UG162: Simplicity Commander Reference Guide

This document describes how and when to use the Command-Line Interface (CLI) of Simplicity Commander. Simplicity Commander supports all EFR32 Wireless SoCs, EFR32 Wireless SoC modules (such as the MGM111 or MGM12P), EFM32 MCU families, and EM3xx Wireless SOCs. EFM8 MCU families are not supported at this time.

This document is intended for software engineers, hardware engineers, and release engineers. Silicon Labs recommends that you review this document to familiarize yourself with the CLI commands and their intended uses. You can refer to specific sections of this document to access operational information as needed. This document also includes examples so you can gain an understanding of Simplicity Commander in action.

This document is up-to-date with Simplicity Commander version 1.15. See section 7. Software Revision History for a list of new and modified commands for the current and previous versions of the application.

KEY POINTS

- Introduces Simplicity Commander.
- Adds new features and commands.
- Describes the file formats supported by Simplicity Commander.
- Includes detailed syntax of all Simplicity Commander commands and example command line inputs and outputs.

 Copyright © 2024 by Silicon Laboratories

Table of Contents

1. Introduction . . . 8

2. File Format Overview . . . 9

2.1 Motorola S-record (s37) File Format . . . 9

2.2 Update Image File Formats . . . 9

2.3 Intel HEX-32 File Format . . . 10

3. General Information . . . 11

3.1 Installing Simplicity Commander . . . 11

3.2 Command Line Syntax . . . 11

3.3 General Options . . . 12

3.3.1 Help (`--help`) . . . 12

3.3.2 Version (`--version`) . . . 14

3.3.3 Device (`--device <device name>`) . . . 14

3.3.4 J-Link Connection Options . . . 15

3.3.5 Debug Interface Configuration . . . 16

3.3.6 Graphical User Interface . . . 16

3.3.7 Timestamp (`--timestamp`) . . . 17

3.4 Output and Exit Status . . . 17

4. EFR32 Custom Tokens . . . 18

4.1 Introduction . . . 18

4.2 Custom Token Groups . . . 18

4.3 Creating Custom Token Groups . . . 18

4.4 Defining Tokens . . . 19

4.5 Memory Regions . . . 19

4.6 Token File Format Description . . . 20

4.7 Using Custom Token Files . . . 20

4.8 Using Custom Token Files in Any Location . . . 20

5. Security Overview . . . 21

5.1 Security Store . . . 21

5.2 Access Certificate . . . 21

5.3 Challenge and Command Signing . . . 22

6. Simplicity Commander Commands . . . 23

6.1 Device Flashing Commands . . . 23

6.1.1 Flash Image File . . . 24

6.1.2 Flash Using IP Address without Verification and Reset . . . 24

6.1.3 Flash Several Files . . . 25

6.1.4 Patch Flash . . . 26

6.1.5 Patch Using Input File . . . 27

6.1.6 Flash Tokens . . . 28

6.2 Flash Verification Command 29
6.3 Memory Read Commands 29
6.3.1 Print Flash Contents. 30
6.3.2 Dump Flash Contents to File 30
6.4 Token Commands 31
6.4.1 Print Tokens 31
6.4.2 Dump Tokens to File 31
6.4.3 Dump Tokens from Image File 32
6.4.4 Generate C Header Files from Token Groups 32
6.5 Convert and Modify File Commands 32
6.5.1 Combine Two Files 33
