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 and Connect applications.

KEY POINTS

- Key/value pair data object storage in flash
- Wear-leveling to maximize flash lifetime
- Resilient to power and reset events
- Shared by EmberZNet, Connect, 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 Silicon Labs Connect (installed as part of the Flex SDK), as well as Persistent Store (PS Store) used with the Silicon Labs Bluetooth stack on EFR32 Series 1 devices. Because NVM3 can be used with both EmberZNet and Bluetooth, it allows a single data storage instance to be shared in Dynamic Multiprotocol (DMP) applications with Bluetooth and EmberZNet.

NVM3 is designed to work in EmberZNet, Connect, 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 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 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 with EmberZNet, Connect, or 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 EmberZNet, Connect, or Bluetooth applications. 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 shall trigger. This can be done by setting a positive value in the repackHeadroom variable in the initialization structure. 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 EmberZNet, Connect, or Bluetooth 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 200 elements, but it can be configured either in the NVM3 Library plugin option in AppBuilder, as shown in Figure 2.1 NVM3 Library Plugin in AppBuilder on page 5, or using defined symbols as described in section 2.5 Using NVM3 with the Bluetooth SDK.

Note: The cache size must be set to a value greater than or equal to the number of objects found in NVM3. 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 cache must also be large enough to hold any deleted NVM3 objects.

The `nvm3_countObjects()` and `nvm3_countDeletedObjects()` functions can be used to find the number of live and deleted objects in NVM3 at any given point. Silicon Labs recommends checking these functions after initialization of tokens, PS Store, and native NVM3 objects to figure out the correct size of the NVM3 default cache size. The cache must be large enough to hold the sum of both the live and deleted objects.

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
Connect 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, Connect, or DMP example application, the **NVM3 Library** plugin should be included in the project. All PS Store and SimEE plugins should be deselected.

