AN1154: Using Tokens for Non-Volatile Data Storage

This application note describes tokens and shows how to use them for non-volatile data storage in EmberZNet PRO and Silicon Labs Connect (part of the Flex SDK).

KEY POINTS

- Defines tokens.
- Discusses dynamic token access.
- Explains manufacturing tokens.
- Describes default tokens.
- Details how to add a custom application dynamic token.

1. Introduction

This application note explains how to use tokens for non-volatile data storage. First, the concept of tokens, different types of tokens, and how to categorize them from multiple perspectives is introduced. Then, practical instructions on how to create and access (that is, read and write) tokens from the application are provided.

After reading this application note you should expect to understand what tokens are, why and when you would use them, and how to create and access them from certain types of applications, currently EmberZNet PRO and Silicon Labs Connect (part of the Flex SDK).

A new infrastructure was introduced with Gecko SDK Suite (GSDK) v3.x. As part of that infrastructure update, the HAL APIs used to access and manage tokens have been replaced with Token Manager APIs. Variances are noted throughout.

2. About Tokens

2.1 Concept of Tokens

A token is an abstract data constant that has special persistent meaning for an application. Tokens are used to preserve certain important data across reboots and during power loss. These tokens are stored in non-volatile memory. 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. Usually it is clear from the context which meaning to use. In this application note, "token" always refers to the key + data pair.

The rest of this section examines the categorization of tokens from a few different angles. The following figure is a conceptual illustration of the “universe” of tokens, and how this application note relates to other documents on non-volatile data storage. Understanding the types of tokens is relevant to understanding how to create and use them, as described in Section 3. Creating and Accessing Tokens.

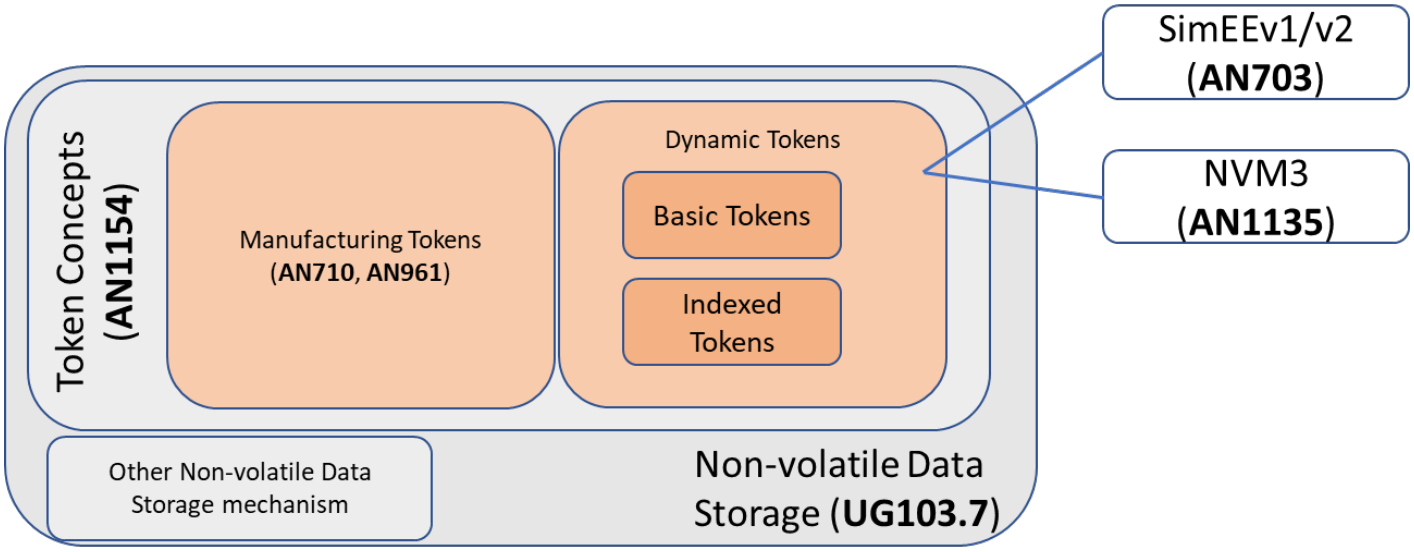


Figure 2.1. Non-volatile Data Storage and Tokens

2.2 Types of Tokens: Manufacturing Tokens and Dynamic Tokens

Tokens are written differently depending on how they are going to be used.