6.5.2 Define Specific Bytes 33
6.5.3 Define Tokens. 34
6.5.4 Dump File Contents 34
6.5.5 Signing an Application for Secure Boot 35
6.5.6 Signing an Application for Secure Boot using a Hardware Security Module 35
6.5.7 Signing an Application for Secure Boot Signing using a Signature Created by a Hardware Security Module 36
6.5.8 Adding a CRC32 for Gecko Bootloader 36
6.5.9 Signing an Application for Secure Boot using an Intermediary Certificate 37
6.5.10 Add a Trust Zone Decryption Key .38
6.5.11 Extract Sections from ELF Files .38
6.6 EBL Commands 39
6.6.1 Print EBL Information 39
6.6.2 EBL Key Generation 39
6.6.3 EBL File Creation 40
6.6.4 EBL File Parsing 40
6.6.5 Memory Usage Information from AAT 41
6.7 GBL Commands 41
6.7.1 GBL File Creation 41
6.7.2 GBL File Creation with Compression 42
6.7.3 Create a GBL File for Bootloader Upgrade 42
6.7.4 Creating a GBL File for Secure Element Upgrade 43
6.7.5 Creating a Signed and Encrypted GBL Upgrade Image File from an Application 43
6.7.6 Creating a Partial Signed and Encrypted GBL Upgrade File for Use with a Hardware Security Module 44
6.7.7 Creating a Signed GBL File Using a Hardware Security Module 45
6.7.8 GBL File Parsing 45
6.7.9 GBL Key Generation 45
6.7.10 Generating a Signing Key 45
6.7.11 Generate a Signing Key Using a Hardware Security Module 46
6.7.12 Creating a Signed GBL File Using a Hardware Security Module 46
6.7.13 Create a GBL File from an ELF File 46
6.7.14 Create an Encrypted GBL File with an Unencrypted Secure Element Upgrade File 47
6.7.15 Create a GBL File with Version Dependencies 48
6.8 Kit Utility Commands 49
6.8.1 Firmware Upgrade 49

6.8.2 Kit Information Probe 50
6.8.3 Adapter Reset Command 50
6.8.4 Adapter Debug Mode Command 51
6.8.5 List Adapter IP Configuration Command 51
6.8.6 Adapter DHCP Command 51
6.8.7 Set Static IP Configuration Command 52

6.9 Device Erase Commands 52
6.9.1 Erase Chip 52
6.9.2 Erase Region 52
6.9.3 Erase Pages in Address Range 53

6.10 Device Lock and Protection Commands 53
6.10.1 Debug Lock 53
6.10.2 Debug Unlock 53
6.10.3 Write Protect Flash Ranges 54
6.10.4 Write Protect Flash Region 54
6.10.5 Disable Write Protection 54

6.11 Device Utility Commands 54
6.11.1 Device Information Command 55
6.11.2 Device Reset Command 55
6.11.3 Device Recovery Command 55
6.11.4 Device Z-Wave QR Code Command 56

6.12 External SPI Flash Commands 56
6.12.1 Erase External SPI Flash Command 56
6.12.2 Read External SPI Flash Command 57
6.12.3 Write External SPI Flash Command 57

6.13 Advanced Energy Monitor Commands 57
6.13.1 Measure Average Current in a Time Window 58
6.13.2 Log Current Measurements as Time Series Data 58
6.13.3 Start Logging on Trigger Event. .59

6.14 Serial Wire Output Read Commands .59
6.14.1 Configure SWO Speed 60
6.14.2 Read SWO Until Timeout 60
6.14.3 Read SWO Until a Marker Is Found 60
6.14.4 Dump Hex Encoded SWO Output. .61

6.15 NVM3 Commands 61
6.15.1 Read NVM3 Data From a Device .61
6.15.2 Parse NVM3 Data 62
6.15.3 Initialize NVM3 Area in a File 62
6.15.4 Write NVM3 Data Using a Text File 63
6.15.5 Write NVM3 Data Using CLI Options. 64