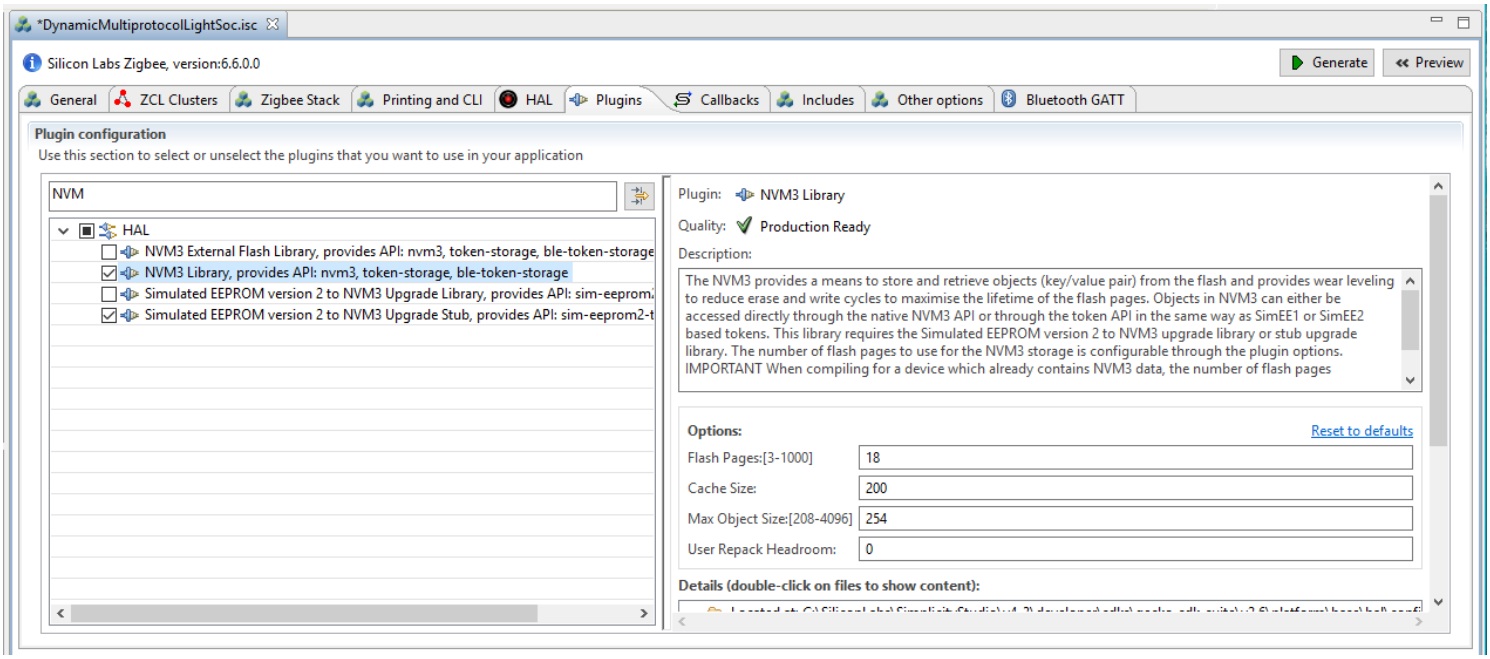


Figure 2.1. NVM3 Library Plugin in AppBuilder

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 and deleted objects, stored in NVM3 at any time.
- **Max Object Size:** Size of largest allowed NVM3 object in bytes. Must be between 208 and 4096 bytes. Note that the token API can only be used to access objects of 254 bytes or smaller. When accessing larger objects, the native NVM3 API must be used.
- **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.

When the NVM3 Library plugin is used the **Simulated EEPROM version 2 to NVM3 Upgrade Library** or **Simulated EEPROM version 2 to NVM3 Upgrade Stub Library** must be included, as described in section 2.4 SimEEv2 to NVM3 Upgrade Plugin.

2.4 SimEEv2 to NVM3 Upgrade Plugin

An AppBuilder plugin (**Simulated EEPROM version 2 to NVM3 Upgrade Library**) is provided for EmberZNet applications that upgrade 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.

