AN1010: Building a Customized NCP Application

The ability to build a customized NCP application image was introduced in EmberZNet PRO 5.4.1 and Silicon Labs Thread 1.0.6. This application note provides instructions for configuring various aspects of the application using the Application Builder tool within Simplicity Studio.

KEY POINTS

- Instructions cover starting from an example or from a new file.
- Customizations include target hardware, initialization, main loop processing, event definition and handling, and host/NCP command extensions.

1. Introduction

In previous stack releases, developers were restricted to using the preconfigured NCP application images ncp-spi and ncp-uart, delivered with the Silicon Labs Thread stack, and similar precompiled binaries delivered in pre-5.4.1 versions of the EmberZNet PRO stack. Both stacks now support the ability to build NCP applications in the Simplicity Studio IDE (also known as AppBuilder), with customizations for target hardware, initialization, main loop processing, event definition and handling, and host/NCP command extensions. If you are not familiar with using Simplicity Studio to configure, build, and load applications, refer to *QSG113: Getting Started with Silicon Labs Thread*, and *QSG106: Getting Started with EmberZNet PRO*.

This application note describes how to configure a customized NCP application using the Silicon Labs Thread or EmberZNet PRO stack for either the EM35x or members of the Wireless Gecko (EFR32) platforms.

2. Silicon Labs Thread

The Silicon Labs Thread stack includes example applications that work over either SPI or UART. These can be used as starting point for building a customized NCP application, or you can start from a blank application. The first two sections describe these two approaches. The third section details the customizations you can make after starting your application in one of the two ways described.

2.1 Starting from an Example Application

Silicon Labs recommends choosing one of the Silicon Labs Thread example applications as a starting point for building a customized NCP application. Three examples are provided: NCP-SPI, NCP-UART (with software flow control enabled) and NCP-UART-HW (with hardware flow control enabled). As configured, these NCP example applications are used to build the corresponding binary images delivered with the Silicon Labs Thread stack. Therefore, you not only have a configuration starting point, but can also easily evaluate changes in functionality from your customizations against the binary images provided.

When you start a new project in AppBuilder, you can select the target part for which the application is being built. The Silicon Labs Thread NCP example applications are preconfigured with the EM3xx Device Type, shown in the Device Type box on the General tab. For these examples, there is no material functional difference between EM3xx and EFR32 device types. The device type selections merely choose a set of included default plugins for the build. As these currently are equivalent between the two device types, you can safely leave this setting unchanged.

2.2 Starting from a Blank Application

If, instead of beginning with one of the example applications, you prefer to begin with a blank application, make the following changes under the indicated AppBuilder project tabs to configure basic NCP functionality. Following these changes, other customizations can be made as described in section 2.3 Customizations.

General Tab

1. Click the Device type box to open it.
2. Select and add either EM3xx or EFR32, according to selected part.
3. Select and add NCP.
4. Select and add either NCP SPI or NCP UART, according to host/NCP link type.

HAL Configuration Tab

Set **Bootloader** as Standalone.

Printing Tab

In Command Line Configuration, uncheck the **Enable command line interface** checkbox. (Interaction with the NCP firmware will be via TMSP binary protocol with the host processor.)

2.3 Customizations

Some of these customizations involve enabling callbacks. Select a callback in AppBuilder to see detail about the callback.

Board Header Configuration

If using custom hardware, go to the HAL configuration tab and, under **Board Header**, specify the path to a board header file suitable for that hardware. See the board header files provided for supported development kits for examples. Modify the HAL Configuration plugin in the HAL plugin group to configure custom peripheral interfaces.

Custom Code Implementation Files

Source files, include paths, and libraries that implement custom capabilities should be registered in the corresponding configuration table found on the Other tab. These files will contain implementation for callbacks, event handlers, and messaging customizations.

NCP Initialization

Enable and implement either or both of the following initialization callbacks if needed.