Manufacturing tokens are written either only once or very infrequently during the lifetime of the chip, and they are stored at absolute addresses of the flash. For more information about manufacturing and token programming, refer to document *AN710: Bringing up Custom Devices for the Ember® EM35x SoC or NCP Platform* and *AN961: Bringing up Custom Devices for the EFR32MG and EFR32FG Families.*

Dynamic tokens can be accessed (both read and written) frequently. They are stored in a dedicated area of the flash which uses a memory-rotation algorithm to prevent flash overuse. Silicon Labs offers three different dynamic token implementations: Simulated EEPROM Version 1 (SimEEv1), Simulated EEPROM Version 2 (SimEEv2), and Third Generation Non-Volatile Memory (NVM3). For an overview of non-volatile data storage concepts and a description of the three implementations, see *UG103.7: Non-Volatile Data Storage Fundamentals*.

The fundamental purpose of the dynamic token system as compared to generic RAM usage 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 exact storage location of the data. This simplifies application design and code reuse.

Because EM3x and EFR32 process technology does not offer an internal EEPROM, the storage mechanism for dynamic tokens is implemented to use a section of internal flash memory for stack and application token storage. For SimEEv1, the EM35x and EFR32 Series 1 use 8 kB of upper flash memory for non-volatile data storage. SimEEv2 requires 36 kB of upper flash storage. Using SimEEv2 requires a special key from Silicon Labs. The purpose of this is to prevent an unintended upgrade from Version 1 to Version 2. The only way to downgrade requires full data loss and upgrading might not retain every token. With NVM3, storage size is configurable from 3 flash pages on up.

Parts that use dynamic tokens to store non-volatile data have different levels of flash performance with respect to guaranteed erase cycles. EM35x-I flash cells are qualified for a guaranteed 2,000 erase cycles across voltage and temperature, other EM35x flash cells are qualified for a guaranteed 20,000 erase cycles, and EFR32 flash cells are qualified for a guaranteed 10,000 erase cycles. See the datasheet for your specific part in order to determine the number of guaranteed erase cycles across voltage and temperature. Due to the limited erase cycles, the storage mechanism for dynamic tokens implements a wear-leveling algorithm that effectively extends the number of erase cycles for individual tokens.

Silicon Labs recommends that application designers familiarize themselves with the different dynamic token storage mechanisms, so that they design the application's use of tokens for optimal flash erase cycles. Refer to document *AN703: Using Simulated EEPROM Version 1 and Version 2 for the EM35x and EFR32 Series 1 SoC Platforms*, for more information about SimEEv1/v2. Note that SimEEv1/v2 are not implemented on EFR32 Series 2. Refer to *AN1135: Using Third Generation Non-Volatile Memory (NVM3) Data Storage* for more information about NVM3. If you are unsure where to start and are developing for EFR32, Silicon Labs recommends NVM3. NVM3 is more configurable (allows for better balance of token capacity versus reserved flash) and is compatible with DMP in case the application needs to go that way in the future. Here are some very general guidelines on the mechanisms' distinguishing features:

- Simulated EEPROM version1: low data requirement, single protocol, EM35x and EFR32 Series 1.
- Simulated EEPROM version2: high data requirement, single protocol, EM35x and EFR32 Series 1.
- NVM3: high data requirement, dynamic multi-protocol, EFR32 Series 2.

2.2.1 Types of Dynamic Tokens: Basic Tokens and Indexed Tokens

There are two types of dynamic tokens, with the types distinguished by their format:

- **Non-indexed or basic** dynamic tokens. These can be thought of as a simple char variable type. They can be used to store an array, but if one element changes the entire array must be rewritten.
  - A counter token is a special type of non-indexed dynamic token meant to store a number that increments by 1 at a time.
- **Indexed** dynamic tokens can be considered as a linked 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.