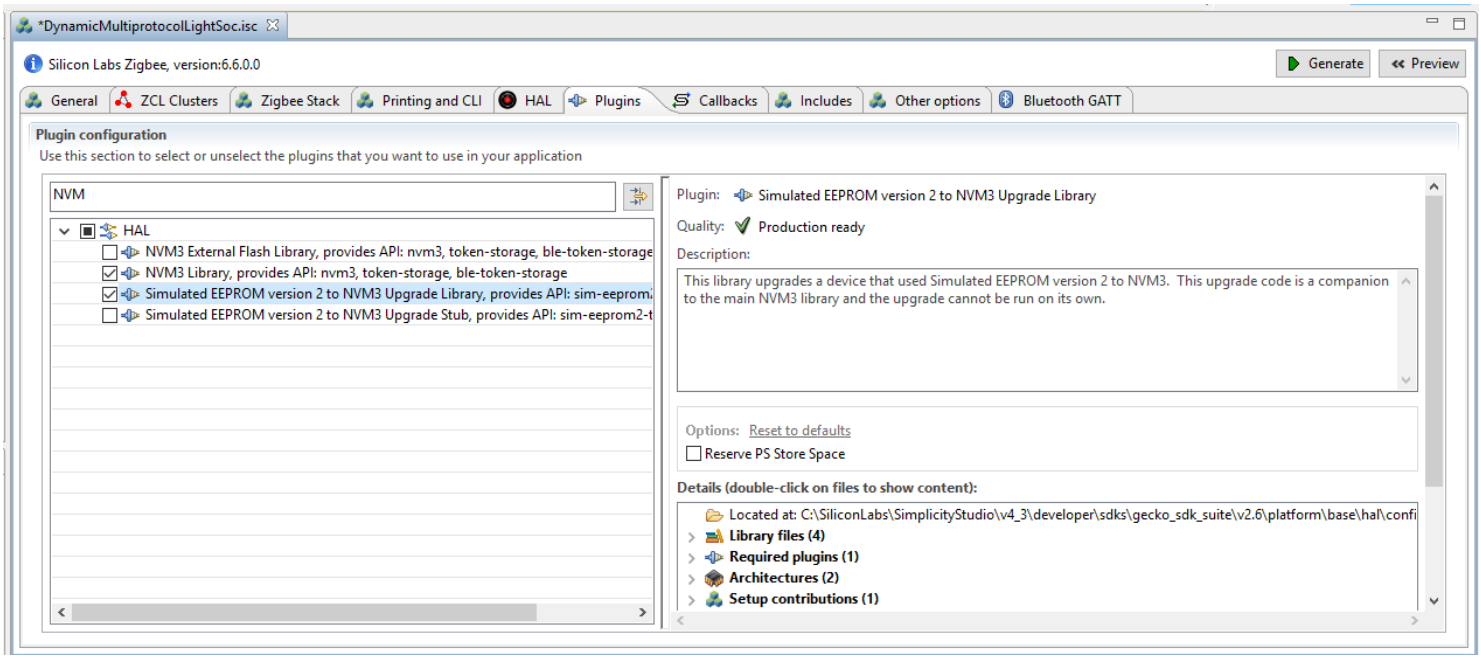


Figure 2.2. SimEEv2 to NVM3 Upgrade Library and Stub Plugins in AppBuilder

2.5 Using NVM3 with the Bluetooth SDK

Traditionally the Bluetooth stack uses its own proprietary solution to store data in non-volatile memory, called **Persistent Store (PS Store)**. PS Store stores both data handled by the stack (such as temporary Bluetooth address, bonding keys, and so on) and user data (such as the device state) that has to be preserved on resetting the device. To learn more about PS Store, read the related section of the Bluetooth API Reference Guide.

Some of the sample applications in the Bluetooth SDK are still configured to use PS Store, while others are already configured to use NVM3. This means that:

- On Series 1 devices sample apps are configured to use PS Store, except Bluetooth Mesh NCP sample projects, where NVM3 is used by default.
- On Series 2 devices all sample apps are configured to use NVM3.

While Series 2 devices support NVM3 only, on Series 1 devices both PS Store and NVM3 can be used. This section describes how to configure NVM3 in the Bluetooth SDK, and how to switch between PS Store and NVM3 if needed.

2.5.1 Configuring NVM3 in the Bluetooth SDK

In the Bluetooth SDK AppBuilder is not available to configure NVM3 parameters such as flash pages or cache size. Therefore, the parameters have to be defined manually. To overwrite the default parameters:

1. Open the project settings.
2. Find the defined symbols:
   a. If you use GCC as your compiler, go to C/C++ Build>Settings>GNU ARM C Compiler>Symbols>Defined Symbols
   b. If you use IAR as your compiler, go to C/C++ Build>Settings>IAR C/C++ compiler for ARM>Preprocessor>Defined Symbols
3. Add any of the following defines to overwrite the default parameters:

   NVM3_DEFAULT_NVM_SIZE

   NVM3_DEFAULT_CACHE_SIZE

   NVM3_DEFAULT_MAX_OBJECT_SIZE

   NVM3_DEFAULT_REPACK_HEADROOM

You can find the description of each parameter in section 2.3 NVM3 Library Plugin with one difference: instead of the number of flash pages, you must define NVM size in bytes. Define this to a number that is a multiple of the flash page size. (Note: On Series 1 devices the flash page size is typically 2 kB, while on Series 2 devices the flash page size is typically 8 kB. Check the datasheet for your device.)

2.5.2 Switching from PS Store to NVM3

Beginning with Bluetooth SDK v2.13.0, both PS Store and NVM3 are supported as non-volatile memory solutions on Series 1 devices. Most sample applications are configured to use PS Store by default, but for some applications (where larger non-volatile memory is needed) NVM3 may be a better solution.

Note: PS Store and NVM3 are not compatible with each other, therefore upgrading an already existing application from PS Store to NVM3 will result in losing all data stored on the device. If you have an application running in the field, it may be wiser to stay with PS Store. If you still want to upgrade, see *AN1086: Using the Gecko Bootloader with the Silicon Labs Bluetooth® Application* for details.

Note: PS Store uses only 2 flash pages (=4 kB on an EFR32BG1/12/13 device). Therefore, changing to NVM3 will affect the available space in flash. You must be particularly careful when you upgrade the firmware not to overwrite the NVM3 area with the application.

To change your project configuration from PS Store to NVM3, use the following procedure.

1. Copy the following folder with all of its content:

   `C:\SiliconLabs\SimplicityStudio\v4\developer\sdks\gecko_sdk_suite\<version>\platform\emdrv\nvm3`

   under the `/platform/emdrv` folder of your project.
2. Remove the following files from the project:
   - `/platform/emdrv/nvm3/src/nvm3_hal_extflash.c`
   - `/platform/emdrv/nvm3/src/nvm3_default_extflash.c` (NVM3 use with external flash is deprecated)
3. If you use Apploader in your project, also copy the NVM3 version of Apploader from

