UG103.7: Non-Volatile Data Storage Fundamentals

This document provides a general introduction to non-volatile data storage using flash, with a focus on the three different dynamic data storage implementations offered for Silicon Labs microcontrollers and radio SoCs (Systems on Chip). It offers a comparison of the three implementations and provides recommendations on when to use each. Additional detail on using the various data storage implementations may be found in the following documents:

- *AN1154: Using Tokens for Non-Volatile Data Storage*
- *AN703: Using Simulated EEPROM Version 1 and Version 2 for the EM35x and EFR32 SoC Platforms*
- *AN1135: Using Third Generation Non-Volatile Memory (NVM3) Data Storage in Dynamic Multiprotocol Applications*

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

KEY POINTS

- Review of the challenges and design options for implementing non-volatile data storage.
- Review of the three dynamic data storage implementations.
- Introduction to tokens and the token API.
- Comparison of the three dynamic data storage implementations.

1. Introduction

Non-volatile memory (NVM) is memory that persists even when the device is power-cycled. On Silicon Labs microcontrollers and radio SoCs, the NVM is implemented as flash memory. In many applications the flash is not only used to store the application code but also to store data objects that are written and read frequently by the application. As flash memory can only be erased a limited number of times, several methods exist to efficiently read and write non-volatile data without wearing out the flash.

Some data is considered manufacturing data that is written only once at manufacturing time. This document is concerned with dynamic data that changes frequently over the life of the product.

This document provides an introduction to the main design options for dynamic data storage in microcontrollers and radio SoCs, along with guidelines on what factors affect flash lifetime. In addition it introduces the main flash data storage implementations offered by Silicon Labs:

- NVM3
- Simulated EEPROM version 1 (SimEEv1) and version 2 (SimEEv2)
- Persistent Store (PS Store)

2. Implementations of Non-Volatile Data Storage

This chapter introduces some of the challenges and design options when implementing non-volatile data storage in flash memory. It describes at a high level how non-volatile data storage is implemented in flash memory in PS Store, SimEEv1/v2, and NVM3.

2.1 Basic Implementations

One of the characteristics of flash memory is that it is writable in smaller pieces, usually 32-bit words, while it can only be erased in larger chunks, usually pages of several kilobytes. When using flash as data storage, the most straightforward implementation option would be to store each data object in its own flash page, as shown in the following figure. This way each object can be erased and re-written without influencing the other data objects. Usually the data objects are much smaller than the page size, and this solution is not an effective way of using the available flash space.

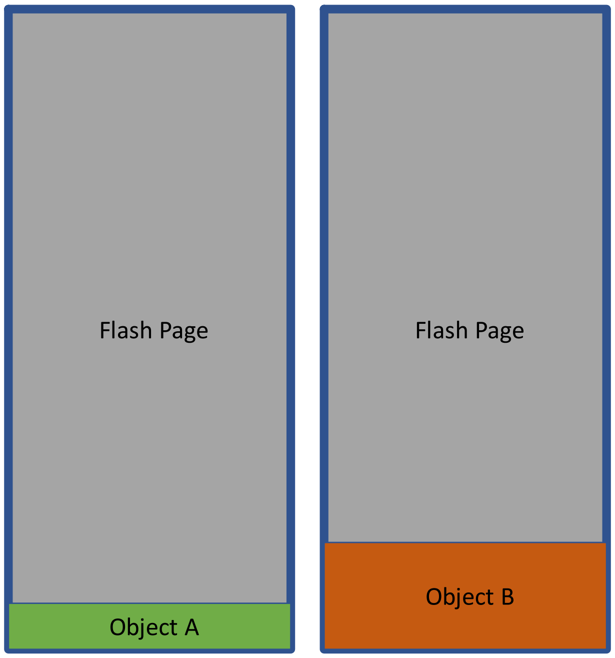


Figure 2.1. One Object Per Page

To avoid wasting flash space we can store several data objects in one flash page, as shown in the following figure. This solution then introduces a challenge when we want to write a new value to one of the data objects. In that case the page must be erased before all objects are written back to the page again, including the object we changed. As flash memory can only endure a limited amount of flash erases before the flash cells are worn out, this solution results in a very limited device lifetime.