More information on basic and indexed tokens can be found in *AN703: Using Simulated EEPROM Version 1 and Version 2 for the EM35x and EFR32 Series 1 SoC Platforms* and *AN1135: Using Third Generation Non-Volatile Memory (NVM3) Data Storage*.

2.3 Types of Tokens: Default Tokens and Custom Tokens

Depending on whether a token is provided by Silicon Labs as part of a networking protocol stack or created by a user, the tokens can also be categorized as default tokens or custom tokens.

2.3.1 Default Tokens

The networking stack contains default tokens that are grouped by their software purpose:

- **Manufacturing Tokens** are set at manufacturing time and cannot be changed by the application.
- **Stack Tokens** are runtime configuration options set by the stack. These dynamic tokens should not be changed by the application.
- **Application Framework Tokens** are application tokens used by the Application Framework and generated by AppBuilder in Connect v2.7.x and EmberZNet SDK v6.7.x/v6.8.x. These dynamic tokens should not be changed by the application after project generation. Examples of these are ZCL attribute tokens and plugin tokens.

2.3.2 Custom Tokens

In addition to default tokens, users can add these types of tokens specific to their application:

- **Custom Manufacturing Tokens** are defined by the user and set at manufacturing time.
- **Custom Application Tokens** are additional application tokens users may add to meet their unique application needs for non-volatile data storage.

3. Creating and Accessing Tokens

Now that you understand what tokens are and a few ways they may be categorized depending on how you view them, you can learn how to use tokens. This includes knowing where to find default tokens, how to create new tokens, and how to read and potentially modify tokens. Keep in mind that methods may vary depending on the type of tokens involved.

3.1 Token Header Files

A token header file is simply a .h file that contains token definitions. Manufacturing tokens and dynamic tokens have separate token header files. There may be more than one header file in an application for dynamic tokens: one for stack tokens, variable number of application framework tokens, and possibly one for custom application tokens.

3.2 Creating Dynamic Tokens

Adding a dynamic token to the header file involves three steps:

1. Define the token name.
2. Add any typedef needed for the token, if it is using an application-defined type.
3. Define the token storage.

The rest of this section looks at each of these steps one at a time.

3.2.1 Define the Token Name

When defining the name, do not prepend the word `TOKEN`. For SimEEv1/v2 dynamic tokens, use the word `CREATOR`:

```
/**
* Custom Application Tokens
*/
// Define token names here
#define CREATOR_DEVICE_INSTALL_DATA (0x000A)
#define CREATOR_HOURLY_TEMPERATURES (0x000B)
#define CREATOR_LIFETIME_HEAT_CYCLES (0x000C)
```

For NVM3 dynamic tokens, use the word `NVM3KEY`. Note that the example below assumes a Zigbee application. For a different stack, the NVM3 domain would be different. Note also that the `NVM3KEY` value for `HOURLY_TEMPERATURES` is set to a value where the subsequent 0x7F values are unused because this is an indexed token. Refer to *AN1135: Using Third Generation Non-Volatile Memory (NVM3) Data Storage* for more information on NVM3 default instance key spaces and restrictions on selecting `NVM3KEY` values for indexed tokens.

```
/**
* Custom Zigbee Application Tokens
*/
// Define token names here
#define NVM3KEY_DEVICE_INSTALL_DATA  (NVM3KEY_DOMAIN_USER | 0x000A)
// This key is used for an indexed token and the subsequent 0x7F keys are also reserved
#define NVM3KEY_HOURLY_TEMPERATURES  (NVM3KEY_DOMAIN_USER | 0x1000)
#define NVM3KEY_LIFETIME_HEAT_CYCLES (NVM3KEY_DOMAIN_USER | 0x000C)
```

These examples define the token key and link it to a programmatic variable. The token names are actually `DEVICE_INSTALL_DATA`, `HOURLY_TEMPERATURES` and `LIFETIME_HEAT_CYCLES`, with different tags prepended to the beginning depending on the usage. Thus, they are referred to in the example code as `TOKEN_DEVICE_INSTALL_DATA`, and so on.

