AN961: Bringing Up Custom Nodes for the Mighty Gecko and Flex Gecko Families

This application note describes how to initialize a piece of custom hardware (a “device”) based on the Mighty Gecko and Flex Gecko families so that it interfaces correctly with a network stack. The same procedures can be used to restore devices whose settings have been corrupted or erased.

KEY FEATURES

- Information required before board bring-up
- Setting manufacturing tokens
- Bootloading
- Performing functional testing using Node-Test
- Setting stack tokens

1 Introduction

Before an EFR32-based product can be initialized and tested, manufacturing tokens within the EFR32 User Data Page and the Lock Bits Page must be configured. Similarly, before any application specific code can be programmed into the EFR32 flash, a board header needs to be created either manually or from the Application Builder tool set. In order to perform these tasks, the product design team must know how the EFR32 is to be used in the wireless system.

In particular, the design team must know the following:

- PCB Manufacturing-specific information (serial number, product numbers, EUI, and so on)
- Bootloader architecture (serial dataflash)
- External components in the RF Path (PA, LNA, and so on)
- 38.4 MHz crystal oscillator specification and a CTUNE value to match your crystal so you hit the center frequency
- Application security tokens (Keys, certificates, and so on)

Note: Even though the EFR32 flash is fully tested during production test, the flash contents in the main flash block are not set to a known state before shipment. The User Data and Lock Bits pages are erased to all 0xFF, except in kits where they might be preloaded with configuration values such as CTUNE.

2 EFR32 Wireless System

Once the hardware design of a custom device has been completed, the assembled product is ready for test and functional validation. Before testing, developers must understand how the device will operate at both the device-level and system-level. Table 1 describes the different items that developers should know before board bring-up.

Table 1. Information Needed Before Board Bring-Up

Information	Required or Optional	Description
Device Level		
Product Information	Optional	Most products have unique serial numbers as well as generic product codes that might be stored in the EFR32. This information might include items such as where and when the device was assembled, a product serial number, and product name.
Custom EUI	Optional	IEEE 64-bit unique number. Each EFR32 comes with an EUI programmed into the Device Information Block. Customers that have their own IEEE Block can use it in place of the Device Information Block's EUI.
RF Component Information	Required	If the product uses external PAs or LNAs, then developers must know the pin-connections between the off-chip components and the EFR32. They should also understand the LNA Gain as it will be used to offset the clear channel assessment (CCA) threshold.
ZigBee-Specific Information	Optional	Developers must know the ZigBee-assigned manufacturer code before testing.
System Level		
Bootloader Option	Required	Silicon Labs offers several bootloading options for different system designs. For more information, see *UG103.6, Application Development Fundamentals: Bootloading.*
System Security	Required	ZigBee profiles define specific security protocols for a device to follow.

As detailed in Table 1, Silicon Labs offers its customers an opportunity to guarantee a device's uniqueness on the network. In addition, it allows customers a way to store product descriptions, manufacturer-specific information and device information.

3 Setting Manufacturing Tokens

EFR32 chips are delivered to customers with all memory erased. Exceptions to this rule are chips that come with kits. Before the EFR32 chips can be used to run applications for a networking stack, the customer or a contract manufacturer/test house must prepare them. Preparation includes programming the proper application and bootloader, if necessary, into the Main flash block, as well as programming customer manufacturing tokens in the User Data block and Lock Bits block.

Manufacturing tokens are values programmed into special, non-volatile storage area of flash. The User Data page and Lock Bits page contain data that manufacturers of EFR32-based devices can program. The Device Information Block also contains manufacturing tokens, but these tokens are fixed values that cannot be modified.

Note: Applications and the stack can read any manufacturing tokens at any time.

Simplicity Commander is a single, all-purpose tool to be used in a production environment. It is invoked using a simple Command-Line Interface (CLI) that is also scriptable. Simplicity Commander enables customers to complete these essential tasks:

- Flash their own applications.
- Configure their own applications.
- Create binaries for production.

For more information, refer to *UG162, Simplicity Commander Reference Guide*.

Table 2 identifies the User Data manufacturing tokens for EmberZNet PRO that OEMs and CMs may want to program at manufacturing time. Refer to `\hal\micro\cortexm3\efm32\token-manufacturing.h` for the token definition, because it may differ from Table 2 and Table 3 depending on the stack release version.

