AN1135: Using Third Generation Non-Volatile Memory (NVM3) Data Storage

The NVM3 driver provides a means to write and read data objects (key/value pairs) stored in flash. Wear-leveling is applied to reduce erase and write cycles and maximize flash lifetime. The driver is resilient to power loss and reset events, ensuring that objects retrieved from the driver are always in a valid state. A single NVM3 instance can be shared among several wireless stacks and application code, making it well-suited for multiprotocol applications. This application note explains how NVM3 can be used as non-volatile data storage in Zigbee applications, including Dynamic Multiprotocol applications with Zigbee and Bluetooth.

KEY POINTS

- Key/value pair data object storage in flash
- Wear-leveling to maximize flash lifetime
- Resilient to power and reset events
- Shared by Zigbee and Bluetooth stacks
- Compatible with PS Store and Token APIs through wrappers
- Data upgradable from Simulated EEPROM version 2 to NVM3

1. Introduction

The third generation Non-Volatile Memory (NVM3) data storage driver is an alternative to Simulated EEPROM (SimEE), used with EmberZNet (the Silicon Labs Zigbee stack), and Persistent Store (PS Store), used with the Silicon Labs Bluetooth stack. Because NVM3 can be used with both of these stacks, it allows a single data storage instance to be shared in Dynamic Multiprotocol (DMP) applications.

NVM3 is designed to work in EmberZNet, Silicon Labs Thread, Flex, and Bluetooth applications running on EFR32 as well as MCU applications running on EFM32.

Some of the main features of NVM3 are:

- Key/value pair data storage in flash
- Runtime object creation and deletion
- Persistence across power loss and reset events
- Wear-leveling to maximize flash lifetime
- Object sizes configurable up to 4096
- 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 are documented in the EMDRV->NVM3 section of the Gecko HAL & Driver API Reference Guide. Users who are developing EFM32 MCU applications or accessing NVM3 through its native API should refer to this API reference guide for information. Users who are developing EmberZNet applications or DMP applications with EmberZNet and Bluetooth should use this application note to understand how to use NVM3 in these use cases.

2. Using NVM3

This chapter provides information on how to use NVM3 with Zigbee applications or DMP applications with Zigbee and Bluetooth. First information is provided about NVM3, including

- NVM3 repacking
- Default NVM3 instance
- NVM3 Library plugin
- Simulated EEPROM version 2 (SimEEv2) to NVM3 Upgrade plugin
- NVM3 stack usage
- Max basic storage

2.1 NVM3 Repacking

As the flash fills up, it will reach a point where it can no longer store additional objects. A repacking operation is required to release out-of-date objects to free up flash. Because erasing pages takes a long time, the NVM3 driver does not trigger the process by itself unless free memory reaches a critical low level. This level is called the **Forced Repack Limit**, and when it is reached the NVM3 driver automatically runs a repack when the user starts a write operation. The Forced Repack Limit is calculated automatically based on the page size of the device and the initialization parameters for NVM3.

In some applications it can be beneficial to schedule repacks at times when the CPU is idle so as not to interfere with the timing of other tasks. In such cases the application code can trigger the repacking process by calling the `nvm3_repack()` function. This function will only trigger a repack if the free memory is below the **User Repack Limit**. If there is more free memory, the function returns immediately. The User Repack Limit can be configured relative to the Forced Repack Limit by setting how many bytes below the Forced Repack Limit the User Repack Limit should be placed in the NVM3 initialization struct. The default value is 0, which means the User Repack and Forced Repack Limits are the same. If the timing requirements of the application are tight, it could be desirable to place the User Repack Limit well below the Forced Repack Limit to ensure that all repacks are triggered by calls to `nvm3_repack()`and that forced repacks are avoided. In such cases the User Repack Limit should be placed far enough below the Forced Repack Limit to allow the worst-case number of object writes (including overhead) between `nvm3_repack()`calls without hitting the Forced Repack Limit.

During the `nvm3_repack()`call, the NVM3 either moves data to a new page or erases an obsolete page. At most, the call will block for a period equal to a page erasure time plus a small execution overhead. Page erasure time for the EFM32 and EFR32 parts can be found in their respective data sheets.

More information on repacking is found in the EMDRV->NVM3 section of Gecko HAL & Driver API Reference Guide.

