UG435.08: Network Co-Processor Applications with Silicon Labs Connect v3.x

This chapter of the Connect v3.x User's Guide describes how to run the Silicon Labs Connect stack in Network Co-Processor (NCP) mode. The Connect stack is delivered as part of the Silicon Labs Proprietary Flex SDK v3.0 and higher. The Connect v3.x User's Guide assumes that you have already installed the Simplicity Studio development environment and the Flex SDK, and that you are familiar with the basics of configuring, compiling, and flashing Connect-based applications. Refer to UG435.01: About the Connect v3.x User's Guide for an overview of the chapters in the Connect v3.x User's Guide.

KEY POINTS

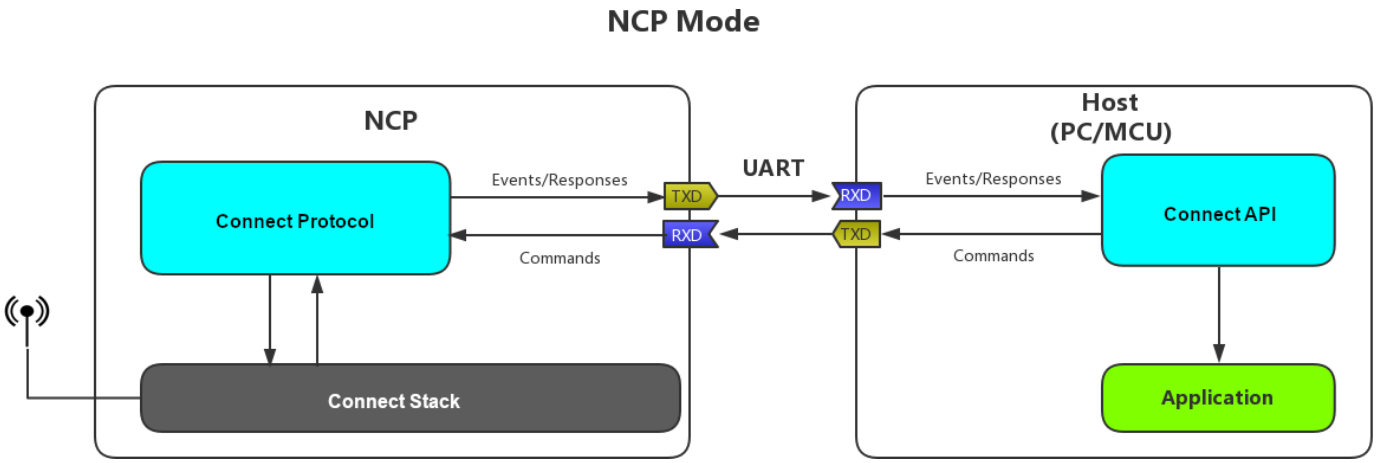
- Introduces the NCP.
- Discusses NCP, CPC, and Host applications.
- Explains the Connect Serial Protocol.

The Connect v3.x User's Guide is a series of documents that provides in-depth information for developers who are using the Silicon Labs Connect Stack for their application development. If you are new to Connect and the Proprietary Flex SDK, see QSG168: Proprietary Flex SDK v3.x Quick Start Guide.

Proprietary is supported on all EFR32FG devices. For others, check the device's data sheet under Ordering Information > Protocol Stack to see if Proprietary is supported. In Proprietary SDK version 2.7.n, Connect is not supported on EFR32xG22.

1 Introduction

NCP stands for Network Co-processor. By adding a Wireless Gecko (EFR32™) System on Chip (SoC) in NCP mode to your system, you can implement a Connect-based wireless application that leverages the EFR32 Radio feature set. The original customer application (running on a Host device—PC, MCU) interfaces to and controls the NCP through the Universal Asynchronous Receiver-Transmitter (UART) interface as shown in the following figure.



Note: NCP should not be confused with virtual Network Co-processor (vNCP). For more information on vNCP, see *AN1153: Developing Connect vNCP Applications with Micrium OS*.

Messages sent from the Host to the NCP are known as commands. Messages sent from the NCP to the Host are known as callbacks.

To carry commands and responses between a Host processor and an NCP, Connect uses a Co-Processor Communication (CPC) endpoint.