The token key values must be unique within this device. The token key is critical to linking application usage with the proper data. As such, a unique key should always be used when defining a new token or even changing the structure of an existing token. Always using a unique key guarantees a proper link between application and data. `CREATOR` code values are 16-bit and `NVM3KEY` code values are 20-bit. For SimEEv1/v2, the first bit is reserved for manufacturing tokens, stack tokens, and those application tokens defined by the application framework, so all custom tokens should have a token key less than 0x8000.

For NVM3, custom application tokens should use the `NVM3KEY_DOMAIN_USER` range so they do not collide with the stack tokens in the `NVM3KEY_DOMAIN_ZIGBEE` range.

3.2.2 Define the Token Type

The token type can be either a built-in C data type or defined as a custom data structure using `typedef`. Note that the token type must be defined only in one place, as the compiler will complain if the same data structure is defined twice.

Each token in the code examples in Section 3.2 Creating Dynamic Tokens is a different type; the `HOURLY_TEMPERATURES` and `LIFETIME_HEAT_CYCLES` types are built-in types in C, while the `DEVICE_INSTALL_DATA` type is a custom data structure:

```
#ifdef DEFINETYPES
// Include or define any typedef for tokens here
typedef struct {
int8u install_date[11] /** YYYY-mm-dd + NULL */
int8u room_number; /** The room where this device is installed */
} InstallationData_t;
#endif //DEFINETYPES
```

3.2.3 Define the Token Storage

After any custom types are defined, the token storage is defined. This informs the token management software about the tokens being defined. Each token, whether custom or default, gets its own entry in this part:

```
#ifdef DEFINETOKENS
// Define the actual token storage information here
DEFINE_BASIC_TOKEN(DEVICE_INSTALL_DATA,
InstallationData_t,
{0, {0,...}})
DEFINE_INDEXED_TOKEN(HOURLY_TEMPERATURES, int16u, HOURS_IN_DAY, {0,...})
DEFINE_COUNTER_TOKEN(LIFETIME_HEAT_CYCLES, int32u, 0}
#endif //DEFINETOKENS
```

The following expands on each step in this process.

```
DEFINE_BASIC_TOKEN(DEVICE_INSTALL_DATA,
InstallationData_t,
{0, {0,...}})
```

`DEFINE_BASIC_TOKEN` takes three arguments: the name (`DEVICE_INSTALL_DATA`), the data type (`InstallationData_t`), and the default value of the token if it has never been written by the application (`{0, {0,...}}`).

The default value takes the same syntax as C default initializers. In this case, the first value (`room_number`) is initialized to 0, and the next value (`install_date`) is set to all 0s because the {0,...} syntax fills the remainder of the array with 0.

The syntax of `DEFINE_COUNTER_TOKEN` is identical to `DEFINE_BASIC_TOKEN`.

`DEFINE_INDEXED_TOKEN` requires a length of the array—in this case, `HOURS_IN_DAY` or 24. Its final argument is the default value of every element in the array. Again, in this case it is initialized to all 0s.

3.3 Accessing Dynamic Tokens With HAL APIs

Refer to this information if you are developing with any EmberZNet or Connect in GSDK v2.7x, and with EmberZNet in GSDK v3.0.

The networking stack provides a simple set of APIs for accessing token data. The APIs differ slightly depending on the type of the tokens. The rest of this section discusses how to access tokens depending on the types. You can find the full documentation in the stack API Reference.

3.3.1 Accessing Basic (Non-indexed) Tokens

The non-indexed/basic token API functions include:

```
void halCommonGetToken(data, token)
void halCommonSetToken(token, data)
```

In this case, 'token' is the token key, and 'data' is the token data. Note that `halCommonGetToken()` and `halCommonSetToken()` are general token APIs that can be used for both basic dynamic tokens and manufacturing tokens (see Section 3.5.1 Accessing Manufacturing Tokens).