2.2 Default NVM3 Instance

Several NVM3 instances can be created on a device and live independently of each other, but to save memory it is usually desirable to use only one NVM3 instance as each instance adds some overhead. For Zigbee or DMP applications built with the Gecko SDK, a common default instance is used. In the current version this NVM3 instance is set to use 36 kB of flash space for data storage. NVM3 also has a cache to speed up access to NVM3 objects. The default size of this cache is 100 elements, but it can be configured in the NVM3 Library plugin option in AppBuilder as shown in Figure 2.1 NVM3 Library Plugin in AppBuilder on page 5.

Note: The cache size must be set to a value greater than or equal to the number of NVM3 objects used. This includes the number of tokens, PS Store objects, and NVM3 objects created through the native NVM3 API. For indexed tokens, add one cache item for each index.

The `nvm3_countObjects()` function can be used to find the number of used NVM3 objects at any given point. Silicon Labs recommends checking this function after initialization of tokens, PS Store, and native NVM3 objects to figure out the correct size of the NVM3 default cache size.

2.2.1 NVM3 Default Instance Key Space

NVM3 uses a 20-bit key to identify each object. To avoid using the same key for more than one object, the NVM3 key space for the default NVM3 instance has been divided into several domains as outlined in the following table. For example, NVM3 objects defined in the EmberZNet stack should use NVM3 keys in the range 0x10000 to 0x1FFFF, while user application tokens should use keys below 0x10000.

Table 2.1. NVM3 Default Instance Key Space

Domain	NVM3 Key
User	0x00000 - 0x0FFFF
EmberZNet stack	0x10000 - 0x1FFFF
Silicon Labs Thread stack	0x20000 - 0x2FFFF
Connect (Flex) stack	0x30000 - 0x3FFFF
Bluetooth stack	0x40000 - 0x4FFFF
Z-Wave stack	0x50000 - 0x5FFFF
Reserved	0x60000 - 0xFFFFF

2.3 NVM3 Library Plugin

To use NVM3 with an EmberZNet or DMP example application, the **NVM3 Library** plugin should be included in the project. All PS Store and SimEE plugins should be deselected.

