bytes plus 10 bytes for the AppTok.

The AppTok adds 10 bytes because scalar tokens require 2 bytes of management information and 8 bytes for the token's size. An indexed token requires 2 bytes for management information for each element as well as room for the data in each

element. To calculate the total base token storage needed for an indexed token, add 2 to the byte size of an element, and multiply that number by the number of elements. For example, a token with 7 elements and an element size of 12 bytes requires adding 98 bytes to the base token storage.

#### Only AppTok writes

To estimate the number of write cycles available when just the AppTok is being written, begin with the equation and the numbers provided above along with the word size of the token in question, which is 4. The equation is now:

$$997600 = \left( \left( \frac{4096 - (4 \times 13) - 934}{8 + 2} \right) + 1 \right) \times (1000 \times 2)$$

The AppTok can be written 997,600+ times.

#### Only nonce writes

To estimate the number of write cycles available when just the nonce is being written (the nonce is written every 4,096 secure messages that are sent), begin with the equation and the numbers provided above along with the bytes size of the token in question, which is 4. Because this is a counter token though, 50 bytes must be added to the byte size. The equation is now:

$$179786 = \left( \left( \frac{4096 - (4 \times 13) - 934}{54 + 2} \right) + 1 \right) \times (1000 \times 2)$$

The nonce token can be written 179,786+ times; Or incremented 8,989,285+ times.

#### Multiple token writes

If both the AppTok and nonce are being written, we must employ ratios to estimate the write cycles of the Simulated EEPROM. The ratios are simply a percentage of the individual write cycles calculated above. If the AppTok is written 90% of the time, the estimated write cycles for the AppTok alone are only 897,840:

$$0.9 \times 997600 = 897840$$

If the nonce is written the other 10% of the time, the estimated write cycles for the nonce alone are only 17,978 or 898,930 increments:

$$0.1 \times 179786 = 17978$$

The total set token calls (write cycles) for the entire Simulated EEPROM is the sum of these values: 915,818.

Reversing this example and having 10% AppTok writes and 90% nonce writes requires only changing the percentage in the equations above. The three resulting values are:

- AppTok: 99,760
- Nonce: 161,807
- Total: 261,567

As mentioned before, the calculations performed above are available by using the `WRITECYCLES` command in the EM250 Token Utility. The ratio calculations can be expanded to include as many or all of the tokens in the system.

### Calculation notes

Due to the inherent complexity of token usage patterns and the Simulated EEPROM, these simplified sample equations and calculations can only be used to provide ballpark estimates of worst-case scenarios. The equations assume the flash has precisely 1,000 write cycles. By using this number, a ballpark estimate can be provided for token write cycles based on flash write cycles.

On average, implementing the previous examples would result in an order of magnitude or greater increase in write cycles because 1,000 is the guaranteed minimum across voltage and temperature. Operating at standard voltage and temperature (3.3V and 25°C) will typically result in an increased number of flash write cycles and the estimations above will be increased. Therefore, the equations and calculations given can be considered estimates on the minimum number of write cycles.

### Token Definition Tips

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

#### 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 of avoiding extraneous memory by defining a structured token type.

```
typedef struct {  
    int8u dataA;  
    int16u dataB;  
    int8u 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 {
    int16u dataB;
    int8u dataA;
    int8u dataC;
} tokTypeData
```

### When to define a counter token

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 16-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.

### 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 int8u 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 int8u 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 just 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 of 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.

### 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, it consumes two words for management data, because the Simulated EEPROM must always know the token exists, whether empty or not.