The following example explains the usage of these APIs. Assume an application needs to store configuration data at installation time. A basic token has been defined to use the token key `DEVICE_INSTALL_DATA` and the data structure looks like this:

```
typedef struct {
int8u install_date[11] /** YYYY-mm-dd + NULL */
int8u room_number; /** The room where this device is installed */
} InstallationData_t;
```

Then you can access it with a code snippet like this:

```
InstallationData_t data;
// Read the stored token data
halCommonGetToken(&data, TOKEN_DEVICE_INSTALL_DATA);
// Set the local copy of the data to new values
data.room_number = < user input data >
MEMCOPY(data.install_date, < user input data>, 0, sizeof(data.install_date));
// Update the stored token data with the new values
halCommonSetToken(TOKEN_DEVICE_INSTALL_DATA, &data);
```

3.3.1.1 Accessing Counter Tokens

There is a special API to increment the counter token (which is a special type of non-indexed tokens):

```
void halCommonIncrementCounterToken(token)
```

Note that although you can write the counter token with the common `halCommonSetToken()` call, doing so is inefficient and defeats the purpose of using a counter token.

The following example explains the usage of a counter token and the special API to increment the counter token. Counting the number of heating cycles a thermostat has initiated is a perfect use for a counter token. Assume it is named `LIFETIME_HEAT_CYCLES`, and it is an int32u data type.

```
void requestHeatCycle(void) {
/// < application logic to initiate heat cycle >
halCommonIncrementCounterToken(TOKEN_LIFETIME_HEAT_CYCLES);
}
int32u totalHeatCycles(void) {
int32u heatCycles;
halCommonGetToken(&heatCycles, TOKEN_LIFETIME_HEAT_CYCLES);
return heatCycles;
}
```

Note that to read a counter token, use `halCommonGetToken()` just as you would read a general basic token.

3.3.2 Accessing Indexed Tokens

The indexed token API functions include:

```
void halCommonGetIndexedToken(data, token, index)
void halCommonSetIndexedToken(token, index, data)
```

The following example explains the usage of these APIs for an indexed token. To store a set of similar values, such as an array of preferred temperature settings throughout the day, use the default data type int16s to store the desired temperatures and define an indexed token called `HOURLY_TEMPERATURES`.

A local copy of the entire data set would look like this:

```
int16s hourlyTemperatures[HOURS_IN_DAY]; /** 24 hours per day */
```

In the application code, you can access or update just one of the values in the day using the indexed token functions:

```
int16s getCurrentTargetTemperature(int8u hour) {
int16s temperatureThisHour = 0; /** Stores the temperature for return */
if (hour < HOURS_IN_DAY) {
halCommonGetIndexedToken(&temperatureThisHour,
TOKEN_HOURLY_TEMPERATURES, hour);
}
return temperatureThisHour;
}
void setTargetTemperature(int8u hour, int16s targetTemperature) {
if (hour < HOURS_IN_DAY) {
halCommonSetIndexedToken(TOKEN_HOURLY_TEMPERATURE, hour,
&temperatureThisHour);
}
}
```

3.4 Accessing Dynamic Tokens with Token Manager APIs

Refer to this information if you are developing with Connect in GSDK v3.0.

The networking stack provides a simple set of APIs for accessing token data. The APIs differ slightly depending on the type of the tokens. The rest of this section discusses how to access tokens depending on the types. You can find the full documentation in the stack API Reference.

3.4.1 Accessing Basic (Non-Indexed) Tokens

The non-indexed/basic token API functions include:

```
Ecode_t sl_token_get_data(uint32_t token,
uint32_t index,
void *data,
uint32_t length);

Ecode_t sl_token_set_data(uint32_t token,
uint32_t index,
void *data,
uint32_t length);
```

In this case, 'token' is the token key, ‘index’ is 1 for non-indexed/basic tokens, 'data' is the token data, and ‘length’ is the size in bytes of the data. Note that `sl_token_get_data()` and `sl_token_set_data()` are specific for basic dynamic tokens. For manufacturing tokens see section 3.5.1 Accessing Manufacturing Tokens.