2 NCP, CPCd, and Host Applications

In the NCP approach, a typical Connect-based application is divided into three separate applications: NCP, Host, and the Co-processor Communication daemon (CPCd). The NCP runs on the EFR32, while the Host application and CPCd runs on the Host device.

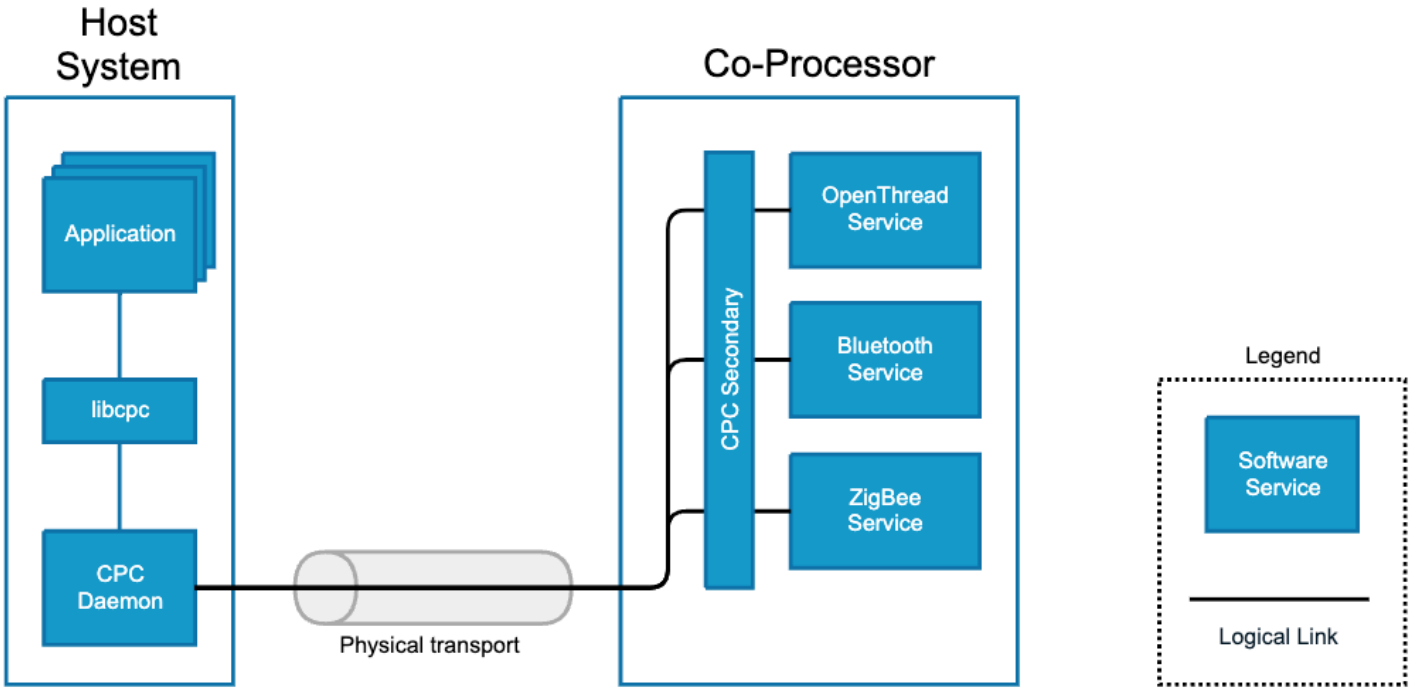
2.1 CPC Daemon

The CPC daemon provides a unified interface for various projects requiring co-processor communication. CPC calls the host device primary, and the NCP (EFR32) device secondary.

CPCd can be downloaded from github: https://github.com/SiliconLabs/cpc-daemon.

In the most common use case, CPCd is running as a daemon on Linux, connecting to the primary's application via the CPC library (libcpc.so) and connecting to the secondary's application on UART (other communication channels are supported as well).

CPC supports multiple endpoints, so it can be used to implement dynamic multiprotocol applications, or another endpoint can be created on the same device to implement something on the secondary that is not connected to the radio stack.



Modifying the CPCd is not recommended and should not be needed.

2.2 NCP Application (SoC)

The NCP application runs on the EFR32 and supports communication with CPCd over a CPC endpoint. The NCP application can be built with the default configuration, or optionally can be augmented with customizations for target hardware. The application provided is called *Connect – NCP*.

