UG103.7: Application Development Fundamentals: Tokens

This document describes tokens and shows how to use them in EmberZNet PRO, EmberZNet RF4CE, and Silicon Labs Thread applications. It also discusses bindings, the application-defined associations between two devices on a network, used in EmberZNet PRO applications.

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

KEY POINTS

- Accessing the various token types
- Mechanics of custom tokens
- EmberZNet PRO bindings

1. Introduction

A token is an abstract data constant that has special persistent meaning for an application. This document describes tokens and shows how to use them in code. A token can be one of several types:

- Standard
- Indexed
- Counter
- Manufacturing

Tokens can be categorized in the following general groups:

- Stack tokens – these tokens are read/write and defined in every application to support stack behavior. These tokens live in simulated EEPROM.
- Application tokens – these tokens are read/write and defined by the application to support application behavior. It is left open to a customer to decide if there are application tokens, how they are defined, and what they do. These tokens live in simulated EEPROM.
- Fixed manufacturing tokens – these tokens are non-modifiable read only tokens specific to each individual chip and set during chip production. They support key low-level and high-level functionality.
- Customer manufacturing tokens – these tokens are read only from on chip code and read/write from external programming tools. Some of these tokens have a defined purposes to support stack behavior. It is left open to a customer to decide if there are any additional manufacturing tokens, how they are defined, and what they do.

The document also discusses bindings, the application-defined associations between two devices on a network, used in EmberZNet PRO applications.

2. About Tokens

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 typically stored in NVRAM which is most commonly directly in flash (manufacturing tokens) or in the simulated EEPROM (stack and application tokens) which operates on top of flash.

2.1 Purpose

The fundamental purpose of the 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. Tokens are also useful when the underlying storage of the data may change over time across implementations.

Because EM3x process technology does not offer an internal EEPROM, a simulated EEPROM is implemented to use a section of internal flash memory for stack and application token storage. Parts that use the simulated EEPROM to store non-volatile data have different levels of flash performance with respect to guaranteed write cycles, specifically 2,000 and 20,000 write cycles. Recently, version 2 of the simulated EEPROM has been released. For version 1, the EM3x utilizes either 4 kB or 8 kB of upper flash memory to store the simulated EEPROM. For version 2, the simulated EEPROM requires 36 kB of upper flash storage. See the datasheet for your specific part in order to determine the number of guaranteed write cycles across voltage and temperature. Due to the limited write cycles, the simulated EEPROM implements a wear-leveling algorithm that effectively extends the number of write cycles for individual tokens.

The simulated EEPROM is designed to operate below the token module as transparently as possible. The simulated EEPROM requires flash pages to be erased as data moves around. The application controls the erasure of flash pages, because erasing a page will prevent interrupts from being serviced for 21 ms. However, the application is only required to implement one callback and periodically call one utility function. In addition, a status function is available to provide the application with two basic statistics about simulated EEPROM usage.

Silicon Labs recommends that application designers familiarize themselves with the simulated EEPROM and its limitations, so that they design the application's use of tokens for optimal flash write cycles. Refer to document AN703, *Using the Simulated EEPROM for the Ember EM3x SoC Platforms*, for more information.

2.2 Usage

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

The basic 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.

Two special types of tokens are available: **indexed tokens** and **counter tokens**. An indexed token can be used when the data to be stored is an array in which each element may be accessed and updated independently from the others. A counter token can be used when the default operation on the token is to retrieve it, increment it by one, and then store it again.

The API functions for these two types of tokens are:

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

The counter token can also be accessed with the normal `halCommonGetToken()`, `halCommonSetToken()` calls. Setting a counter token severely degrades simulated EEPROM performance, and therefore should only be done when absolutely necessary.

3. Accessing Standard (Non-indexed) Tokens

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

Instructions for configuring custom tokens are provided in section 7. Custom Tokens.

4. 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 a 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);
  }
}
```

5. 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;
}
```

6. Accessing Manufacturing Tokens

Manufacturing tokens are defined and treated as standard (non-indexed) tokens. The major difference is manufacturing tokens can only be read from on chip code and customer manufacturing tokens can only be written with external programming tools. In addition to being able to read manufacturing tokens with the usual standard token read, `halCommonGetToken()`, manufacturing tokens can also be read with their own dedicated API, `halCommonGetMfgToken()`. This MfgToken API takes the same parameters at the standard GetToken API. In general, MFG tokens should be accessed through the standard GetToken API. The two primary purposes for using the MfgToken API is for slightly faster access and early on in the boot process before `emberInit()` is called.

7. Custom Tokens

Custom application tokens are defined in a header file. The header file can be specific to each project, and is defined by the preprocessor variable `APPLICATION_TOKEN_HEADER`. Custom manufacturing tokens require editing an existing header file. This header file is located at hal/micro/cortexm3/token-manufacturing.h. Defining custom manufacturing tokens is generally considered a rare practice, but is still possible. Note that to define them requires editing a file that contains key definitions critical to the system so making any additions should be done very carefully and should follow the same format as the other definitions in the file.

The `APPLICATION_TOKEN_HEADER` 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
```

Note: The header files do not have #ifndef HEADER_FILE / #define HEADER_FILE sequence at the top. This is important because this header file is included several times for different purposes.

7.1 Mechanics

Adding a custom 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 following illustrates how to define the three previous token examples.

7.1.1 Define the Token Name

When defining the name, do not prepend the word TOKEN. Instead, 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)
```

This defines the token key and links 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 is a 16-bit value that must be unique within this device. The first-bit is reserved for manufacturing and stack tokens, so all custom tokens should have a token key less than 0x8000.

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.