The following example explains the usage of these APIs. Assume an application needs to store configuration data at installation time. A basic token has been defined to use the token key `DEVICE_INSTALL_DATA` and the data structure looks like this:

```
typedef struct {
int8u install_date[11] /** YYYY-mm-dd + NULL */
int8u room_number; /** The room where this device is installed */
} InstallationData_t;
```

Then you can access it with a code snippet like this:

```
InstallationData_t data;
// Read the stored token data
sl_token_get_data(TOKEN_DEVICE_INSTALL_DATA, 1, &data, sizeof(data));
// Set the local copy of the data to new values data.room_number = < user input data >
MEMCOPY(data.install_date, < user input data>, 0, sizeof(data.install_date));
// Update the stored token data with the new values
sl_token_set_data(TOKEN_DEVICE_INSTALL_DATA, 1, &data, sizeof(data));
```

3.4.1.1 Accessing Counter Tokens

There is a special API to increment the counter token (which is a special type of non-indexed tokens):

```
Ecode_t sl_token_increment_counter(uint32_t token);
```

Note that although you can write the counter token with the common `sl_token_set_data()` call, doing so is inefficient and defeats the purpose of using a counter token.

The following example explains the usage of a counter token and the special API to increment the counter token. Counting the number of heating cycles a thermostat has initiated is a perfect use for a counter token. Assume it is named `LIFETIME_HEAT_CYCLES`, and it is an int32u data type.

```
void requestHeatCycle(void) {
/// < application logic to initiate heat cycle >
sl_token_increment_counter(TOKEN_LIFETIME_HEAT_CYCLES);
}
int32u totalHeatCycles(void) {int32u heatCycles;
sl_token_get_data(TOKEN_LIFETIME_HEAT_CYCLES, 1, &heatCycles, sizeof(heatCycles));
return heatCycles;
}
```

Note that to read a counter token, use `sl_token_get_data()` just as you would read a general basic token.

3.4.2 Accessing Indexed Tokens

The indexed token API functions are the same as the non-indexed/basic token API functions. The only difference is the 'index' parameter can be something other than 1.

The following example explains the usage of these APIs for an indexed token. To store a set of similar values, such as an array of preferred temperature settings throughout the day, use the default data type int16s to store the desired temperatures and define an indexed token called `HOURLY_TEMPERATURES`.

A local copy of the entire data set would look like this:

```
int16s hourlyTemperatures[HOURS_IN_DAY]; /** 24 hours per day */
```

In the application code, you can access or update just one of the values in the day using the indexed token functions:

```
int16s getCurrentTargetTemperature(int8u hour) {
int16s temperatureThisHour = 0; /** Stores the temperature for return */
if (hour < HOURS_IN_DAY) {
 sl_token_get_data(TOKEN_HOURLY_TEMPERATURES, hour, &temperatureThisHour, sizeof(temperatureThisHour));
}
return temperatureThisHour;
}
void setTargetTemperature(int8u hour, int16s targetTemperature) {
if (hour < HOURS_IN_DAY) {
 sl_token_set_data(TOKEN_HOURLY_TEMPERATURE, hour, &temperatureThisHour, sizeof(temperatureThisHour));
}
}
```

3.5 Manufacturing Tokens

Manufacturing tokens are defined in a similar way as basic (non-indexed) dynamic tokens, but use the DEFINE_MFG_TOKEN in the manufacturing token header, instead of the other DEFINE_*_TOKEN macros. Note that the CREATOR_* prefix – not NVM3KEY_* - should always be used with manufacturing tokens, even when the NVM plugin is being used for non-volatile storage. Refer to section 3.2 Creating Dynamic Tokens and to the API documentation on the DEFINE_MFG_TOKEN macro for more information.

There is a major difference, however, on where manufacturing tokens are stored and how they are accessed. Manufacturing tokens reside in the dedicated flash page for manufacturing tokens (with fixed absolute addresses). The rest of this section describes the necessary considerations for accessing manufacturing tokens.

