AN1154: Using Tokens for Non-Volatile Data Storage

This document describes tokens and shows how to use them for non-volatile data storage in EmberZNet PRO, Silicon Labs Flex, and Silicon Labs Thread applications.

KEY POINTS

- Tokens defined.
- Dynamic token access discussed.
- Manufacturing tokens explained.
- Default tokens described.
- Adding a custom application dynamic token described.

1 About 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 document token always refers to the key + data pair.

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.

Dynamic tokens need to be accessed frequently. They are stored in a dedicated area of the flash where we use 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*.

There are two types of dynamic tokens:

- **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.

2 Dynamic Tokens

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 utilize either 4 kB or 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 SoC Platforms*, for more information about SimEEv1/v2. Refer to *AN1135: Using Third Generation Non-Volatile Memory (NVM3) Data Storage* for more information about NVM3.

The networking stack provides a simple set of APIs for accessing token data. The full documentation may be found in stack API reference. 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.

The indexed token API functions include:

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

There is also 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.

2.1 Defining 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.

2.1.1 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 as 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_ZIGBEE | 0x000A)
// This key is used for an indexed token and the subsequent 0x7F keys are also reserved
#define NVM3KEY_HOURLY_TEMPERATURES  (NVM3KEY_DOMAIN_ZIGBEE | 0x1000)
#define NVM3KEY_LIFETIME_HEAT_CYCLES (NVM3KEY_DOMAIN_ZIGBEE | 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 and 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.

2.1.2 Define the Token Type

Each token in the CREATOR code example above is a different type; however, the `HOURLY_TEMPERATURES` and `LIFETIME_HEAT_CYCLES` types are built-in types in C. Only the `DEVICE_INSTALL_DATA` type is a custom data structure.

The token type is defined using the structure introduced in section 4.2 Custom Tokens and repeated next. 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.

```
#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
```

2.1.3 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 0.

2.2 Accessing Basic (Non-indexed) Tokens

Some applications may need to store configuration data at installation time. Usually, this is a good application for a basic token. Assume 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);
```

2.2.1 Accessing Counter Tokens

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.