## 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 at `hal/micro/sim-eprom.h`.

### `halSimEepromCallback`

```
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 EM250 and EM35x 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. The callback always reports one of the following four possible `EmberStatus` codes:

- `EMBER_SIM_EEPROM_ERASE_PAGE_GREEN`
- `EMBER_SIM_EEPROM_ERASE_PAGE_RED`
- `EMBER_SIM_EEPROM_FULL`
- `EMBER_ERR_FLASH_VERIFY_FAILED`
- `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.

#### **`EMBER_SIM_EEPROM_ERASE_PAGE_GREEN`**

#### **`EMBER_SIM_EEPROM_ERASE_PAGE_RED`**

When a token is set, the callback is triggered with either the red or green code. The Simulated EEPROM uses this code to determine whether the page needs to be erased. The green and red status codes work together with the constant `ERASE_CRITICAL_THRESHOLD` (defined in `hal/micro/xap2b/sim-eeprom.h` or `hal/micro/cortexm3/sim-eeprom.h`). This constant defines the threshold boundary for remaining space in the Simulated EEPROM, and determines whether the `EmberStatus` code is set to green or red:

- `EMBER_SIM_EEPROM_ERASE_PAGE_GREEN` indicates the Simulated EEPROM has enough space available ( $>ERASE\_CRITICAL\_THRESHOLD$ ) for the wear-leveling algorithm. The application can safely defer the lengthy page erase operation until it is convenient.
- `EMBER_SIM_EEPROM_ERASE_PAGE_RED` indicates that the Simulated EEPROM is critically low on available room ( $<ERASE\_CRITICAL\_THRESHOLD$ ) for the wear-leveling algorithm to continue, and data loss due to a full Simulated EEPROM is possible.

By default, `ERASE_CRITICAL_THRESHOLD` is set to one quarter of available space.

The application has control of erasing pages because an erase operation causes the EM250 and EM35x to ignore interrupts for 21 milliseconds while the flash is busy. See “Erase Timing and Callback Implementation” for a complete discussion of this implementation.

### 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 its data is dropped.

If this status is passed to the callback, the application should immediately call `halSimEepromErasePage()` four times on an EM250, which completely erases all of the physical pages that comprise the virtual page, and enables the Simulated EEPROM to continue operating. On an 8kB EM35x, the function should be called twice and on a 4kB EM35x, the function should be called once.

**Note:** An application that erases pages in a timely fashion 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 EM250 function call `halInternalSysReset(CE_REBOOT_F_INHIBIT)` or the EM35x 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, and every address inside the Simulated EEPROM fails 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.

While the callback can still first call `halInternalSimEeRepair(FALSE)` and then the EM250 function `halInternalSysReset(CE_REBOOT_F_VERIFY)` or the EM35x 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 Ember stack) and default into a safe mode.

## halSimEepromErasePage

```
halSimEepromErasePage( void )
```

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 also be called from anywhere at any time. 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.

**Note:** Erasing a page of flash takes 21 milliseconds, during which the EM250 and EM35x ignore 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.

## halSimEepromStatus

```
halSimEepromStatus  
( int16u * freeWordsInCurrPage, int16u * 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 (see “What Is Simulated EEPROM”): while one virtual page fills with set token calls, the other virtual page is erased by `halSimEepromErasePage()`. 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 1,000 or 20,000 write cycles, so under ideal conditions this variable is read at least 2,000 or 40,000 times before the flash cells fail, as two virtual pages are used.

## Design Constraints

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

- Non-indexed versus indexed tokens
- Counter tokens
- Numerical parameters

### Non-indexed versus indexed tokens

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**, also called array 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.

### Counter tokens

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.

### 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: 2,000 bytes
- Average read time of one 26 byte token/element: 240 microseconds
- Average write time of one 26 byte token/element: 1.2 milliseconds
- Worst-case write time of one token/element: 54 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: 15 milliseconds
- Worst-case Simulated EEPROM initialization time with repairing: 150 milliseconds
- Write cycles: See “Write Cycle Constraints.”

#### About Simulated EEPROM repairs

The worst-case initialization time with repairing (150ms) is an order of magnitude larger than the worst case initialization time without repairing (15ms) 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 if 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.

## Erase Timing and Callback Implementation

This section provides a more in-depth look at implementing the callback function `halSimEepromCallback()` as it pertains to the time needed to erase flash pages. This callback 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 the EM250 and EM35x completely unresponsive for 21 milliseconds while the flash is busy.

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 has become critically low on available 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, Ember 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.

**After Reading This Document**

If you have questions or require assistance with the procedures described in this document, contact Ember Customer support at <http://www.portal.ember.com>.

Copyright © 2006-2009 Ember Corporation. All rights reserved.

The information in this document is subject to change without notice. The statements, configurations, technical data, and recommendations in this document are believed to be accurate and reliable but are presented without express or implied warranty. Users must take full responsibility for their applications of any products specified in this document. The information in this document is the property of Ember Corporation.

Title, ownership, and all rights in copyrights, patents, trademarks, trade secrets, and other intellectual property rights in the Ember Proprietary Products and any copy, portion, or modification thereof, shall not transfer to Purchaser or its customers and shall remain in Ember and its licensors.

No source code rights are granted to Purchaser or its customers with respect to all Ember Application Software. Purchaser agrees not to copy, modify, alter, translate, decompile, disassemble, or reverse engineer the Ember Hardware (including without limitation any embedded software) or attempt to disable any security devices or codes incorporated in the Ember Hardware. Purchaser shall not alter, remove, or obscure any printed or displayed legal notices contained on or in the Ember Hardware.

Ember, Ember Enabled, EmberZNet, InSight, and the Ember logo are trademarks of Ember Corporation.

All other trademarks are the property of their respective holders.