Silicon Labs recommends that User Data and Lock Bits page tokens be written using Simplicity Commander at the same time as programming the main flash. Simplicity Commander also allows for patching and reprogramming the manufacturing blocks as many times as necessary. Some situations though, may require that a manufacturing token be programmed at runtime from code running on the chip itself. The manufacturing token module of the HAL provides a token API to write the manufacturing tokens. However, this API only writes manufacturing tokens that are in a completely erased state. If a manufacturing token must be reprogrammed, you must use an external utility. The API on SoC platforms is `halCommonSetMfgToken( token, data ).`The parameters for this API are the same as the API `halCommonSetToken( token, data )`.

Table 2 describes the location of each manufacturing token as an offset to the starting address of the relevant block. For the most accurate and specific information about where the flash regions begin in the address map of your chip, please consult your IC’s Reference Manual or Data Sheet.

Table 2. User Data Manufacturing tokens for the EFR32

Offset from User Data starting address	Size (Bytes)	Name	Description
0x0000	2	`TOKEN_MFG_CUSTOM_VERSION`	Version number to signify which revision of User Data manufacturing tokens you are using. This value should match `CURRENT_MFG_CUSTOM_VERSION` which is currently set to `0x01FE`. `CURRENT_MFG_CUSTOM_VERSION` is defined in `\hal\micro\cortexm3\efm32\token-manufacturing.h`. *Usage*: Recommended for all devices using User Data manufacturing tokens.
0x0002	8	`TOKEN_MFG_CUSTOM_EUI_64`	IEEE 64-bit address, unique for each radio. Entered and stored in little-endian. Setting a value here overrides the Silicon Labs EUI64 stored in the Device Information Page. This is for customers who have purchased their own address block from IEEE. *Usage:* Optionally set by device manufacturer if using custom EUI64 address block.

Offset from User Data starting address	Size (Bytes)	Name	Description
0x001A	16	`TOKEN_MFG_STRING`	Optional device-specific string, for example, the serial number. *Usage:* Optionally set by device manufacturer to identify device.
0x002A	16	`TOKEN_MFG_BOARD_NAME`	Optional string identifying the board name or hardware model. *Usage:* Optionally set by device manufacturer to identify device.
0x003A	2	`TOKEN_MFG_MANUF_ID`	16-bit ID denoting the manufacturer of this device. Silicon Labs recommends setting this value to match your ZigBee-assigned manufacturer code, such as in the stack's `emberSetManufacturerCode()` API call. *Usage:* Recommended for devices utilizing the stand-alone bootloader.
0x003C	2	`TOKEN_MFG_PHY_CONFIG`	Reserved for future use; should be left unprogrammed (0xFFFF).
0x003E	40	`TOKEN_MFG_ASH_CONFIG`	ASH configuration information.
0x00F0	2	`TOKEN_MFG_SYNTH_FREQ_OFFSET`	Reserved for future use; should be left unprogrammed (0xFFFF). Radio synthesizer frequency adjustments should be made using the `TOKEN_MFG_CTUNE` token instead.
0x00F6	2	`TOKEN_MFG_CCA_THRESHOLD`	Threshold used for energy detection clear channel assessment (CCA). • Bits 0-7: Set to the two's complement representation of the CCA threshold in dBm below which the channel will be considered clear. Valid values are -128 through +126, inclusive. +127 is NOT valid and must not be used. • Bit 8: Set to 0 if the threshold is valid and should be used. Set to 1 (the erased state) if the token is invalid or has not been set; in this case the default threshold will be used. • Bits 9-15: Reserved. Must be set to 1 (the erased state). The default CCA threshold for EFR32 devices is -75 dBm. If bit 8 of this token is 1 then -75 dBm will be used. You may want to override the default CCA threshold by setting this token if your design uses an LNA. An LNA changes the gain on the radio input, which results in the radio "seeing" a different energy level than if no LNA was used. Example: A design uses an LNA that provides gain of +12 dBm. Add the default -75 dBm gain to the LNA's gain to get the dBm value for the token: -75 + 12 = -63 dBm. The two's complement signed representation of -63 dBm is 0xC1, and so the complete token value to be programmed is 0xFEC1.
0x00F8	8	`TOKEN_MFG_EZSP_STORAGE`	An 8-byte, general-purpose token that can be set at manufacturing time and read by the host microcontroller via EZSP's `getMfgToken` command frame. *Usage:* Not required. Device manufacturer may populate or leave empty as desired.
0x0100	2	`TOKEN_MFG_CTUNE`	This token.is for tuning the EFR32 system XTAL and consequently also tunes the radio synthesizer frequency.
0x0102	2	`TOKEN_MFG_XO_TUNE`	Reserved for future use; should be left unprogrammed (0xFFFF).