6.16 CTUNE Commands 64
6.16.1 CTUNE Get Command 65
6.16.2 CTUNE Set Command 65
6.16.3 CTUNE Autoset Command 65

6.17 Security Commands. 65
6.17.1 Get Device Status 66

6.17.2 Generate Key Pair 67
6.17.3 Write Public Key to Device 68
6.17.4 Read Public Key from Device 68
6.17.5 Configure Lock Options 69
6.17.6 Lock Debug Access 69
6.17.7 Secure Debug Unlock. 70
6.17.8 Disable Tamper 74
6.17.9 Device Erase using Secure Element .75
6.17.10 Disable Device Erase 75
6.17.11 Roll Challenge 76
6.17.12 Generate Example Authorization File 77
6.17.13 Generate Access Certificate 79
6.17.14 Generate Unsigned Command File .80
6.17.15 Generate Example Configuration File 81
6.17.16 Write User Configuration 83
6.17.17 Read User Configuration 84
6.17.18 Get Security Store Path. 85
6.17.19 Write AES Decryption Key 85
6.17.20 Read Device Certificates 86
6.17.21 Vault Device Attestation 87

6.18 Util Commands 88
6.18.1 Key Generation 88
6.18.2 Generating a Signing Key 88
6.18.3 Key to Token. 88
6.18.4 Key Config Generation 89
6.18.5 Generate Certificate 90
6.18.6 Sign Certificate 90
6.18.7 Verify Signature 91
6.18.8 Application Information 91
6.18.9 Print Section Header Information from an ELF File 92
6.18.10 Get RAM and Flash Usage of an ELF Application 93
6.18.11 Print Header Information of an RPS File 94

6.19 OTA Commands 95
6.19.1 Create an OTA Bootloader File 95
6.19.2 Create a Null OTA File 95
6.19.3 Print OTA File Information 96
6.19.4 Sign an OTA File 97
6.19.5 Create an OTA File for External Signing 97
6.19.6 Externally Sign an OTA File. 98
6.19.7 Verify Signature of an OTA File 98
6.19.8 Create an OTA Matter File 99
6.19.9 Parse a Matter OTA File 100

6.20 Post-Build Command 101
6.20.1 Execute a Project Post-Build File 101

6.21 RPS Commands 105
6.21.1 Create an RPS File From a Binary Image 105
6.21.2 Create an RPS File From an ELF Image 105

6.21.3 Create an RPS File from a Hex/s37 Image . . . 106
6.21.4 Create an RPS File For Upgrading On-Device Key . . . 106
6.21.5 Create a Secure RPS Application Image . . . 107
6.21.6 Convert an Existing RPS Application Image . . . 108
6.21.7 Combine Multiple RPS Images Into a Single RPS File . . . 109
6.22 VUART Commands . . . 109
6.22.1 VUART Communications Until Timeout . . . 109
6.22.2 VUART Communications Until a Marker is Found . . . 110
6.23 RTT Commands . . . 110
6.23.1 RTT Communications Until a Marker is Found . . . 110
6.23.2 RTT Communications Until Timeout . . . 111
6.23.3 RTT Communications Over Virtual Terminals . . . 111
6.23.4 RTT Communications With a Custom RTT Buffer Configuration . . . 112
6.24 Serial Commands . . . 112
6.24.1 Load an RPS Application Over Serial . . . 113
6.24.2 Lock Debug Access to M4/TA Core . . . 113
6.24.3 Unlock Debug Access to M4/TA Core With Existing Token . . . 114
6.24.4 Unlock Debug Access to M4/TA Core Without Existing Token . . . 114
6.25 Manufacturing Commands . . . 115
6.25.1 List Available Memory Regions . . . 116
6.25.2 Read Memory Region Data From Device . . . 117
6.25.3 Read Specific Fields From Memory Region . . . 117
6.25.4 Write Memory Region Data to Device . . . 118
6.25.5 Erase Memory Region Data From Device . . . 118
6.25.6 Dump Configuration Data of Device . . . 119
6.25.7 Initialize PUF And Generate Activation Code . . . 120
6.25.8 Provision Security Keys to the Device . . . 121
6.25.9 Get Information About Device Configuration . . . 122