   `C:\SiliconLabs\SimplicityStudio\v4\developer\sdks\gecko_sdk_suite\<version>\protocol\bluetooth\lib\<device>\<compiler>\binapploader_nvm3.o`

   into the `/protocol/bluetooth/lib/<device>/<compiler>` folder of your project.
4. **If you use GCC as a compiler:**
   a. Go to `Project > Properties > C/C++ Build > Settings > GNU ARM C Compiler > Includes`.
   b. Add `${workspace_loc:/${ProjName}/platform/emdrv/nvm3/inc}` to the include directory.
   c. Go to `Project > Properties > C/C++ Build > Settings > GNU ARM C Linker > Miscellaneous`.
   d. Remove `${workspace_loc:/${ProjName}/protocol/bluetooth/lib/<device>/<compiler>/libpsstore.a}`.
   e. add `${workspace_loc:/${ProjName}/platform/emdrv/nvm3/lib/libnvm3_CM4_gcc.a}`

   If you use IAR as a compiler:
   a. Go to `Project > Properties > C/C++ Build > Settings > IAR C/C++ Compiler for ARM > Preprocessor`.
   b. Add `${workspace_loc:/${ProjName}/platform/emdrv/nvm3/inc}` to the include directory.
   c. Go to `Project > Properties > C/C++ Build > Settings > IAR Linker for ARM > Library`.
   d. Remove `${workspace_loc:/${ProjName}/protocol/bluetooth/lib/<device>/<compiler>/libpsstore.a}`.
   e. Add `${workspace_loc:/${ProjName}/platform/emdrv/nvm3/lib/libnvm3_CM4_iar.a}`.
5. If you use Apploader, also modify

   `${workspace_loc:/${ProjName}/protocol/bluetooth/lib/<device>/<compiler>/binapploader.o}`

   to `${workspace_loc:/${ProjName}/protocol/bluetooth/lib/<device>/<compiler>/binapploader_nvm3.o}`.
6. Configure NVM3 as described in section 2.5.1 Configuring NVM3 in the Bluetooth SDK.

2.5.3 Switching from NVM3 to PS Store

It may happen that, for a reason such as backward compatibility, you have to change the configuration from NVM3 to PS Store. To change your project configuration from NVM3 to PS Store, use the following procedure:

1. Remove the `/paltform/emdrv/nvm3` folder from your project.
2. Copy the PS Store library from

   `C:\SiliconLabs\SimplicityStudio\v4\developer\sdks\gecko_sdk_suite\<version>\protocol\bluetooth\lib\<device>\<compiler>\libpsstore.a`

   into the `/protocol/bluetooth/lib/<device>/<compiler>` folder of your project.
3. If you use Apploader in your project, also copy the PS Store version of Apploader from

   `C:\SiliconLabs\SimplicityStudio\v4\developer\sdks\gecko_sdk_suite\<version>\protocol\bluetooth\lib\<device>\<compiler>\binapploader.o`

   into the `/protocol/bluetooth/lib/<device>/<compiler>` folder of your project.

   Note: PS Store is not supported on Series 2 devices (EFR32xG2x), therefore there is only an NVM3 version of the Apploader for these devices.

4. **If you use GCC as your compiler**
   a. Go to `Project > Properties > C/C++ Build > Settings > GNU ARM C Compiler > Includes`.
   b. Remove `${workspace_loc:/${ProjName}/platform/emdrv/nvm3/inc}` from the include directory.
   c. Go to `Project > Properties > C/C++ Build > Settings > GNU ARM C Linker > Miscellaneous`.
   d. Add `${workspace_loc:/${ProjName}/protocol/bluetooth/lib/<device>/<compiler>/libpsstore.a}`.
   e. Remove `${workspace_loc:/${ProjName}/platform/emdrv/nvm3/lib/libnvm3_CM4_gcc.a}`.

   If you use IAR as your compiler
   a. Go to `Project > Properties > C/C++ Build > Settings > IAR C/C++ Compiler for ARM > Preprocessor`.
   b. Remove `${workspace_loc:/${ProjName}/platform/emdrv/nvm3/inc}` from the include directories.
   c. Go to `Project > Properties > C/C++ Build > Settings > IAR Linker for ARM > Library`.
   d. Add `${workspace_loc:/${ProjName}/protocol/bluetooth/lib/<device>/<compiler>/libpsstore.a}`.
   e. Remove `${workspace_loc:/${ProjName}/platform/emdrv/nvm3/lib/libnvm3_CM4_iar.a}`.
5. If you use Apploader, also change

   `${workspace_loc:/${ProjName}/protocol/bluetooth/lib/<device>/<compiler>/binapploader_nvm3.o}`

   to `${workspace_loc:/${ProjName}/protocol/bluetooth/lib/<device>/<compiler>/binapploader.o}`.