Table 3. Lock Bits Data Manufacturing tokens for the EFR32

Offset from Lock Bits starting address	Size (Bytes)	Name	Description
0x0204	16	`TOKEN_MFG_BOOTLOAD_AES_KEY`	Sets the AES key used by the bootloader utility to authenticate bootloader launch requests of the Stand-alone bootloader. *Usage:* Required for devices that utilize the Stand-alone bootloader. This is not used by the application bootloader.
0x0214	92	`TOKEN_MFG_CBKE_DATA`	Defines the security data necessary for Smart Energy devices. It is used for Certificate Based Key Exchange to authenticate a device on a Smart Energy network. The first 48 bytes are the device's implicit certificate, the next 22 bytes are the Root Certificate Authority's Public Key, the next 21 bytes are the device's private key (the other half of the public/private key pair stored in the certificate), and the last byte is a flags field. The flags field should be set to 0x00 to indicate that the security data is initialized. *Usage:* Required by Smart Energy Profile certified devices.
0x0270	20	`TOKEN_MFG_INSTALLATION_CODE`	Defines the installation code for Smart Energy devices. The installation code is used to create a pre-configured link key for initially joining a Smart Energy network. The first 2 bytes are a flags field, the next 16 bytes are the installation code, and the last 2 bytes are a CRC. Valid installation code sizes are 6, 8, 12, or 16 bytes in length. All unused bytes should be 0xFF. The flags field should be set as follows depending on the size of the install code: • 6 bytes = 0x0000 • 8 bytes = 0x0002 • 12 bytes = 0x0004 • 16 bytes = 0x0006 *Usage*: Required by Smart Energy Profile certified devices.
0x0284	2	`TOKEN_MFG_SECURITY_CONFIG`	Defines the security policy for application calls into the stack to retrieve key values. The API calls `emberGetKey()` and `emberGetKeyTableEntry()` are affected by this setting. This prevents a running application from reading the actual encryption key values from the stack. This token may also be set at runtime with `emberSetMfgSecurityConfig()` (see that API for more information). The stack utilizes the `emberGetMfgSecurityConfig()` to determine the current security policy for encryption keys. The token values are mapped to the EmberKeySettings stack data type (defined in ember-types.h). See Table 5 below for the mapping of token values to the stack values.
0x0286	16	`TOKEN_MFG_SECURE_BOOTLOADER_KEY`	This token holds the 128 bit key used by the secure bootloader to decrypt encrypted EBL files. A value of all F's is considered an invalid key and will not be used by the secure bootloader.

Offset from Lock Bits starting address	Size (Bytes)	Name	Description
0x0296	148	`TOKEN_MFG_CBKE_283K1_DATA`	Defines the security data necessary for Smart Energy 1.2 devices using the ECC 283k1 curve. It is used for Certificate Based Key Exchange to authenticate a device on a Smart Energy network. The first 74 bytes are the device's implicit certificate, the next 37 bytes are the Root Certificate Authority's Public Key, the next 36 bytes are the device's private key (the other half of the public/private key pair stored in the certificate), and the last byte is a flags field. The flags field should be set to 0x00 to indicate that the security data is initialized.

Table 4. Mapping of EmberKeySettings to `TOKEN_MFG_SECURITY_CONFIG`

EmberKeySettings Value	TOKEN_MFG_SECURITY_CONFIG Value
`EMBER_KEY_PERMISSIONS_NONE`	0x0000
`EMBER_KEY_PERMISSIONS_READING_ALLOWED`	0x00FF
`EMBER_KEY_PERMISSIONS_HASHING_ALLOWED`	0xFF00

For more information on the Smart Energy tokens, see document AN708, *Setting Manufacturing Certificates and Installation Codes*.

4 Running the NodeTest Application

4.1 About NodeTest

NodeTest is a pre-built application supplied by Silicon Labs for the purpose of performing RF functional testing and hardware validation on development boards or custom-designed hardware. It contains RF test functions pertinent to IEEE 802.15.4-based radio configurations such as those used by the Silicon Labs ZigBee and Thread networking stacks on the EFR32MG. Bluetooth Low Energy (BLE) and other non-802.15.4-based modulation schemes are not supported by NodeTest. RF testing of other modulation schemes requires a different application or mechanism, such as BLE's Direct Test Mode (DTM) or a test application based on Silicon Labs Radio Abstraction Interface Layer (RAIL) libraries.