7. Software Revision History . . . 123
7.1 Version 1.16 . . . 123
7.2 Version 1.15 . . . 123
7.3 Version 1.14 . . . 124
7.4 Version 1.13 . . . 125
7.5 Version 1.12 . . . 125
7.6 Version 1.11 . . . 126
7.7 Version 1.10 . . . 126
7.8 Version 1.9 . . . 126
7.9 Version 1.8 . . . 127
7.10 Version 1.7 . . . 127
7.11 Version 1.5 . . . 127
7.12 Version 1.4 . . . 127
7.13 Version 1.3 . . . 127
7.14 Version 1.2 . . . 127

7.15 Version 1.1 127
7.16 Version 1.0 127
7.17 Version 0.25 128
7.18 Version 0.24 128
7.19 Version 0.22 128
7.20 Version 0.21 128
7.21 Version 0.16 128
7.22 Version 0.15 129
7.23 Version 0.14 129
7.24 Version 0.13 129
7.25 Version 0.12 129
7.26 Version 0.11 129

1. Introduction

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.

Simplicity Commander is designed to support the Silicon Labs Wireless STK and STK platforms.

The primary intended audience for this document is software engineers, hardware engineers, and release engineers who are familiar with programming the EFR32 and EM3xx. This reference guide describes how to use the Simplicity Commander CLI. It provides general information on file formats supported by Simplicity Commander and the Silicon Labs bootloaders, and includes details on using the Simplicity Commander commands, options, and arguments. It also includes example command line inputs and outputs so you can gain a better understanding of how to use Simplicity Commander effectively.

2. File Format Overview

Simplicity Commander works with different file formats: .bin, .s37, .ebl, .gbl, and .hex. Each file format serves a slightly different purpose. The file formats supported by Simplicity Commander are summarized below.

2.1 Motorola S-record (s37) File Format

Silicon Labs uses the Simplicity Studio as its Integrated Development Environment (IDE) and leverages the IAR Embedded Workbench for ARM platforms. This tool combination produces Motorola S-record files, s37 specifically, as its output. (For more information on Motorola S-record file format, see http://en.wikipedia.org/wiki/S_record.) In Silicon Labs development, an s37 file contains programming data about the built firmware and generally only represents a single piece of firmware—application firmware or bootloader firmware—but not both. An application image in s37 format can be loaded into a supported target device using the Simplicity Commander `flash` command. The s37 format can represent any combination of any byte of flash in the device. The Simplicity Commander `convert` command can also be used to read multiple s37 files and hex files; output an s37 file for combining multiple files into a single file; and modify individual bytes of a file.

2.2 Update Image File Formats

An update image file provides an efficient and fault-tolerant image format for use with Silicon Labs bootloaders to update an application without the need for special programming devices. Two image formats are supported: Gecko Bootloader (GBL) format for use with the Silicon Labs Gecko Bootloader introduced for use with EFR32 devices and Ember Bootloader (EBL) format for use with legacy Ember bootloaders. See *UG103.6: Application Development Fundamentals: Bootloading* for more details about these image file formats and bootloader use with different platforms.

Update image files are generated by the Simplicity Commander `gbl create` or `ebl create` command. These formats can only represent firmware images; they cannot be used to capture Simulated EEPROM token data (as described by *AN703: Using Simulated EEPROM Version 1 and Version 2 for the EM35x and EFR32 Series 1 SoC Platforms*). GBL upgrade files may contain data that gets flashed outside the main flash.

Bootloaders can receive an update image file either over-the-air (OTA) or via a supported peripheral interface, such as a serial port, and reprogram the flash in place. Update image files are generally used in later stage development and for upgrading manufactured devices in the field.

During development, bootloaders should be loaded onto the device using the .s37 or .hex file format. If the Gecko Bootloader with support for in-field bootloader upgrades is used, it is possible to perform a bootloader upgrade using a GBL update image. For other bootloaders or file formats, do not attempt to load a bootloader image onto the device as an update image.