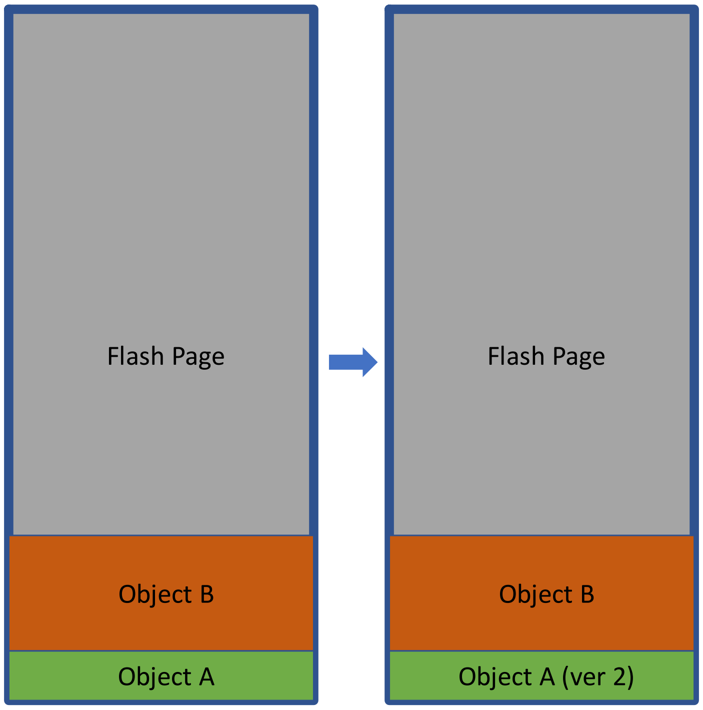


Figure 2.2. Multiple Objects in One Flash Page

To avoid erasing the flash page for every object write, we can instead write new versions of each object to new empty locations in the flash page. This is a simple form of wear-levelling that reduces the number of page erases. However, this requires that we store some identification information along with the object data that tells us what object the data belongs to, so we know how to find the latest version of the data object. This is illustrated in the following figure, where a key is added to each version of the object data to identify what object the data belongs to. When accessing an object we then need to search through the flash page for the most recent version of the object. In this case the newest version is the one with the highest address, as we start writing from the lowest address of the page.

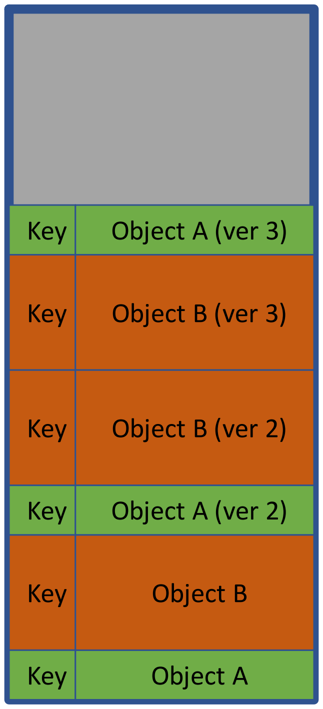


Figure 2.3. Object Versions With Keys

2.2 Handling Resets and Power Failures

As we fill up the flash page with new versions of the data objects, eventually no room is left to write new object data. At this point we need to erase the page and start over by writing only the latest versions of each object to the flash page. In many applications, however, power failures or resets can happen at any time, and we should not risk losing data if this occurs. If a reset occurs after the flash page is being erased, but before the data objects are written back, then we will lose this data. To handle this case we introduce a second page, to which we copy the latest version of the data objects, before erasing the original page, as shown in the following figure Then we can start filling the second page with data. When the second page is full we move the latest data back to the first page and so on. This mechanism, where the storage is alternated between two flash pages, is how PS Store operates.