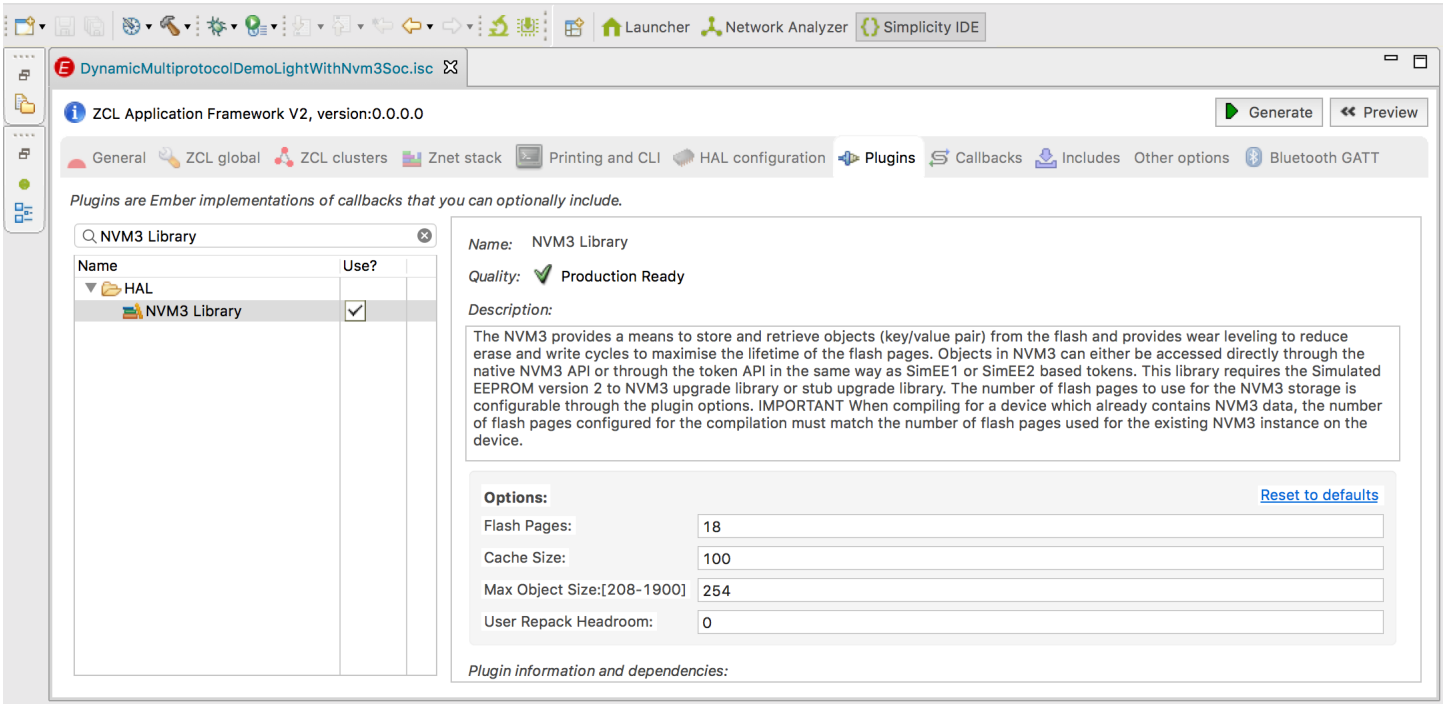


Figure 2.1. NVM3 Library Plugin in AppBuilder

The **NVM3 Library** plugin offers four plugin options:

- **Flash Pages**: Number of flash pages to use for NVM3 data storage. Must be 3 or higher.
- **Cache Size:** Number of objects to cache. To reduce access times, this number should be equal to or higher than the number of objects, including tokens, stored in NVM3 at any time.
- **Max Object Size:** Size of largest allowed NVM3 object in bytes. Must be between 208 and 1900 bytes.
- **User Repack Headroom:** Headroom determining how many bytes below the forced repack limit the user repack limit is placed. The default value is 0, which means that the forced and the user repack limits are the same.

  IMPORTANT: When creating an application with the current version of the NVM3 library plugin for a device that already contains an NVM3 instance in flash, the number of flash pages configured for the NVM3 instance must match the number of flash pages for the NVM3 instance already found on the device. Hence it is not possible to change the size of an NVM3 instance once it has been installed on a device, without first erasing the flash pages holding the NVM3 instance and NVM3 objects stored there.

2.4 SimEEv2 to NVM3 Upgrade Plugin

An AppBuilder plugin is provided that upgrades tokens stored in SimEEv2 to NVM3. For tokens to be successfully upgraded to NVM3, CREATOR_* and NVM3KEY_* defines must be added for all tokens as described in section 3.1 Token API. The upgrade plugin will replace the SimEEv2 storage in-place with an NVM3 storage instance. The plugin does this by compacting the SimEEv2 storage down to 12 kB, and then creates an NVM3 instance in the remaining 24 kB of the original 36 kB SimEEv2 storage space. After the token data has been copied over from SimEEv2 to NVM3, the SimEEv2 storage is erased and the NVM3 instance is resized to use the entire 36 kB storage space. Apart from the code space needed for the upgrade library code, the upgrade does not require any additional flash space to the 36 kB storage area. The upgrade plugin requires that the existing SimEEv2 storage space and new NVM3 storage space are located at the same address and have the same size.

The **Simulated EEPROM version 2 to NVM3 Upgrade Library** plugin should be included to enable the upgrade as shown in the figure below. If no SimEEv2 token data is found, the upgrade plugin will look for NVM3 data, and if neither is found it will create a new NVM3 instance with tokens set to their default values. For applications that do not need to upgrade any SimEEv2 tokens, the **Simulated EEPROM version 2 to NVM3 Upgrade Stub** plugin should be included instead.