3.5.1 Accessing Manufacturing Tokens

Manufacturing tokens, as the name suggests, are usually written once at manufacturing time into fixed locations in a dedicated flash page. Because their addresses are fixed, they can be easily read from external programming tools. Note, however, when the Read Protection feature has been enabled on the chip or on the dedicated flash page, manufacturing tokens can only be read from on-chip code.

Manufacturing tokens are not meant to be written often or from on-chip code, because they are at fixed locations. The same flash cell cannot be written repeatedly without erase operations in between. Writing a manufacturing token from on-chip code works only if the token is currently in an erased state. This means that manufacturing tokens can only be overwritten with external programming tools and not with on-chip code. Overwriting any manufacturing token that has been already written requires erasing the dedicated flash page for the manufacturing tokens. If Read Protection is enabled, it must be disabled first, which will erase the contents of the chip as a side effect. Manufacturing tokens are not wear-leveled so they should be overwritten sparingly.

Acess manufacturing tokens with their own dedicated APIs, which take the same parameters as the basic token APIs:

- HAL: `halCommonGetMfgToken()` and `halCommonSetMfgToken()`
- Token Manager: `sl_token_get_manufacturing_data()` and `sl_token_set_manufacturing_data()`

The two primary purposes for using the dedicated manufacturing token access APIs are:

1. For slightly faster access;
2. For access early in the boot process before `emberInit()` or `sl_token_init()` is called.

If you are using the HAL APIs, manufacturing tokens can also be accessed through the basic token APIs: `halCommonGetToken()` and `halCommonSetToken()`.

3.6 Accessing Default and Custom Tokens

3.6.1 Where to Find Default Token Definitions

To view the stack tokens, refer to the file:

```
<install-dir>/stack/config/token-stack.h
```

To view the Application Framework tokens, refer to the file `<project_name>_tokens.h` and the protocol-specific token file (such as `znet-token.h`) under your project directory after the project has been generated in AppBuilder in Connect v2.7.x and EmberZNet SDK v6.7.x/v6.8.x SDKs. `<project_name>_tokens.h` contains tokens for ZCL attributes that the application has selected to be stored in non-volatile memory. The protocol-specific token file includes plugin token headers and the custom application token header.

To view the manufacturing tokens for the EFR32 or EM3x series of chips refer to the following files, respectively:

```
<install-dir>/hal/micro/cortexm3/efm32/token-manufacturing.h
<install-dir>/hal/micro/cortexm3/token-manufacturing.h
```

Search for `CREATOR` to see the defined names. If the entire file seems overwhelming, focus only on the section describing the tokens. Some of the fixed manufacturing tokens may be set by the manufacturer when the board is created. For example, a custom EUI-64 address may be set by the vendor to override the internal EUI-64 address provided by Silicon Labs. Other tokens, such as the internal EUI-64, cannot be overwritten.

3.6.2 Add Custom Tokens

Refer to Section 3.2 Creating Dynamic Tokens on the methods and APIs for creating custom tokens. You can also refer to the stack token definition file as mentioned near the top of Section 3.6.1 Where to Find Default Token Definitions as a guide for creating custom application tokens.

After creating the tokens in a custom token header file, you need one more step: add the header file to the application by using the **Includes** tab in the .isc file in Simplicity Studio under the “Token Configuration” section. If you define both custom manufacturing tokens and custom application tokens, Silicon Labs recommends that you separate them into two different header files.