2.6 NVM3 Stack Usage

NVM3 uses up to 420 bytes of stack when the max object size is set to 1900 bytes or lower, and any operating system task stack that calls NVM3 functions should be large enough to account for the NVM3 stack usage.

2.7 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 a varying number of 2 kB or 8 kB flash pages ,and the minimum (208 bytes), default (254 bytes), high (1900 bytes) and maximum (4096 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	Max object size = 4096 bytes
3	6144	1596	1504	0	0
4	8192	3624	3532	240	0
5	10240	5652	5560	2268	0
6	12288	7680	7588	4296	0
7	14336	9708	9616	6324	0
8	16384	11736	11644	8352	0
9	18432	13764	13672	10380	1900
10	20480	15792	15700	12408	3928
11	22528	17820	17728	14436	5956
12	24576	19848	19756	16464	7984
13	26624	21876	21784	18492	10012
14	28672	23904	23812	20520	12040
15	30720	25932	25840	22548	14068
16	32768	27960	27868	24576	16096
17	34816	29988	29896	26604	18124
18	36864	32016	31924	28632	20152

Table 2.3. Max Allowed Basic Storage with 8 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	Max object size = 4096 bytes
3	24516	7740	7648	4356	0
4	32688	15912	15820	12528	8136
5	40860	24084	23992	20700	16308
6	49032	32256	32164	28872	24480

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 with the EmberZNet and Connect stacks, 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 generally 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. For indexed tokens, however, the token initialization checks if indexed tokens have more or less indexes than the number of NVM3 objects found in the indexed token's NVK3KEY range. If there are fewer indexes, the token initialization deletes the extra NVM3 objects. If the number of indexes has been increased, new NVM3 objects will be created to hold these indexes.

When NVM3 objects are deleted, the actual object data remains in NVM3 but is marked as deleted. The deleted object data remains in NVM3 and consumes cache space until NVM3 repacks have erased the page(s) holding all versions of these objects.

3.1.2 Special Considerations for Indexed Tokens

NVM3 does not have native support for indexed tokens. 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

While Bluetooth projects use NVM3 (and not PS Store) as the storage mechanism on Series 2 devices, the PS Store API can still be used in the same way as for Series 1 devices. The Bluetooth stack will automatically translate PS Store API calls to NVM3 API calls in the background. The same applies for Zigbee + Bluetooth DMP projects, where NVM3 is applied as the storage mechanism, but the PS Store API can still be used. 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 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, must be created and accessed using the native NVM3 API.

If you want to use the PS Store API and the native NVM3 API in the same app, then:

1. Call `gecko_init(pconfig)` to initialize PS Store and open its own NVM3 instance. This is done in all Bluetooth sample apps.
2. Open NVM3 by calling the `nvm3_open()` function with the default (!) parameters to open your NVM3 instance:

```
nvm3_open(nvm3_defaultHandle, nvm3_defaultInit);
```

User data can now be saved to:

- PS key range 0x4000 - 0x407F. All other PS keys (0x0000-0xFFFF) are reserved for the stack (for example for storing bonding data).
- NVM3 key range 0x00000-0x0FFFF (NVM3 user data area), and 0x44000-0x4407F (PS Store user data area).

For example, the following API calls will have the same effect:

- `gecko_cmd_flash_ps_save(0x4000,len,data);`
- `nvm3_writeData(nvm3_defaultHandle,0x44000,(void*)data,len);`

Similarly, you can read the same data with the following API calls:

- `gecko_cmd_flash_ps_load(0x4000);`
- `nvm3_readData(nvm3_defaultHandle,0x44000,(void*)read_buffer,maxlen);`