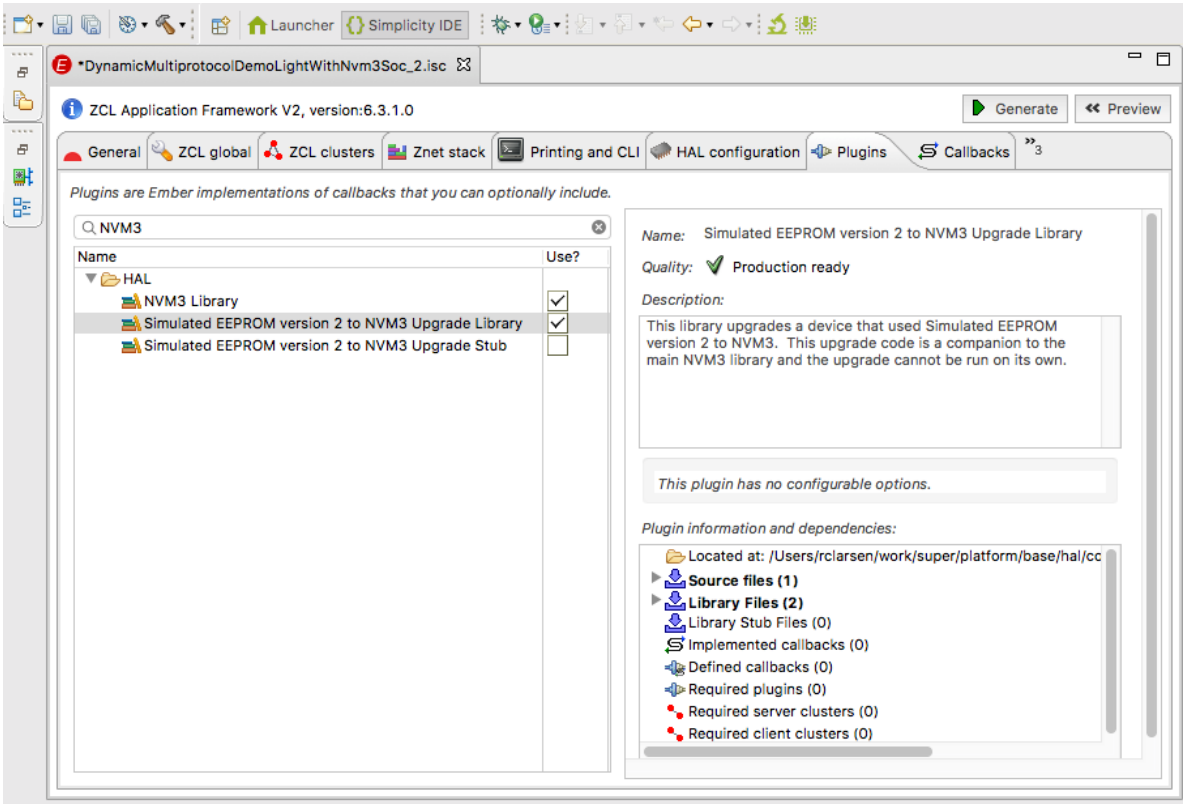


Figure 2.2. SimEEv2 to NVM3 Upgrade Library and Stub Plugins in AppBuilder

2.5 NVM3 Stack Usage

NVM3 uses up to 380 bytes of stack, and any operating system task stack that calls NVM3 functions should be large enough to account for the NVM3 stack usage.

2.6 Max Basic Storage

Basic storage is defined as the size of on instance of all objects including any overhead stored with the data. For NVM3 the maximum amount of data you can store is dependent on the number of flash pages used for storage and the max object size used for NVM3. The following table shows the maximum allowed basic storage for up to 18 2-kB flash pages and the minimum (208 bytes), default (254 bytes), and maximum (1900 bytes) max object size. Note that this is a theoretical limit and if the basic storage is at this limit, no space is left for wear-levelling and page erases will be forced for every object write. The NVM3 instance should therefore be configured with enough flash pages to put the maximum allowed basic storage significantly higher than the actual basic storage.

Table 2.2. Max Allowed Basic Storage with 2 kB page size

Flash pages	Total size (bytes)	Max allowed basic storage (bytes)		
		Max object size = 208 bytes	Max object size = 254 bytes	Max object size = 1900 bytes
3	6144	1596	1504	0
4	8192	3624	3532	240
5	10240	5652	5560	2268
6	12288	7680	7588	4296
7	14336	9708	9616	6324
8	16384	11736	11644	8352
9	18432	13764	13672	10380
10	20480	15792	15700	12408
11	22528	17820	17728	14436
12	24576	19848	19756	16464
13	26624	21876	21784	18492
14	28672	23904	23812	20520
15	30720	25932	25840	22548
16	32768	27960	27868	24576
17	34816	29988	29896	26604
18	36864	32016	31924	28632

3. NVM3 API Options

This chapter describes the three different APIs available to access NVM3 objects.