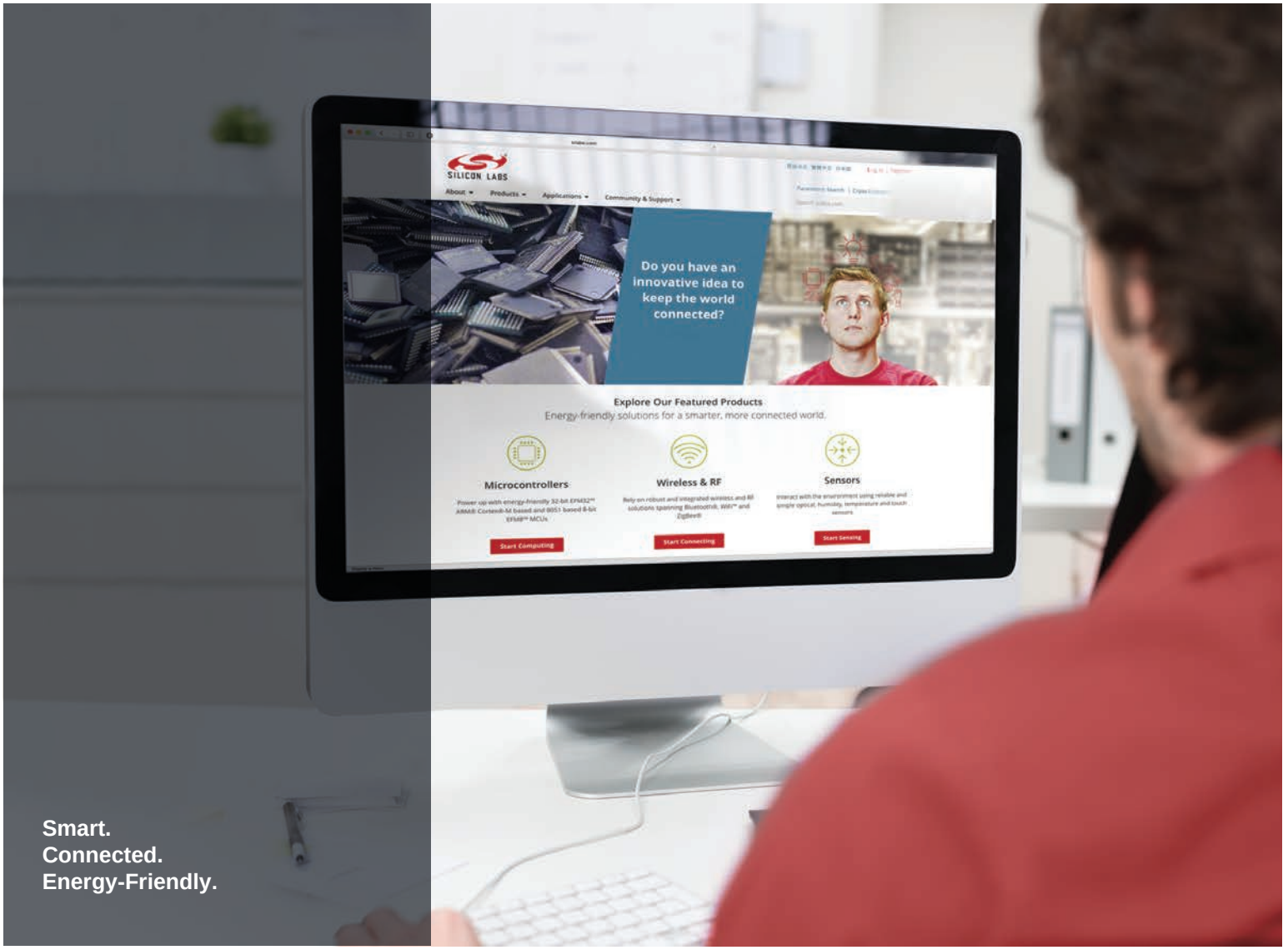
Note: For custom token files, the inclusion guards (#ifndef) seen in `token-stack.h` should not be used.

4. Token Manager Component

To use the Token Manager, the token_manager component (token_manager.slcc) should be added to the project. The token_manager component will default to using the token_manager_nvm3 component, which brings in the NVM3 system. Using NVM3 for storage is highly recommended.

To use Simulated EEPROM v1 or v2 (SimEEv1, SimEEv2) for storage, the component token_manager_simee1 or token_manager_simee2 should be part of the project along with token_manager.

To upgrade SimEEv2 to NVM3, the sim_eeprom2_to_nvm3_upgrade component should be used along with token_manager. This upgrade component will pull in the necessary SimEE and NVM3 code.



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