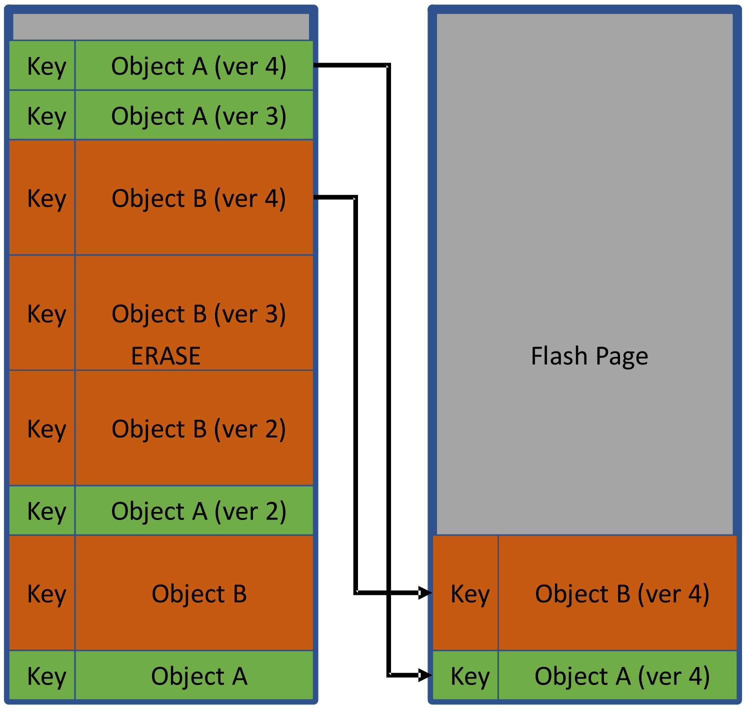


Figure 2.4. Latest Data Copied to New Page Before Erase

2.3 Introducing Virtual Pages

In some applications we write to data objects frequently, and the flash pages therefore also need to be erased frequently. As the data objects in the implementation so far are only spread across two flash pages, each page will frequently get erased and the flash lifetime will be limited. To increase the lifetime we can use more flash pages to store the data objects. In this example, instead of two physical pages, we operate with two virtual pages (A and B) that each consist of several physical flash pages. The virtual pages are erased and written to as if they were one larger flash page. The difference is simply that each virtual page is bigger and we can write more data before we need to erase the virtual page, hence the lifetime is extended. In addition to increasing flash lifetime, using several flash pages per virtual page allows you to store more or larger objects. SimEEv1 uses this design, with each virtual page consisting of two flash pages, A and B, as shown in the following figure.

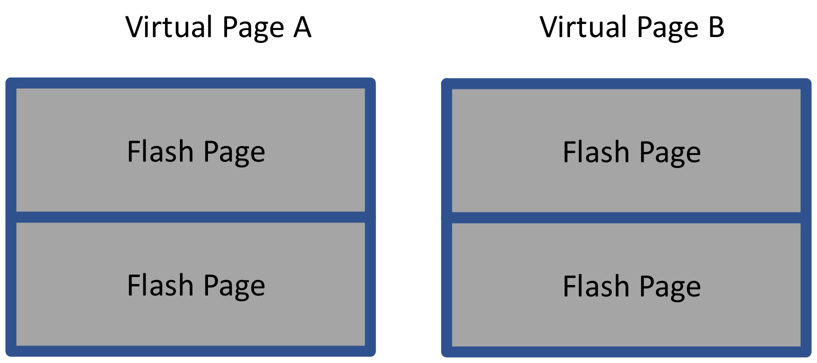


Figure 2.5. Virtual Pages

In some applications the time it takes to write a non-volatile data object must be minimized so as to not interfere with the timing of other critical operations. If an object write is triggered when a virtual page is full, the latest version of all objects must first be copied to the new virtual page before writing the new object version in question to the new page. All objects must be copied over immediately to allow the first page to be quickly erased so we can move data there in case of a failure. Copying all objects at once increases the worst-case object write time.

To reduce the write times, a third virtual page can be introduced that is always kept erased. Instead of copying over the latest version of every object when the first page is full, we can instead copy over only some of the objects. The rest of the objects are copied over as a part of subsequent write operations. This way we spread the copy process to the new page over more write operations, hence each write operation takes less time to complete. With this approach we have live object data spread out across two virtual pages and the third page is always kept erased, so we have somewhere to move the data to in case of a failure. SimEEv2 uses this implementation with 3 virtual pages where each virtual page consists of 6 flash pages.

2.4 Basic Storage

Basic storage is defined as the size of all the latest version of all objects, including any overhead stored with them. Every time a flash page or virtual page is erased we must first move the basic storage over to a new page. The basic storage size is important, as it determines how much flash space is left over in a page to store any new versions of the object data. If the basic storage takes up almost the entire page, we can only write a few new object versions before we need to move to a new page and erase the old one. This leads to frequent page erases and a short flash lifetime.