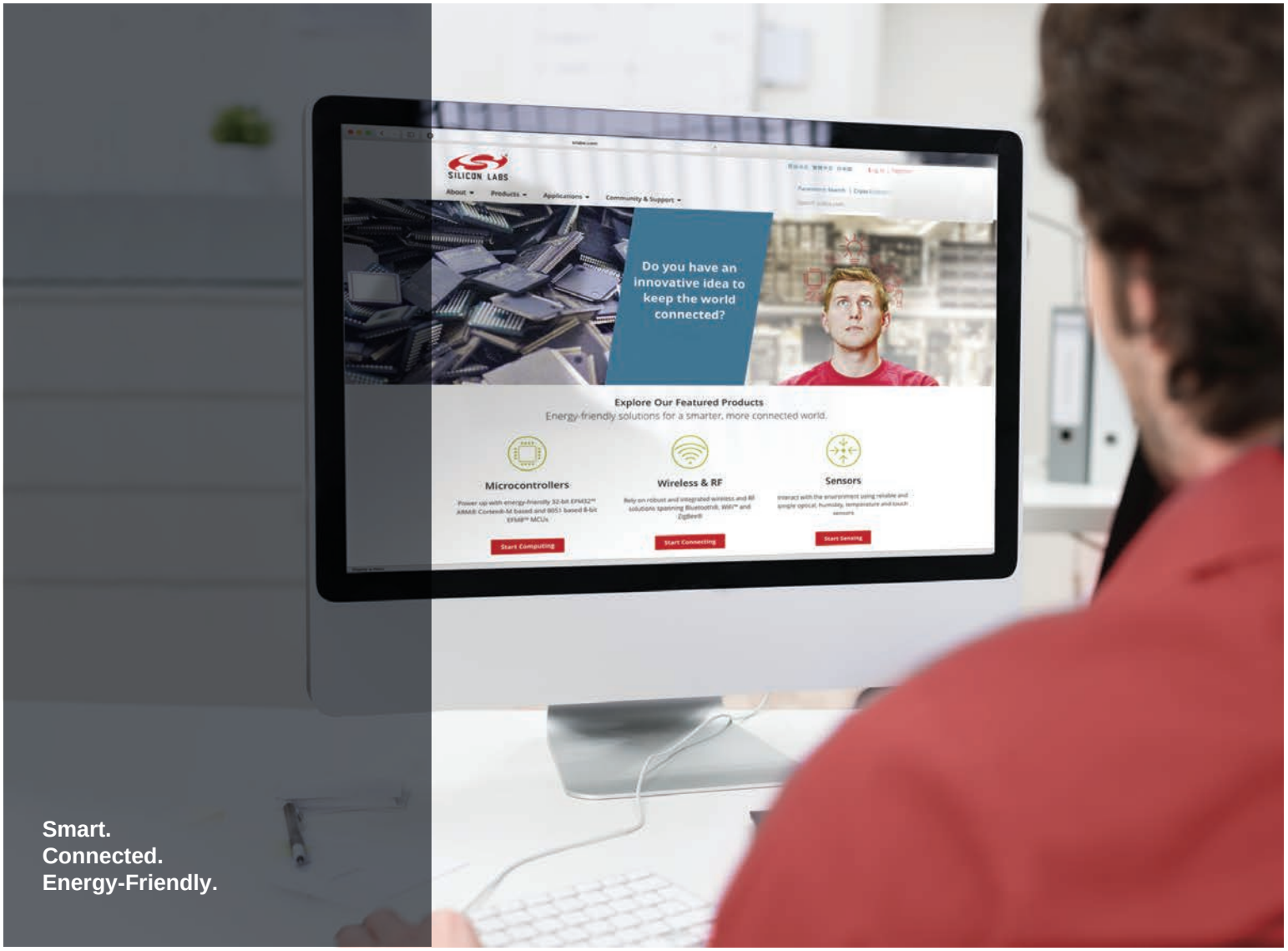
3.3 Native NVM3 API

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.

4. Simplicity Commander and NVM3

Simplicity Commander is a single, all-purpose tool to be used in a production environment. It is invoked using a simple CLI (Command Line Interface) that is also scriptable. Simplicity Commander supports reading out the NVM3 data area from a device and parsing the NVM3 data to extract stored values. This can be useful in a debugging scenario where you may need to find out the stored state of an application that has been running for some time.

More information on how to use the Simplicity Commander with NVM3 can be found in *UG162: Simplicity Commander Reference Guide*.



Products
www.silabs.com/products

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

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

Trademark Information
Silicon Laboratories Inc.® , Silicon Laboratories®, Silicon Labs®, SiLabs® and the Silicon Labs logo®, Bluegiga®, Bluegiga Logo®, ClockBuilder®, CMEMS®, 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®, Gecko OS, Gecko OS Studio, ISOmodem®, Precision32®, ProSLIC®, Simplicity Studio®, SiPHY®, Telegesis, the Telegesis Logo®, USBXpress® , Zentri, the Zentri logo and Zentri DMS, Z-Wave®, and others are trademarks or registered trademarks of Silicon Labs. ARM, CORTEX, Cortex-M3 and THUMB are trademarks or registered trademarks of ARM Holdings. Keil is a registered trademark of ARM Limited. Wi-Fi is a registered trademark of the Wi-Fi Alliance. All other products or brand names mentioned herein are trademarks of their respective holders.

SILICON LABS

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

http://www.silabs.com