The majority of NodeTest's RF test functionality is also available through the Manufacturing Test Library API ("mfglib") exposed in EmberZNet and Thread through the mfglib library (for SOC platforms) or mfglib serial commands (for NCP platforms). As a result, custom-built applications can incorporate this functionality natively in an application designed specifically for their board configuration without relying on the pre-built NodeTest firmware.

The NodeTest application is included in the EmberZNet PRO and Thread stack installation directories, both of which contain a set of /build/nodetest-*xxx* subdirectories for a given processor/architecture variant, *xxx*. These directories contain.s37 and ebl file images that can be installed onto a chip through the normal, SerialWire- or JTAG-based firmware loading procedure.

Note: These file images are built to expect a bootloader in the main flash.

4.2 Uploading and Running NodeTest

NodeTest can be installed using either Simplicity Studio to upload the application or using a Command Line Interface (CLI) with Simplicity Commander. The following is an example of loading NodeTest and an accompanying bootloader using the Simplicity Commander CLI:

```
$ commander flash nodetest.ebl ..\..\tool\bootloader-efr32mg1p132f256gm48\serial-uart-bootloader\
serial-uart-bootloader.s37
```

Replace the `bootloader-efr32mg1p132f256gm48` string with the proper `bootloader-xxx` subdirectory for your variant and bootloader choice.

The command above flashes the NodeTest firmware and serial UART bootloader firmware to the USB-connected target device during a single process.

Note: Additional `--serialno` or `--ip` arguments can be added to the command to specify a particular target WSTK. For more information on how to program chips using Simplicity Commander from a command line, refer to *UG162: Simplicity Commander Reference Guide*.

Once the upload completes, you can interact with NodeTest via the hardware UART or the VCOM port. Using the VCOM port is done by accessing Simplicity Studio's **[Launch Console]** action from the Adapters view for your connected WSTK and attach to the **[Serial 1]** tab to interact with the VCOM port.

Press **Enter** in the Console window or serial terminal window to start the NodeTest application.

Note: If you are using the hardware UART directly rather than via the VCOM interface through the WSTK, you must configure your serial terminal program to 115,200 bps, no parity bits, 1 stop bit.

Note: Every time NodeTest resets, it will not automatically print any characters. Instead, NodeTest will listen on both the hardware UART and the VCOM port looking for a carriage return (Enter key). Whichever port NodeTest finds a carriage return on first will become the port that NodeTest uses for all serial interaction until NodeTest resets again.

Tip: If you don't see serial output indicating that NodeTest is starting up after having reset the device and pressed a carriage return on one of the serial ports, check that you've uploaded a bootloader to the device and haven't erased the main flash block since that upload was performed.

4.3 NodeTest Commands

A Return key initiates the NodeTest application on reset. This displays the power-up prompt, ending in the > (greater than) symbol. Pressing the Return key at the prompt will always cause another prompt to be displayed. Executing the **help** command causes NodeTest to print a list of all available commands with a brief description of the commands. The commands are listed by functional modules.

5 Performing Functional Testing

At this point, you may want to use the NodeTest application to perform a simple send/receive test on the device to determine its range and generally test its radio functionality.