`emberAfMainCallback` - Initialization immediately following NCP reset.

`emberAfInitCallback` - Initialization following Silicon Labs Thread stack initialization.

NCP Main Loop Processing

Enable and implement the following callback if needed.

`emberAfTickCallback` - Custom processing per iteration of NCP main processing loop.

NCP Flow Control

The default configuration is hardware flow control. To turn on software flow control, on the Plugins tab under HAL, select the NCP UART Link plugin and check Use software flow control. Note that you may have to scroll over to the left and all the way up to see the plugins parameters.

Note: Software flow control cannot be used through the WSTK's USB-to-serial interface; it requires use of the expansion header (EXP port) on the WSTK to access the UART signals.

NCP Event Definition and Handling

Register custom event controls and handlers in the corresponding table on the AppBuilder project Other tab. Implement event triggers and handlers in the custom code implementation files.

Custom Messaging

To implement custom messages between NCP and host, the developer defines and implements the format, parsing, and serialization of the message set. The serialized messages are conveyed between NCP and host as opaque byte strings.

To send a custom message to the host, construct and serialize the message, then send the resulting byte string to the host using the Silicon Labs Thread API function `emberCustomNcpToHostMessage(const uint8_t* message, uint8_t messageLength)`.

To process a custom message received from the host, enable and implement the following handler (callback).

`emberCustomHostToNcpMessageHandler` - Process byte string received from host.

Similarly, to send/receive custom messages to/from the NCP, a compatible host application would use the complementary pair:

`emberCustomHostToNcpMessage`

`emberCustomNcpToHostMessageHandler`

3. EmberZNet PRO

The EmberZNet PRO stack includes an example application that can be configured to work over either SPI or UART. This can be used as starting point for building a customized NCP application, or you can start from a blank application. The first two sections describe these two approaches. The third section details the customizations you can make after starting your application in one of the two ways described.

Note: Both approaches use AppBuilder, an interactive GUI tool that allows you to configure a body of Silicon Labs-supplied code to implement applications. New customers should access AppBuilder through Simplicity Studio. Simplicity Studio should also work with stack versions 5.4.x and 5.6.0. Otherwise, for stack versions earlier than 5.7.0, use Ember Desktop. Make sure that the EmberZNet stack you are using has been added to AppBuilder's list of included stacks, as described in *QSG106: Getting Started with EmberZNet PRO*.

3.1 Starting from an Example Application

Silicon Labs recommends that you choose the EmberZNet PRO xNCP example application as a starting point for building a customized NCP application. The xNCP LED example illustrates how to extend EZSP with a new custom command that allows the host to configure a blinking LED on the NCP (and pairs with the XNCP_Led Host app in the EmberZNet Host framework).

When you start a new project in Simplicity Studio, you can select the target part for which the application is being built. This should correspond to the selection in the Architecture picklist in the General tab of AppBuilder. Regardless of which part you select (EM3xx or EFR32), the NCP sample application is preconfigured with the EM3xx Device Type, shown in the Device Type box on the General tab. For NCP application purposes, there is no material functional difference between EM3xx and EFR32 device types. If building an NCP application for an EFR32 part starting from one of the NCP sample applications, leave the Device Type setting of EM3xx unchanged. You then only need to make the following changes on the Plugins tab:

- In the Core section, check one, and only one, of the three options (**NCP - SPI**, **NCP - UART**, or **NCP - USB**).The latter is only available for parts that have native USB serial controller functionality and is not recommended for new designs. If selecting **NCP - UART**, select the appropriate **Flow Control Type** option from the picklist in the plugin options area so that it matches the UART flow control style used by your corresponding host application.
- In the HAL section, for Simulated EEPROM Version 2 Library plugin, enter the password for this plugin as described in *AN703: Using the Simulated EEPROM for the EM35x and Mighty Gecko SoC Platforms*, as consent that you understand the implications of choosing SimEEPROM v2.
- If you want to have more than 32 end-device children, on the Plugins tab In the EmberZNet Libraries section, ZigBee PRO Stack Library, change the Child Table Size parameter to the max desired number of end device children joined directly to the NCP. Note that, while the on-screen text says the value range is 0-127, you cannot build the app if you enter a value greater than 64. The precompiled images are limited to 32 children.