2.3 Intel HEX-32 File Format

Production programming uses the standard Intel HEX-32 file format. The normal development process for EFR32 chips involves creating and programming images using the s37 and ebl file formats. The s37 and ebl files are intended to hold applications, bootloaders, manufacturing data, and other information to be programmed during development. The s37 and ebl files, though, are not intended to hold a single image for an entire chip. For example, it is often the case that there is an s37 file for the bootloader, an s37 file for the application, and an s37 file for manufacturing data. Because production programming is primarily about installing a single, complete image with all the necessary code and information, the file format used is Intel HEX-32 format. While s37 and hex files are functionally the same—they simply define addresses and the data to be placed at those addresses—Silicon Labs has adopted the conceptual distinction that a single hex file contains a single, complete image often derived from multiple s37 files. You can use the Simplicity Commander `convert` command to read multiple hex files and s37 files; output a hex file for combining multiple files into a single file; and modify individual bytes of a file.

Note: Simplicity Commander is capable of working identically with s37 and hex files. All functionality that can be performed with s37 files can be performed with hex files. Ultimately, with respect to production programming, Simplicity Commander `flash` command allows the developer to load a variety of sources onto a physical chip. The `convert` command can be used to merge a variety of sources into a final image file and modify individual bytes in that image if necessary.

The following table summarizes the inputs and outputs for the different file formats used by Simplicity Commander.

Table 2.1. File Format Summary

	Inputs					Outputs					
	ebl	s37	hex	bin	chip	ebl	s37	hex	bin	chip	rps
flash		X	X	X						X	
readmem					X		X	X	X		
convert		X	X	X			X	X	X		
ebl create		X	X	X		X					
ebl parse	X						X	X	X		
rps create		X	X	X							X

3. General Information

3.1 Installing Simplicity Commander

You can install Simplicity Commander using Simplicity Studio or by downloading one of the following standalone versions and then completing the installation:

https://www.silabs.com/documents/public/software/SimplicityCommander-Linux.zip

https://www.silabs.com/documents/public/software/SimplicityCommander-Mac.zip

https://www.silabs.com/documents/public/software/SimplicityCommander-Windows.zip

3.2 Command Line Syntax

To execute Simplicity Commander commands, start a Windows command window, and change to the Simplicity Commander directory. The general command line structure in Simplicity Commander looks like this:

```
commander [command] [options] [arguments]
```

where:

- `commander` is the name of the tool.
- `command` is one of the commands supported by Simplicity Commander, such as, `flash`, `readmem`, `convert`, etc. The command-specific help provides additional information on each command.
- `option` is a keyword that modifies the operation of the command. Options are preceded with `--` (double dash) as described for each command. Some commands have single-character short versions which are preceded by `-` (single dash). Refer to the command-specific help for the single-dash shorthands.
- `argument` is an item of information provided to Simplicity Commander when it is started. An argument is commonly used when the command takes one or more input files.
- square brackets indicate *optional* parameters as in this example: `commander flash [filename(s)] [options]`
- angle brackets indicate *required* parameters as in this example: `commander readmem --output <filename>`

3.3 General Options

3.3.1 Help (`--help`)

Displays help for all Simplicity Commander commands and command-specific help for each command.

Command Line Syntax

```
$ commander --help
```

Command Line Usage Output

Simplicity Commander help displays a list of all Simplicity Commander commands. The following figure is an example.