The NCP application can be used out-of-the-box without any modification. However, the radio configuration must be set in the NCP Application.

We do not recommend implementing new commands on the same endpoint, if you need further, not Connect related control of the NCP device, we recommend creating a separate CPC endpoint for the new commands.

2.3 Host Application

The Host application can be compiled to almost any device where libcpc is available. Currently, the Host application examples are implemented for the Linux operating system only. These applications are not available in Simplicity Studio, but available on github: https://github.com/SiliconLabs/Connect-NCP-Host.

Physical layer (PHY) parameters can be set only at compile time and only in the NCP application. Radio parameters are not configurable on the Host side.

3 Compiling NCP and Host Applications

3.1 Compiling the NCP Application

The procedure for compiling the NCP application is identical to that for any other Connect-based EFR32 application. Use the project configurator if any configuration of the "stock" examples is necessary (radio parameters, components, etc.).

3.2 Compiling CPCd and the Host application

Documentation on how to compile CPCd and the Host application can be found in the readme file of their github repositories:

- https://github.com/SiliconLabs/cpc-daemon
- https://github.com/SiliconLabs/Connect-NCP-Host

3.3 Modifying Host Application or Building One from Scratch

The Connect stack API available to a Host application is nearly identical to the API in an SoC application, with the exception of a few calls that make sense only on one or the other. Functions are invoked with the same arguments and generate the same return values, the same callbacks, and events govern stack behavior, etc. Therefore, when developing an NCP mode system, interaction with the Connect stack API by the Host application is largely identical to the development experience for an NCP application.

One important difference is that the host application doesn't save the security key. Since we don't have a uniform secure key storage on Linux, we leave it to the user to implement it according to their security requirements. The host application loads the default dummy key at boot (all 0xA) and can be changed via a CLI command.

We recommend implementing host applications by building on the connecthost library (which can be built from the previously mentioned github repository), and the APIs are implemented in `connect/ncp.h` and `connect/callback_dispatcher.h`. The host application should always start with calling `sl_connect_ncp_init()` which will initialize the connection to the secondary (i.e. the EFR32).

A host application should periodically call `sl_connect_poll_ncp_messages()` in a separate thread. This will store the data from the secondary (typically return values and callbacks) in a queue on the host side. To process the queue, the host should call `sl_connect_ncp_handle_pending_callback_commands()` from another thread, which will deserialize the message and call the callback functions.

For more details on the usage of the connecthost library, see the documentation on its github page.

4 Connect Serial Protocol

The Connect Serial Protocol (CSP) is used by a Host application processor to interact with the Connect stack running on an NCP. CSP messages are sent between the Host and the NCP over a CPC endpoint. CSP is a binary protocol encapsulated within the CPC protocol. For details on the CPC protocol, see the documentation of the CPCd github project.

Every API call from the application translates into a CSP message. The file `csp-command-app.c` contains the Host API implementation that converts API calls into serial commands, while `csp-command-callbacks.c` converts incoming serial commands into calls to the application callback functions. There is also a table that maps serial command IDs to handlers. All API calls are referenced by an identifier that must match on both the Host and the NCP side. The file `csp-api-enum-gen.h` contains these identifiers for both the host and the NCP project.

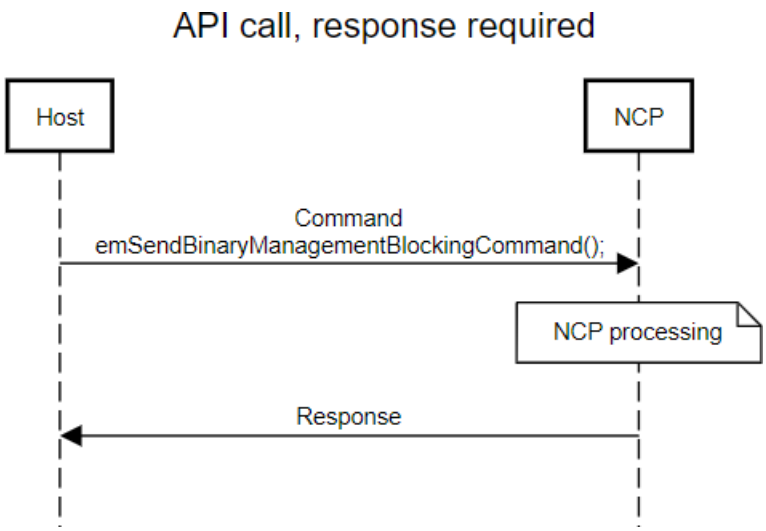
The `csp-command-vncp.c` is the NCP code that corresponds to `csp-command-app.c`. For every API in `csp-command-app.c`, a corresponding handler exists in `csp-command-vncp.c`. The `csp-command-callbacks.c` file exists both on host and NCP, but they are not the same. On the host side, `sli_connect_ncp_handle_indication()` should be called to deserialize CSP into callbacks, while on the NCP side, every supported callback is implemented and serialized into CSP frames.