2.5 FIFO Model

The flash data storage implementations can be modelled as a First-In First-Out (FIFO) buffer (see the following figure), where we write new versions to the input of the FIFO. As the FIFO fills up, we need to free up space by erasing one or more pages at the end of the FIFO. Before we erase a page, we need to copy any object versions for which no newer version of the same object exists in the rest of the flash pages. Other object data can be discarded, because newer versions exist. To maximize flash lifetime we want to copy as few object versions as possible, so that most of the writes to the flash memory are new versions of the objects. If the FIFO is implemented over a large flash space, it is more likely that the new version of the object has been written and that object versions at the end of the FIFO can be discarded. In this case the flash page can be erased with few or no object versions copied.

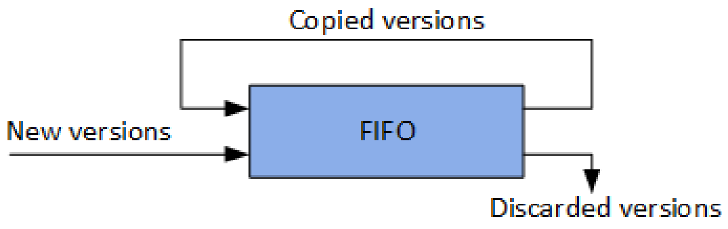


Figure 2.6. FIFO Buffer

A drawback of using a few virtual pages is that the available memory for the FIFO is limited to only the virtual pages that can hold live data. For implementations using two virtual pages, like SimEEv1, this means only half the storage space is used for the FIFO, while for SimEEv2 two thirds of the storage space is used for the FIFO.

To allow a higher portion of the storage space to be used for the FIFO, we can instead implement the FIFO as a circular buffer over the entire flash storage space allocated, as shown in the following figure. In this implementation we always need to keep enough erased pages in front of the leading edge of the buffer to write the largest-sized object in case of a failure. When the FIFO fills up to reach the critical number of erased flash pages, we copy over any object versions that have not been superseded and erase the page at the back of the FIFO. This means that, instead of keeping a full virtual page erased, we only need to keep enough space erased to fit the largest-sized object. We can use the rest of the space for the FIFO. NVM3 is implemented as a circular buffer implemented over the entire storage space, thus increasing flash lifetime compared to implementations using smaller virtual pages.

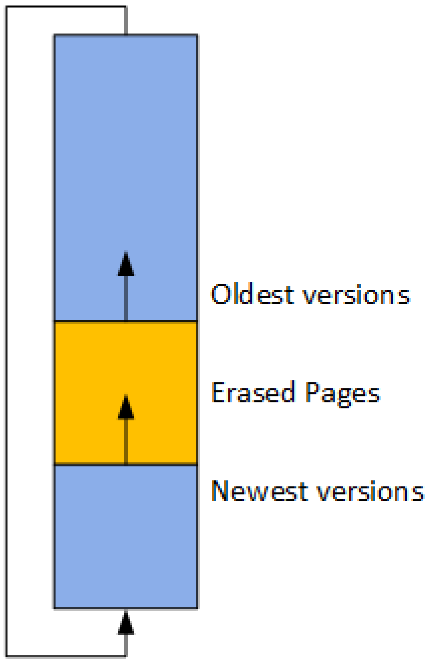


Figure 2.7. Circular Buffer

2.6 Counter Objects

For some types of data, the storage format can be optimized for the flash medium. For example, counter values are usually incremented by 1 or some other low value every time they are written. Normally this means that we would have to write the entire counter value in addition to identification bytes every time the counter is incremented. We can optimize this by only storing a start value in addition to the identification value the first time a counter is written. Then we reserve a number of the words following the initial value for writing increments. As the flash words in Silicon Labs devices can be written twice for every erase, we can use one halfword for each increment. One increment is then written by writing the increment value in one halfword. To find the current counter value, we start with the initial value and add the increment values in the halfwords following the initial value. This means that we only need to write one halfword for each increment, instead of the whole counter value and identification value. NVM3 and SimEEv1/v2 support counter objects.

2.7 Indexed Objects

When storing data such as arrays in flash, we often only update one index at a time. If we store the whole array as one regular object, we must write the whole array to flash even if just one index is updated. Instead of storing the entire array in one object, we can divide each data array over multiple objects and only update the objects holding the changed array indexes. While it is possible to split arrays into multiple objects manually for all the Silicon Labs storage implementations, SimEEv1/v2 allow all of the indexes to share one object key. The index entry into the array to be looked up is then provided as a separate parameter.