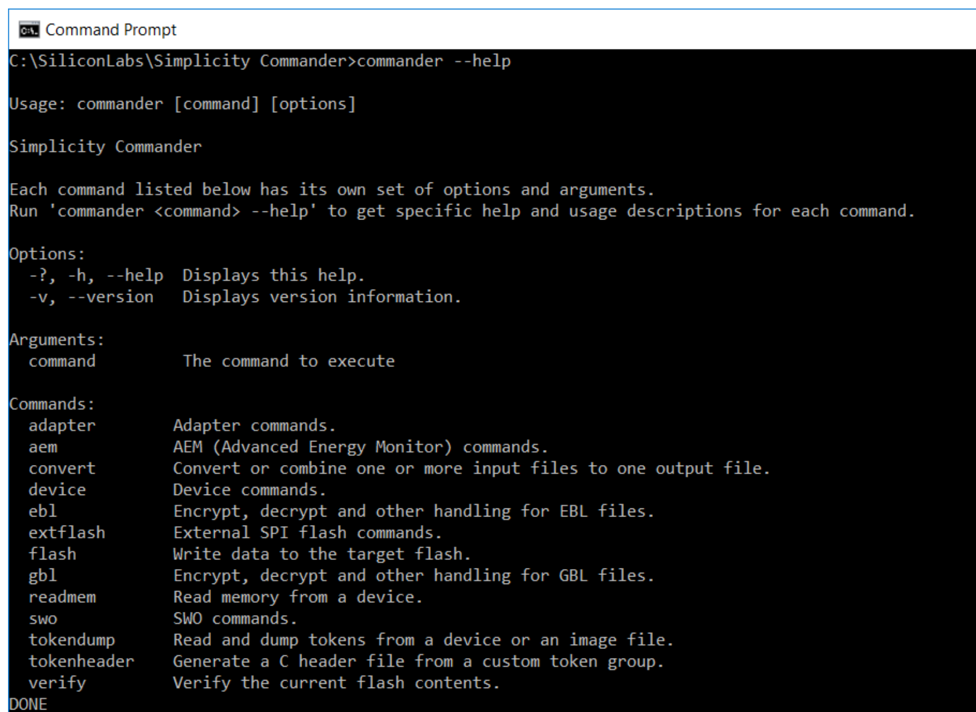


Figure 3.1. Simplicity Commander Help

To display help on a specific Simplicity Commander command, enter the name of the command followed by `--help`.

Command Line Input Example

```
$commander flash --help
```

Command Line Output Example

Simplicity Commander displays help for the flash command in the following figure.

Command Prompt

```
C:\SiliconLabs\Simplicity Commander>commander flash --help

Usage: commander flash [filename(s)] [options]
Write one or more files to the target flash.

Options:
  -?, -h, --help                       Displays this help.
  -v, --version                        Displays version information.
  --device, -d <device>                The device, device family or platform to
                                       target. Examples of strings that are
                                       understood: "EFR32MG1P233F256GM48",
                                       "EFR32MG", "EFR32", "EFR32F256". Required
                                       for some operations.
  --force                              Force operation. This will convert
                                       non-fatal errors to warnings, allowing
                                       the process to continue.
  --serialno, -s <serial number>       J-Link serial number.
  --ip <IP>                            IP Address.
  --speed <speed in kHz>               Debug interface speed.
  --tif <SWD|JTAG|C2>                  Target debug interface.
  --irpre <IR length>                  JTAG: Total length of instruction
                                       registers of all devices closer to TDI
                                       than the addressed ARM device.
  --drpre <Data bits>                  JTAG: Total number of data bits closer
                                       to TDI than the addressed ARM device.
  --address <address>                  Address to flash to. Not applicable for
                                       hex or s37 files which contain address
                                       information.
  --halt                               Leave the target halted after flashing.
                                       By default the device is reset by a pin
                                       reset after flashing.
  --masserase                          Supply this to do a mass erase of the
                                       entire main flash before flashing.
                                       Otherwise only affected pages are erased.
  --noverify                           Don't verify contents written to flash
                                       (verification is enabled by default).
  --patch, -p <address:data[:length]>  Patch memory contents.
                                       Data is interpreted as an unsigned
                                       integer. The optional length parameter
                                       can be used to define the number of bytes
                                       write, up to 8.
  --token <TOKEN_NAME:value>           Single token with its new value.
  --tokenfile <filename>               File describing tokens to write.
  --tokengroup <tokengroup>            Which set of tokens to use. Supported:
                                       znet

Arguments:
  flash
  filename(s)                          File(s) to flash.

DONE
```

Figure 3.2. Simplicity Commander Flash Command Help

3.3.2 Version (`--version`)

Displays the version information for Simplicity Commander, J-Link DLL, and EMDLL, and a list of detected USB devices. If you use this option in conjunction with another command or command/option, Simplicity Commander displays this extra information before any command is executed.