- Token API
- Persistent Store API
- Native NVM3 API

3.1 Token API

The token API is used to access data stored in SimEEv1 and SimEEv2 in the EmberZNet, Silicon Labs Thread and Flex SDKs, as well as multiprotocol applications. Information on how to define and access tokens can be found in *AN1154: Using Tokens for Non-Volatile Data Storage*, and users should read this document before using the token API. When selecting the NVM3 Library plugin instead of one the SimEE plugins, the NVM3 default instance is used to store the token data instead of SimEE. The same token API can still be used to access tokens stored in NVM3, but the token definition needs some modifications to work with NVM3, as described below.

When defining a token to be used with SimEE, a creator code must be defined as an identifier for the token. Similarly, when defining a token to be used with NVM3, an NVM3 key must be defined for the token. A token definition that is compatible with both NVM3 and SimEE would include both an NVM3 key and a creator code and look like this:

```
#define CREATOR_name 16bit_value
#define NVM3KEY_name 20bit_value
#ifdef DEFINETYPES
  typedef data_type type
#endif
#ifdef DEFINETOKENS
  DEFINE_*_TOKEN(name, type, ... ,defaults)
#endif
```

Select a 20-bit NVM3 key for the token, according to the domains in Table 2.1 NVM3 Default Instance Key Space on page 4. Each token must have a unique NVM3 key, except for indexed tokens, where more NVM3 keys must be reserved as outlined in section 3.1.2 Special Considerations for Indexed Tokens.

3.1.1 Deleting Tokens

As tokens are created at compile time, they cannot be created or deleted at run time. NVM3 objects are, however, created and deleted at run time, and the token initialization function creates NVM3 objects for each defined token if they do not already exist. The token initialization does not delete NVM3 objects found that do not have a corresponding token associated with them. Therefore, if a token is no longer included in an application, the application should manually delete the associated NVM3 object by using the NVM3 Native API described in section 3.3 Native NVM3 API.

3.1.2 Special Considerations for Indexed Tokens

NVM3 does not have native support for indexed tokens, and, therefore, an extra requirement is imposed on the NVM3 key selection for indexed tokens. With NVM3, indexed tokens are implemented by storing each index in a separate object, starting with index 0 stored at the defined NVM3KEY_name key value and the last index (127) stored with key NVM3KEY_name + 127. Because of this implementation, 128 NVM3 keys must be reserved for each indexed token. The user still only defines one NVM3KEY_name key value, but no other tokens should be defined with key values in the 127 values following this defined key. Even if the token is defined with fewer than 128 indices, all 128 indices should be reserved as the token might be expanded with more indices later on.

The example below shows two indexed tokens defined in the user key domain:

```
// This key is used for an indexed token and the subsequent 0x7F keys are also reserved
#define NVM3KEY_MY_INDEXED_TOKEN_A 0x00000
// This key is used for an indexed token and the subsequent 0x7F keys are also reserved
#define NVM3KEY_MY_INDEXED_TOKEN_B 0x00080
```

Table 3.1. Indexed Token NVM3 Key Selection Example

NVM3KEY	NVM3 Objects Contents
0x00000	Reserved for TOKEN_MY_INDEXED_TOKEN_A index 0
0x00001	Reserved for TOKEN_MY_INDEXED_TOKEN_A index 1
0x00002	Reserved for TOKEN_MY_INDEXED_TOKEN_A index 2
…	
0x0007F	Reserved for TOKEN_MY_INDEXED_TOKEN_A index 127
0x00080	Reserved for TOKEN_MY_INDEXED_TOKEN_B index 0
0x00081	Reserved for TOKEN_MY_INDEXED_TOKEN_B index 1
…	
0x000FF	Reserved for TOKEN_MY_INDEXED_TOKEN_B index 127