3. Dynamic Data Storage Implementations

This chapter introduces some of the dynamic data storage implementations offered by Silicon Labs, including SimEEv1 and SimEEv2, PS Store, and NVM3.

3.1 SimEEv1/v2

SimEEv1/v2 are used with the EmberZNet PRO, Silicon Labs Thread, and Flex SDKs, as well as with some multiprotocol applications. SimEEv1 uses two virtual pages with a fixed total size of 8 kB, while SimEEv2 uses three virtual pages with a fixed total size of 36 kB.

One characteristic of SimEEv1/v2 is that all objects are defined with size and type at compile time, hence a new object cannot be created or deleted at runtime.

Silicon Labs provides a plugin for upgrading SimEEv1 data to SimEEv2.

Information about the SimEEv1/v2 implementations is found in *AN703: Using Simulated EEPROM Version 1 and Version 2 for the EM35x and EFR32 SoC Platforms*.

3.2 PS Store

PS Store is used with Bluetooth devices. PS Store API commands are used to manage user data in PS keys in the flash memory of the Bluetooth device. User data stored within the flash memory is persistent across reset and power cycling of the device. The persistent store size is 2048 bytes and uses two flash pages for storage. As Bluetooth bondings are also stored in this area, the space available for user data also depends on the number of bondings the device has at the time. The size of a Bluetooth bonding is around 150 bytes. With its simple implementation and few storage flash pages, PS Store is the smallest of Silicon Labs' non-volatile storage options. PS Store allows objects to be created and deleted at runtime.

Information about the PS Store APIs may be found in the *Bluetooth API Software Reference Manual*.

3.3 NVM3

The third generation Non-Volatile Memory (NVM3) data storage driver is an alternative to SimEEv1/v2 and PS Store. The NVM3 driver provides a means to write and read data objects (key/value pairs) stored in flash. Wear-leveling is applied to reduce erase and write cycles and maximize flash lifetime. The driver is resilient to power loss and reset events, ensuring that objects retrieved from the driver are always in a valid state. Because NVM3 can be used with both Bluetooth and EmberZNet PRO, it allows a single data storage instance to be shared in Dynamic Multiprotocol (DMP) applications.

Some of the main features of NVM3 are as follows:

- Key/value pair data storage in flash
- Runtime creation and deletion of objects
- Persistence across power loss and reset events
- Wear-leveling to maximize flash lifetime
- Object sizes from 0 to 1900 bytes
- Configurable flash storage size (minimum 3 flash pages)
- Cache with configurable size for fast object access
- Data and counter object types
- Compatibility layers with token and PS Store APIs provided
- Single shared storage instance in multiprotocol applications
- Repack API to allow application to run clean-up page erases during periods with low CPU load

Detailed information on NVM3 is documented in the EMDRV->NVM3 section of the Gecko HAL & Driver API Reference Guide. Users who are accessing NVM3 through its native API should refer to this API reference guide for information. Users who are developing dynamic multiprotocol applications should refer to *AN1135: Using Third Generation Non-Volatile Memory (NVM3) Data Storage in Dynamic Multiprotocol Applications*.

4. Tokens and the Token API

A token is an abstract data constant that has special persistent meaning for an application, and is used to store data objects as discussed earlier in section 2. Implementations of Non-Volatile Data Storage. A token has two parts: a token key and token data. The token key is a unique identifier that is used to store and retrieve the token data. In many cases, the word "token" is used quite loosely to mean the token key, the token data, or the combination of key and data. In this document token always refers to the key + data pair.

The fundamental purpose of the token system is to allow the token data to persist across reboots and during power loss. By using the token key to identify the proper data, the application requesting the token data does not need to know the storage location of the data. This simplifies application design and code reuse.

There are two types of dynamic tokens:

- Non-indexed (or basic) tokens
- Indexed tokens

Dynamic okens can also be categorized in the following groups based on their software purpose:

- Stack tokens: These tokens are read/write and defined in every application to support stack behavior. These tokens live in either NVM3 or SimEEv1/v2.
- Application tokens: These tokens are read/write and defined by the application to support application behavior. It is left open to a customer to decide if there are application tokens, how they are defined, and what they do. These tokens live in NVM3 or SimEEv1/v2.