3.2 Starting from a Blank Application

If, instead of beginning with one of the example applications, you prefer to begin with a blank application, follow these instructions.

1. Start a new AppBuilder project.
2. Choose **Customizable network coprocessor (NCP) applications** and click **Next**.
3. Check **Start with a blank application** and click **Next.**
4. Name your project and click **Next**.
5. If prompted for Kit and Part, select the appropriate kit for your hardware (EM35x or EFR32).
6. Choose the proper IC variant for your target, then click **Finish**.
7. Modify settings on the AppBuilder tabs as described below,
8. After you have modified the tab settings, click **Generate** and check for successful generation of your files.
9. After generating, right-click on the project in the Project Explorer pane of Simplicity Studio (or open the generated IAR EWW workspace in IAR EWARM separately if not using the Studio IDE) and select **Build Project**.
10. After the EBL file is built, load the EBL file to your target NCP using Simplicity Studio (or Ember Desktop) along with the appropriate ezsp-spi-bootloader or serial-uart-bootloader binary from the tool/bootloader-{mcuName}/{bootloaderName} folder of your EmberZNet PRO installation.

General Tab

(optional) Change the Device Name field to match what you called your project, so that the resulting binary will have a more distinct name than “ncp.ebl”.

Verify that **Selected architecture for this application** matches the part you are targeting.

For Device Type:

1. Click **No device type selected.**
2. Select **EM3xx NCP** from the list on the left, even if your target is an EFR32 NCP. The device type selections merely choose a set of included default plugins for the build. As these currently are equivalent between the two device types, you can safely leave this setting unchanged.

   Note: When you select the “em3xx NCP” device type, subsequent tabs will be configured with almost everything you need as a starting point, except for the choice of SPI/UART/USB plugin as appropriate based on the desired serial interface. After enabling one (and only one) of those three plugins, enter the password for the SimEEPROM v2 plugin as appropriate, and then you can safely generate.

3. Click **Add Device** →. Make sure the EM3xx NCP device type is added to the list on the right.
4. Click **OK**.

HAL Configuration Tab

Under **Board Header**, specify the board header file appropriate for your hardware.

Plugins Tab

- In the Core section, check one, and only one, of the three options (**NCP - SPI**, **NCP - UART**, or NCP - USB). .The latter is only available for parts that have native USB serial controller functionality and is not recommended for new designs. If selecting **NCP - UART**, select the appropriate **Flow Control Type** option from the picklist in the plugin options area so that it matches the UART flow control style used by your corresponding host application.
- In the HAL section, for Simulated EEPROM Version 2 Library plugin, enter the password for this plugin as described in *AN703: Using the Simulated EEPROM for the EM35x and Mighty Gecko SoC Platforms*, as consent that you understand the implications of choosing SimEEPROM v2.
- (optional) In the EmberZNet Libraries section, Binding Table Library, change the “Binding Table Size” parameter to the max desired binding table size used by the NCP.
- (optional) In the EmberZNet Libraries section, Security Link Keys Library, change the “Link Key Table Size” parameter to the max desired number of unique APS link keys used by the NCP. Note that if you are configuring your NCP to act as a Trust Center with ZigBee 3.0 Security (as set in the Security Type area on the Znet Stack tab), it is not necessary to have a unique key table entry for every device; instead, a single security key known as a Master Key will be used to compute unique keys via an AES-HMAC hash function for each device. However, supporting install-code-based keys requires a link key table with as many entries as the number of install-code-based keys you wish to support simultaneously for joining devices with install code support.
- (optional) In the EmberZNet Libraries section, ZigBee PRO Stack Library, change the Child Table Size parameter to the max desired number of end device children joined directly to the NCP. Note that, while the on-screen text says the value range is 0-127, you cannot build the app if you enter a value greater than 64. The precompiled images are limited to 32 children.
- (optional) In the EmberZNet Libraries section, ZigBee PRO Stack Library, increase/decrease other option parameters to meet your needs. You may need to reduce values like Packet Buffer Count (which has a high RAM overhead) if your build fails due to lack of