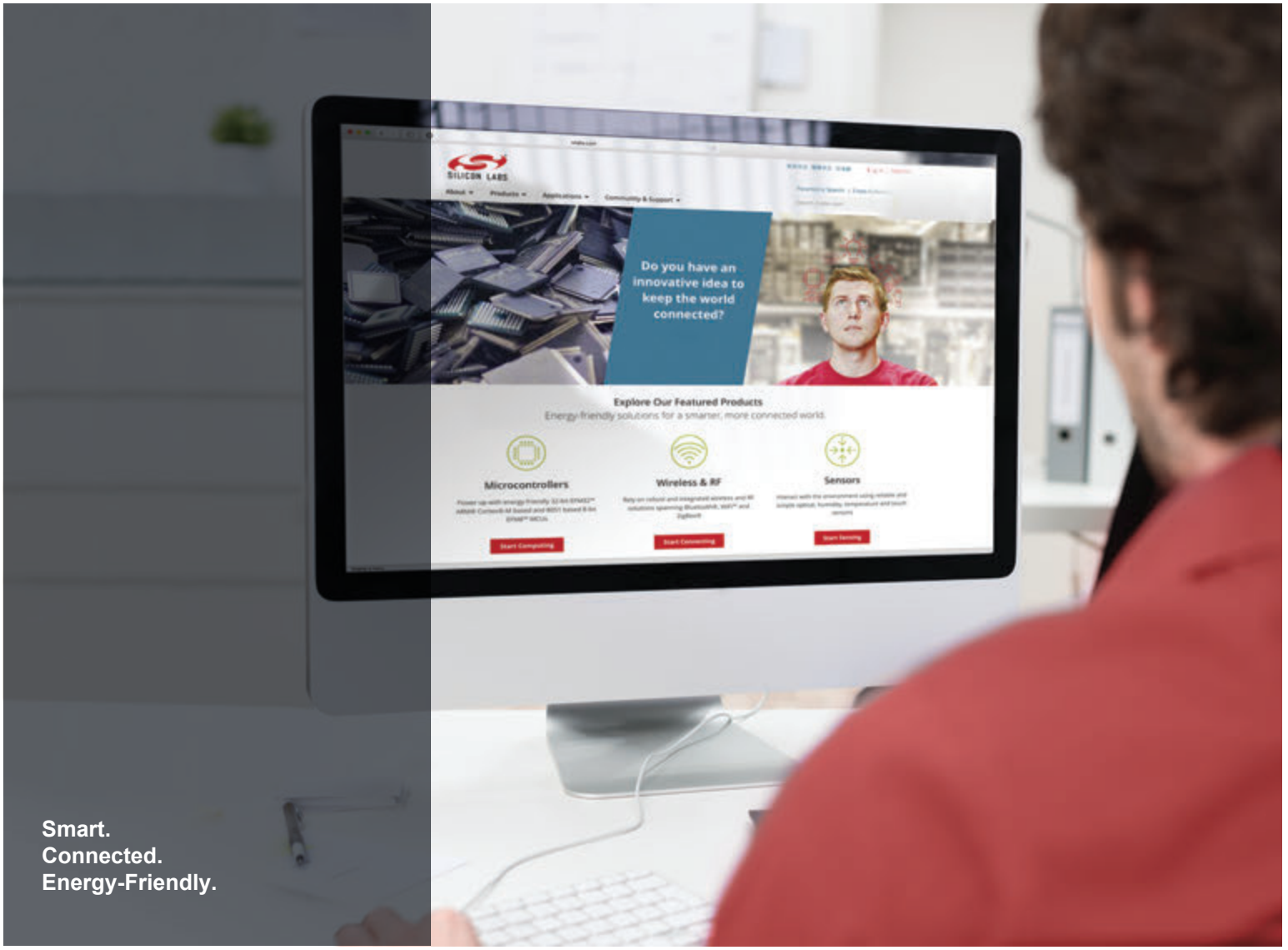
3.2 Persistent Store API

Persistent Store (PS Store) is used in Silicon Labs Bluetooth applications to store data in flash. In DMP applications, PS Store is set up to use the default NVM3 instance as its storage mechanism. The PS Store API is documented in the Bluetooth API Reference Manual.

16-bit keys are used with the PS Store API, which are then mapped to a 20-bit NVM3 key when NVM3 is used as the PS Store storage mechanism. The four most significant bits are set to 0x4 to place these objects in the Bluetooth domain of the NVM3 default instance key space. As the PS Store API is fixed to use only the Bluetooth domain, any objects to be placed in other domains, for example the User domain, should be created and accessed using the native NVM3 API.

3.3 Native NVM3 API

For code accessing NVM3 objects that does not need to be compatible with the token or PS Store APIs, using the native NVM3 API to access NVM3 data is recommended to reduce code size and allow using the full feature set of NVM3. Any PS Store object or token can also be accessed through the native NVM3 API using the same NVM3 key. Complete documentation of this API is found in the EMDRV ->NVM3 section of Gecko HAL & Driver API Reference Guide.



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