Note: Bear in mind that the files `csp-command-app.c`, `csp-command-vncp.c`, `csp-command-callbacks.c`, and `csp-api-enum-gen.h` are automatically generated so they are subject to change without notice.

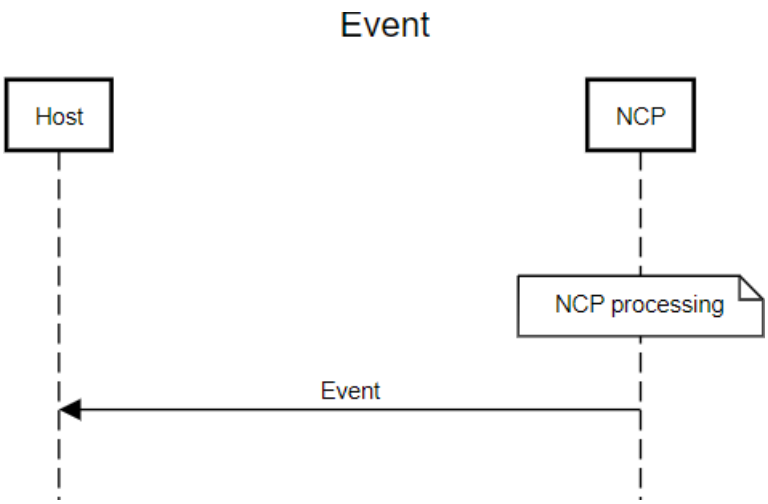
In general, the CSP works as follows:

1. An API call at the Host generates a serial message that gets sent to the NCP.
2. The Host spins waiting for a response from the NCP.
3. The NCP, upon receiving a message from the Host, decodes the message and invokes the actual stack API.
4. The NCP gets a return status from the API, packages it into a response, and sends it to the Host.
5. The Host, which was waiting for the response, gets the response, which is the return value of the API call.
6. Finally, the stack API call at the Host returns the actual return value. The following figure illustrates the typical CSP data flow.

The following figure illustrates the typical CSP data flow.



In the case when data is generated on the NCP side, e.g., by receiving from the network, the NCP sends the message, and the corresponding callback function is invoked on the Host side as shown in the following figure.



4.1 CSP Format

Byte 0	Byte 1	Byte 2	(Bytes > 2)
Command identifier		Arguments	

The CSP message consists of a 16-bit command identifier and the packed arguments. This is generated/decoded on both sides using `csp-format.c`.

5 Bootloading with NCP

5.1 Bootloading the NCP (secondary)

CPC supports bootloading the secondary via xmodem protocol, using CPCd as the programmer. For more details, follow the CPCd documentation.

5.2 OTA Image Distribution and Bootloading

Connect supports OTA Broadcast and Unicast implementations. For details on them, see *UG435.06: Bootloading and OTA with Silicon Labs Connect v3.x*.

In NCP mode, currently only the Unicast implementation is supported, and it differs slightly from the SoC implementation. It is usually the Host-NCP device pair that starts distributing the image. In this case, there is no reason to first load the image to the NCP's flash and then start the distribution. It's much simpler to directly distribute the image from the host's memory. This also means that you don't need Gecko Bootloader installed on the NCP, and you don't need to call `bootloader_init` or `erase` on the host.

To select a gbl file for distribution on the host, the `load_gbl_file` command can be used. Its argument is a filename, relative to the path from where the host application was started. It will return the file size, which can be used for the distribute command later on. The process of image distribution and bootloading request from this point on is the same.

Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support & Community
www.silabs.com/community

SILICON LABS

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

www.silabs.com