available RAM in the memory map. However, note that most memory-related parameters here simply represent defaults when the NCP boots, and these settings can be overridden by the host during run-time configuration when the NCP is initialized.

- (optional) In the Command Handlers section, disable any EZSP command handler plugins for command sets not needed on your NCP.
- (optional) In the EmberZNet Libraries section, change plugins for any stack features not needed on your NCP to stub variants.
- (optional) In the HAL section, if Activity LED is not desired on your NCP, change HAL: LED plugin to HAL: LED Stub plugin.

3.3 Customizations

Some of these customizations involve enabling callbacks. Select a callback in AppBuilder to see detail about the callback.

Custom Code Implementation Files

Source files, include paths, and libraries that implement custom capabilities should be registered in the corresponding configuration table found on the Other tab. These files will contain implementation for callbacks, event handlers, and messaging customizations.

NCP Initialization

Enable and implement the following initialization callback if needed.

`emberAfMainCallback` - Initialization immediately following NCP reset.

NCP Main Loop Processing

Enable and implement the following callback if needed.

`emberAfMainTickCallback` - Custom processing per iteration of NCP main processing loop.

Security

For devices implementing Trust Center functionality (either as a coordinator providing centralized trust center responsibilities for the network or a router in a decentralized trust center configuration), NCP developers may wish to override the EZSP Trust Center policy's decisions about when and how to provide the current network security key to a joining or rejoining device. In EmberZNet PRO releases beginning with version 5.7.1, the following callback may be used to provide this feature:

```
EmberJoinDecision emberAfPluginEzspSecurityTrustCenterJoinCallback(EmberNodeId newNodeId,
                                                                   const EmberEUI64 newNodeEui64,
                                                                   EmberDeviceUpdate status,
                                                                   EmberNodeId parentOfNewNode,
                                                                   EzspDecisionId decisionId,
                                                                   EmberJoinDecision joinDecision)
```

NCP Event Definition and Handling

Register custom event controls and handlers in the corresponding table on the AppBuilder project Other tab. Implement event triggers and handlers in the custom code implementation files.

Custom Messaging

To implement custom messages between NCP and host, the developer defines and implements the format, parsing, and serialization of the message set. The serialized messages are conveyed between NCP and host as opaque byte strings. This "extensible network coprocessor" functionality is provided by the "XNCP Library" plugin (as opposed to the XNCP Stub Library plugin) in the NCP Framework.