The token API is a set of wrapper functions and macros used to define and access non-volatile data objects in EmberZNet PRO, Silicon Labs Thread, and Flex SDKs, as well as with some multiprotocol applications.

Additional information on token types, groups, as well as instructions on how to define and access tokens can be found in *AN1154: Using Tokens for Non-Volatile Data Storage*. Users should read this document before using the token API. Information on the token APIs is provided in the HAL API section of the stack API reference.

5. Comparing Non-Volatile Data Storage Implementations

The following table presents an overview of the main features of the various Silicon Labs Non-Volatile Data Storage implementations.

Table 5.1. NV Data Storage Implementation Comparison

Feature	NVM3	SimEEv1	SimEEv2	PS Store
Compatible devices	EFM32, EFR32	EM2xx, EM3xx, EFR32	EM3xx, EFR32	EFR32
Compatible radio protocols	Dynamic Multiprotocol with EmberZNet and Bluetooth	EmberZNet, Thread, Flex	EmberZNet, Thread, Flex	Bluetooth
Flash used for storage	3 or more flash pages	8 KB	36 KB	4 KB
Virtual pages	NA	2	3	NA
Max basic storage (Sum of all object data with overhead)	Variable (see *AN1135* for details)	2 KB	8 KB	2 KB
Max number of objects	Limited by basic storage	254	254	Limited by basic storage
Max object size (bytes)	Configurable 208-1900	254	254	255
Object creation/deletion	Runtime	Compile time	Compile time	Runtime
Compiled flash size excluding data storage	7.7 KB	3.5 KB	5.4 KB	1.6 KB
Counter object support	Yes	Yes	Yes	No
Indexed object support	Partial[1]	Yes	Yes	No
Overhead per object (bytes)	4 (size <= 128 bytes) 8 (size > 128 bytes)	2	6	10
Counter object size including overhead (bytes)	212	60	56	NA
Counter increments before rewrite	100	25	25	NA

[1] When using NVM3, the token functions implement indexed tokens by storing each index in a separate NVM3 object.

5.1 Flash Lifetime

All Silicon Labs Flash Data Storage implementations use some form of wear-levelling to prolong flash lifetime. The effectiveness of the wear-levelling depends on the implementation, the type of data stored, and how often it is updated. The main factors that affect wear-levelling and thereby flash lifetime are:

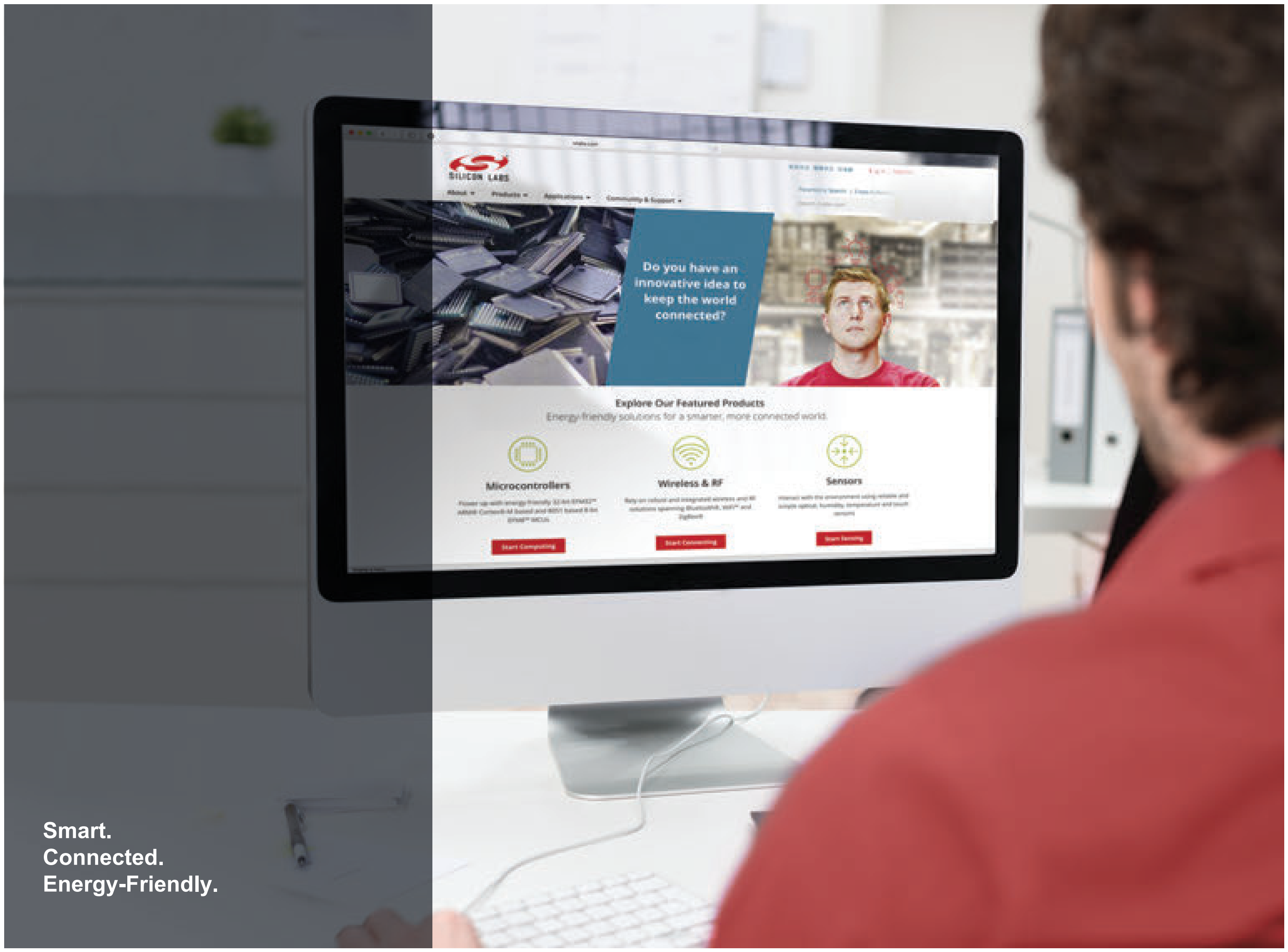
- Size of flash used for data storage: More flash area gives longer flash lifetime. For NVM3, the number of flash pages used for data storage can be configured, while the rest of the implementations use fixed storage sizes.
- Stored overhead per object: When writing data to the object storage, some overhead bytes are added to identify the data. Implementation with less overhead means the data objects take up less space in flash, and gives longer flash lifetime.
- Alignment to minimum object size: Objects are stored in multiples of the smallest object size. If the data size does not align with this size, padding bytes are added, which adds to the stored data and reduces flash lifetime. For instance, when storing 16-bit tokens, NVM3 and PS Store add two extra bytes of padding in addition to the overhead bytes. SimEEv1/v2 are able to store 16-bit data objects without padding.
- Remaining storage after basic storage: For implementations using virtual pages, when switching to a new virtual page one instance of each object is written to the page. The rest of the virtual page can then be used to store new writes of the objects. If a lot of space is used to store one instance of each object, little space is left in the virtual page to use for wear-levelling the subsequent object writes. The flash lifetime will therefore be reduced when the total amount of object data is large relative to the virtual page size. Even for NVM3, where virtual pages are not used, the flash lifetime is limited by the available space of the total NVM3 storage.

To help monitor the actual flash wear, NVM3 and SimEEv1/v2 include function calls for reporting the number of page erases of the data storage flash pages. These erase counters can be read during accelerated lifetime testing of a product to verify if the flash wears at an acceptable rate.

5.1.1 Flash Lifetime Estimator

A tool that can be used to estimate the flash lifetime of an application is available from Silicon Labs on request as a Microsoft Excel spreadsheet (ug103-07s-token-lifetime-estimator-efr32-series1.xls). To create an estimation, a set of objects/tokens must be entered into the spreadsheet along with an estimation on how often these objects will be written in the application. The spreadsheet calculates the estimated flash lifetime of the device when using NVM3, PS Store, and SimEEv1/v2. More information on how to use the use the estimator is found in the README sheet of the spreadsheet.

Note: The spreadsheet only presents an estimation using an ideal model of the storage implementation. The actual flash lifetime of an application may differ.



Products
www.silabs.com/products

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

SILICON LABS

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

http://www.silabs.com