7.1.2 Define the Token Type

Each token in this example 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 at the start of section 7. 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
```

7.1.3 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 built-in, 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 (`installation_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.

The next section discusses default tokens, including stack tokens and manufacturing tokens.

7.2 Default Tokens

The networking stack contains some default tokens that may be useful for the application developer. These tokens come in two flavors:

Stack Tokens are runtime configuration options set by the stack; these should not be changed by the application.

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 manufacturing tokens for your chip, refer to one of the follow files:

```
<install-dir>/hal/micro/xap2b/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 EM35x SoC Platform*.

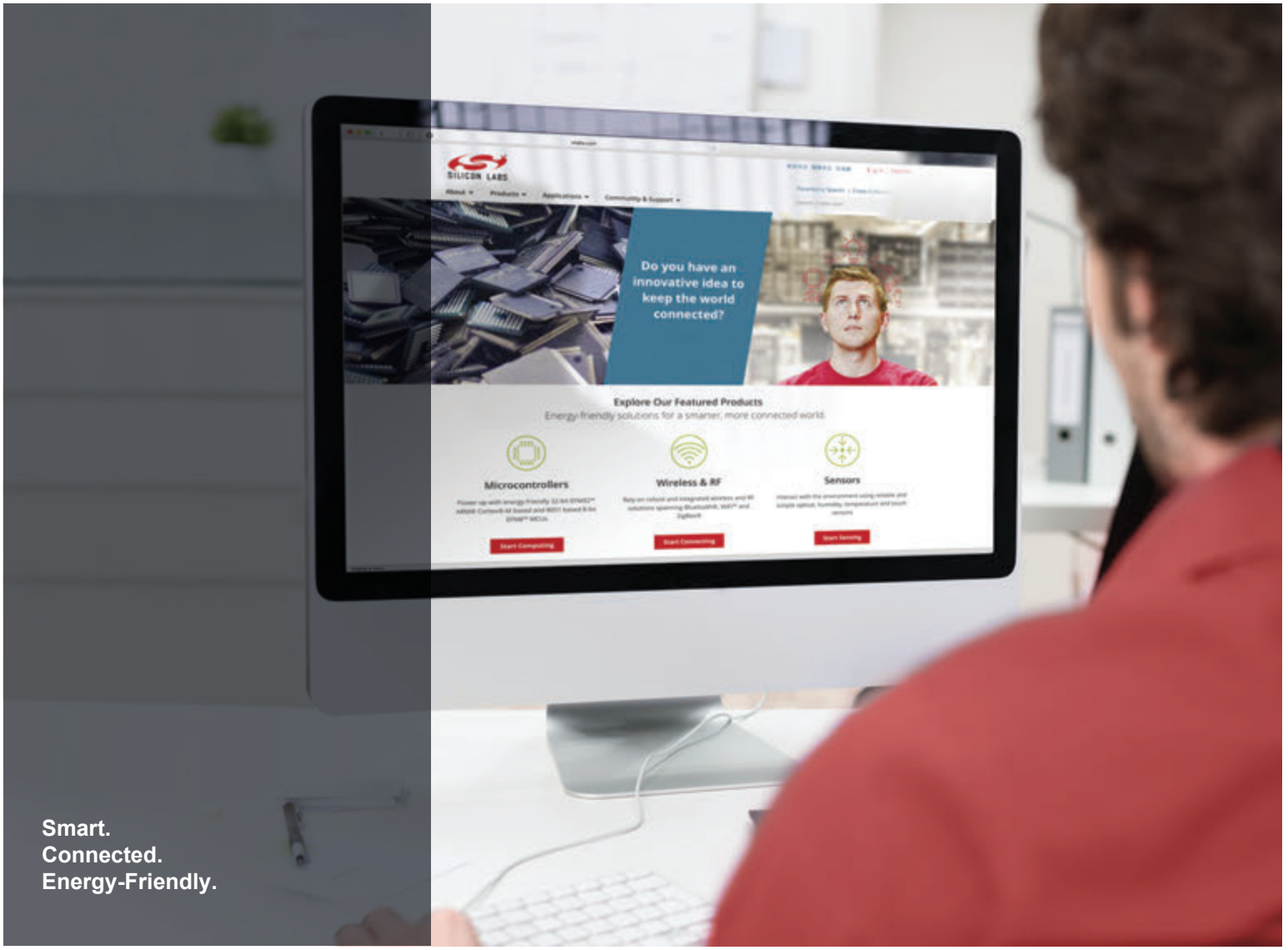
8. EmberZNet PRO Bindings

In the ZigBee PRO protocol, bindings are application-defined associations between two devices on the network. A binding may be used when sending to any message destination (multicast group or unicast point). They are used to provide a persistent means to store a pairing of two devices and the clusters, endpoints, and destinations they are using. The binding table consumes 2 bytes of RAM and 13 bytes of flash for each entry. Parts of this table reside in RAM, but most of the data is stored in token data. Therefore, the number of available bindings is limited by the processor's available RAM and EEPROM. It is important to understand that your application is responsible for managing binding table entries. For detailed information on `EMBER_BINDING_TABLE_SIZE` and `TOKEN_BINDING_TABLE`, refer to the EmberZNet PRO API reference.

The `EMBER_BINDING_TABLE_SIZE` constant specifies the maximum number of bindings supported by the stack. This includes the bindings in EEPROM and in RAM. It is set in stack/include/ember-configuration-defaults.h, and can be reset through preprocessor definitions at compile time.

9. For More Information

For more detailed information about the underlying token mechanism and the simulated EEPROM that provides the EM3x NVRAM implementation, please refer to the stack API reference installed with your software, which has detailed information about the use of these items.



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