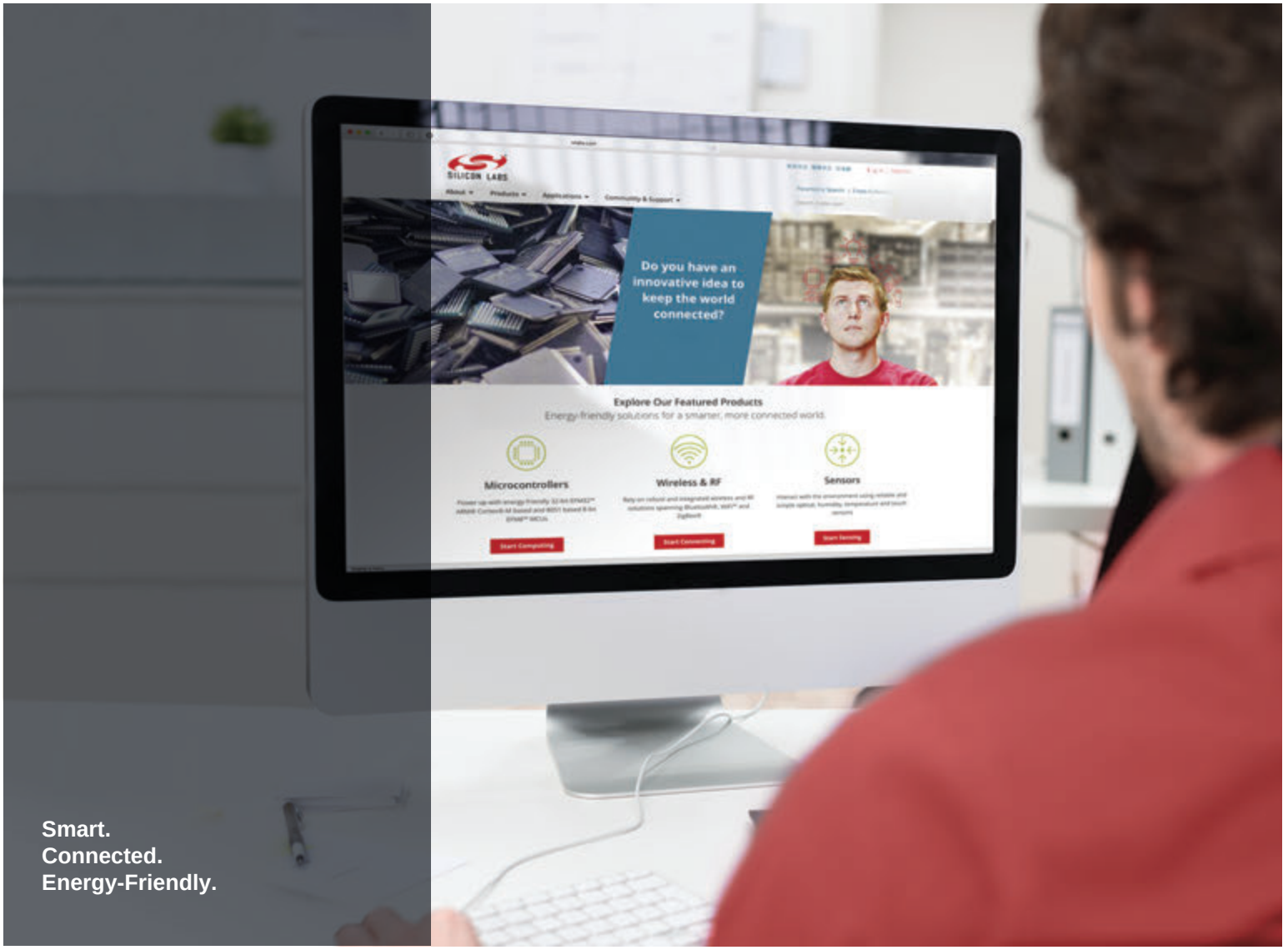
To send a custom message to the host, construct and serialize the message, then send the resulting byte string to the host using the EmberZNet PRO API function `emberAfPluginXncpSendCustomEzspMessage()`.

After enabling the XNCP Library plugin, the following callbacks are provided via the Callbacks tab for custom 2-way messaging via EZSP:

`emberAfPluginXncpIncomingCustomFrameCallback` - Processing of custom incoming serial frames from the EZSP host

`emberAfIncomingMessageCallback` - Custom processing of received ZigBee application layer messages prior to passing these (via Incoming Message Callback frames) to the EZSP host

Note that custom *outgoing* serial frames from the NCP to the EZSP host should be provided as response frames to the host in reply to a Callbacks EZSP command or some custom host-to-NCP EZSP command, where they can be handled by the following host-side callback: `void ezspCustomFrameHandler(int8u payloadLength, int8u* payload)`



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