```
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;
}
```

2.3 Accessing Indexed Tokens

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 Manufacturing Tokens

Manufacturing tokens are defined and treated as basic (non-indexed) tokens. The major difference is where the tokens are stored and how they are accessed, that is in the dedicated flash page for manufacturing tokens (with fixed absolute addresses). When the Read Protection feature has been enabled on the chip, manufacturing tokens can only be read from on-chip code. If the Read Protection feature has not been enabled, they can also be read by external programming tools. Writing a manufacturing token from on-chip code works only if the token is currently in an erased state. Manufacturing tokens should be accessed with their own dedicated API, `halCommonGetMfgToken()` and `halCommonSetMfgToken()`, which take the same parameters as the basic token APIs. The two primary purposes for using the `MfgToken` APIs are 1) for slightly faster access and 2) access early in the boot process before `emberInit()` is called. Manufacturing tokens can also be accessed through the basic token APIs `halCommonGetToken()` and `halCommonSetToken()`.

Manufacturing tokens can only be overwritten with external programming tools, and not with on-chip code, since 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.

4 Default and Custom Tokens

4.1 Default Tokens

The networking stack contains some default tokens. These tokens are grouped by their software purpose:

- **Stack Tokens** are runtime configuration options set by the stack; these should not be changed by the application.
- **Application Framework Tokens** are application tokens used by the Application Framework and generated by AppBuilder; these should not be changed by the application after project generation. Examples of these are ZCL attribute tokens and plugin tokens.
- **Manufacturing Tokens** are set at manufacturing time and cannot be changed by the application.

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 files `<project_name>_tokens.h` and `afv2-token.h` under your project directory after the project has been generated in AppBuilder. `<project_name>_tokens.h` contains tokens for ZCL attributes that the application has selected to be stored in non-volatile memory. The file `afv2-token.h` 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.

For more information about manufacturing and token programming, refer to document *AN710: Bringing up Custom Devices for the Ember® EM35x SoC or NCP Platform*.

4.2 Custom Tokens

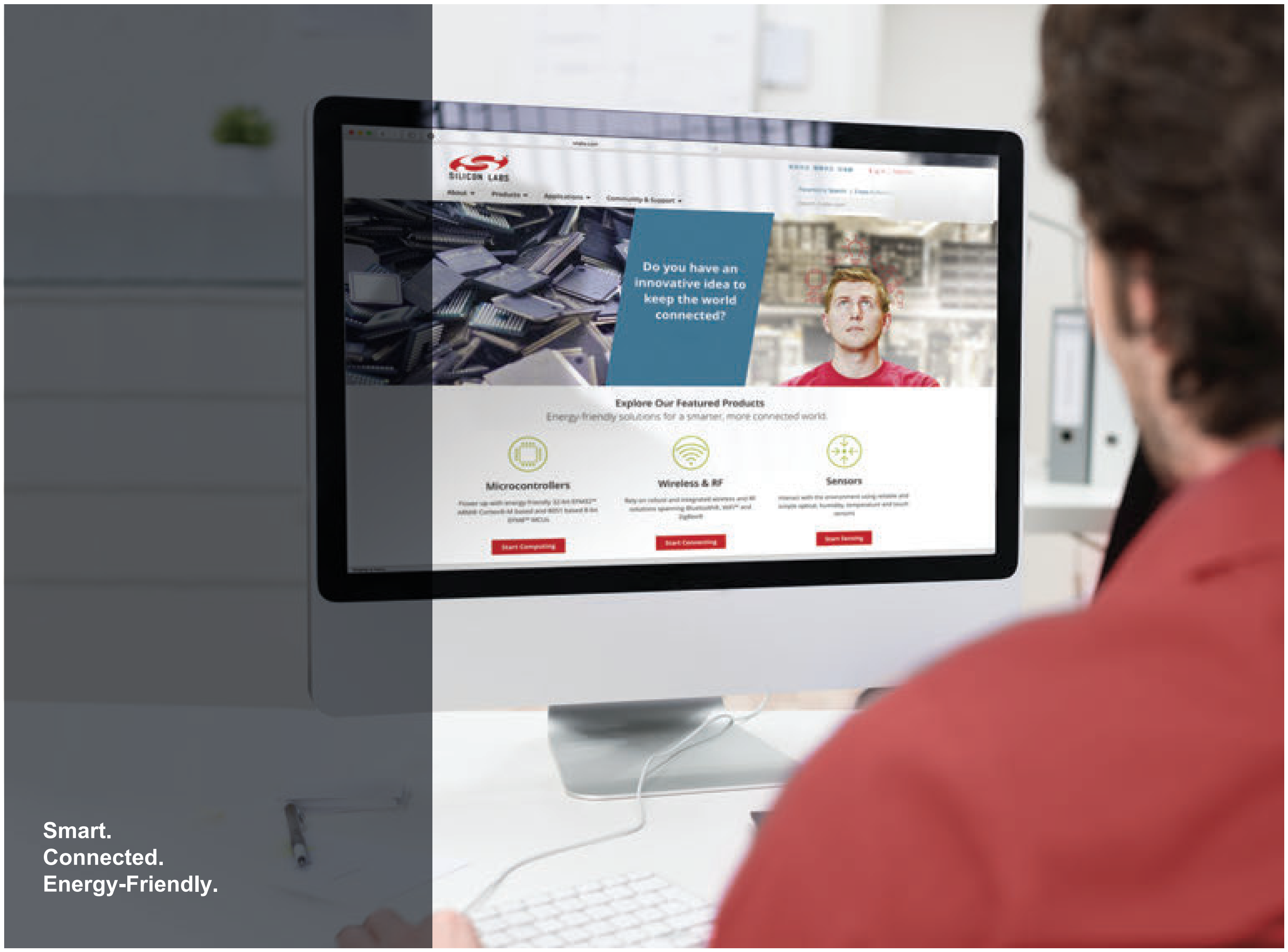
Custom application dynamic tokens and manufacturing tokens can be added through the Includes tab in the .isc file in Simplicity Studio, under the “Token Configuration” section. For details on defining custom application dynamic tokens see section 2.1 Defining Tokens. Defining custom manufacturing tokens is less common; you can refer to the default manufacturing token definition files (see section 4.1 Default Tokens) for a full explanation and examples.

The custom application dynamic token file should have the following structure:

```
/**
* Custom Application Tokens
*/
// Define token names here
#ifdef DEFINETYPES
// Include or define any typedef for tokens here
#endif //DEFINETYPES
#ifdef DEFINETOKENS
// Define the actual token storage information here
#endif //DEFINETOKENS
```

You can refer to the stack token definition file, `<install-dir>/stack/config/token-stack.h`, as a guide for creating application tokens.

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



Products
www.silabs.com/products

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

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

http://www.silabs.com