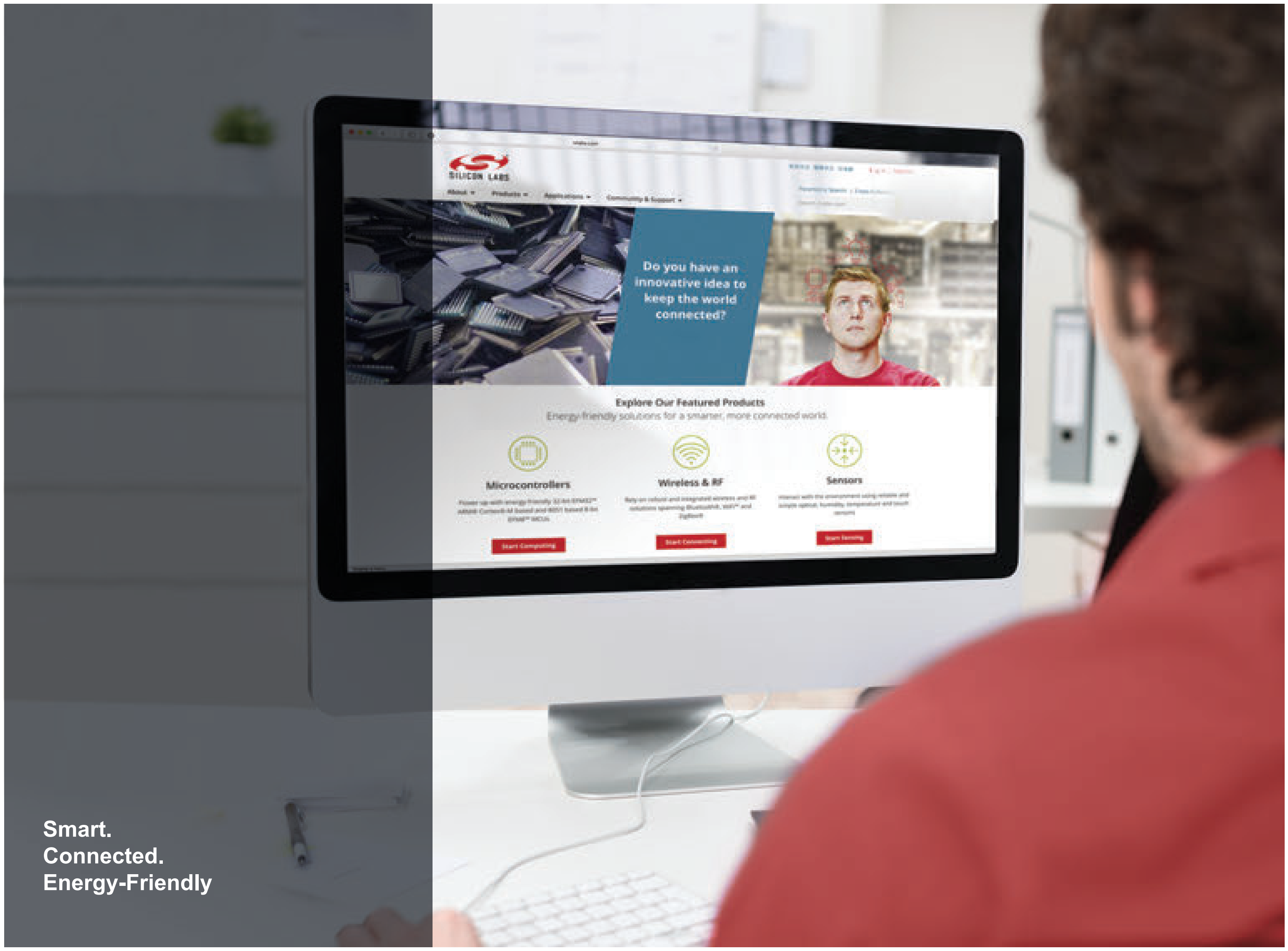
Note: When programming a device for test or retest, use the `--erase` option to ensure that all previous calibration data is erased. Silicon Labs recommends erasing the flash contents of a device prior to testing to ensure the calibration is executed at the time the device is tested.

1. Connect a device known to be in good operating order either to your computer or to a different computer through a serial port.
2. Upload NodeTest to the known good device and then run it.
3. Make sure that NodeTest is still running on the new device (you should see the > prompt).
4. Set both devices to a channel by typing `setchannel X`, where X is the channel in hex.
5. Optional: Set a power level on the test device by typing `settxpower X`, where X is the power level in hex.
6. On the known good device, type `rx`, which sets the device to receive and display statistics for each packet received. Type `e` to exit.
7. On the test device, type `tx X` to transmit X packets where X is in hex. (Note that `tx 0` sends infinite packets.) Type `e` to exit.
8. Reverse this procedure to test receiving on the test device.

Note: The fourth column in the display output, labeled "per", shows the packet error rate. For this value to be accurate, the two devices being used must be configured per section 4.2, **Uploading and Running NodeTest** and the receiver should not hear any other devices. Exiting the test and restarting clears the values and reset the values being displayed.

Note: NodeTest attempts to print packet data as fast as it can, but it is possible to receive packets faster than NodeTest can print. Therefore, there may be gaps in the printed packets.

If the new device fails to successfully transmit or receive packets with the known good device, you may want to attach the new device to a signal generator or network analyzer to verify that generated packets on the target frequency can be received and that the new device can transmit accurately at the center frequency of the selected channel. Other tests may be required for FCC or CE compliance testing.


Smart.
Connected.
Energy-Friendly
Do you have an innovative idea to keep the world connected?
Explore Our Featured Products
Microcontrollers
Wireless & RF
Sensors

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