Command Line Syntax

```
$ commander --version
```

Command Line Usage Output

Simplicity Commander displays version information. The following figure is an example.

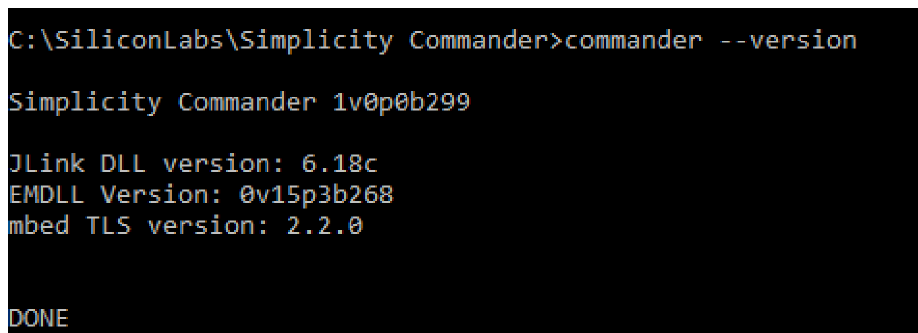


Figure 3.3. Simplicity Commander Version Information

3.3.3 Device (`--device <device name>`)

Specifies a target device for the command. If this option is supplied, no auto-detection of the target device is used. In some cases, such as when using `convert` with the --`token` option, this option is required.

For convenience, Simplicity Commander attempts to parse the `--device` option so that a complete part number is normally not required as a command input. For example, Simplicity Commander interprets `commander --device EFR32` to mean that the selected device is an EFR32, which has implications regarding the memory layout and available features of this specific device. As another example, Simplicity Commander interprets `--device EFR32F256` as an EFR32 with 256 kB flash memory.

Using a complete part number such as `--device EFR32MG1P233F256GM48` is always supported and recommended.

Command Line Syntax

```
$ commander <command> --device <device name>
```

Command Line Input Example

```
$ commander device info --device Cortex M3
```

3.3.4 J-Link Connection Options

Use the following options to select a J-Link device to connect to and use for any operation that requires a connection to a kit or debugger. You can connect over IP (using the `--ip` option), over USB (using the `--serialno` option), or you can provide the serial port name or device file (using the `--identifybyserialport` option) as shown in the following examples. You can use only one of these options at a time. If no option is provided, Simplicity Commander attempts a connection to the only USB connected J-Link adapter.

Note: Providing the `--identifybyserialport` option only lets Simplicity Commander use the serial port name to *identify* the corresponding J-Link device; Simplicity Commander will still connect to the J-Link device over USB (similarly to when you provide the `--serialno` option).

Command Line Syntax

```
$ commander <command> --serialno <J-Link serial number>
```

Command Line Input Example

```
$ commander adapter probe --serialno 440050184
```

Command Line Syntax

```
$ commander <command> --ip <IP address>
```

Command Line Input Example

```
$ commander adapter probe --ip 10.7.1.27
```

Command Line Syntax

```
$ commander <command> --identifybyserialport <serial port name>
```

Command Line Input Example

```
$ commander adapter probe --identifybyserialport /dev/ttyUSB1
```

3.3.5 Debug Interface Configuration

Use the `--tif` and `--speed` options to configure the target interface and clock speed when connecting the debugger to the target device.

Simplicity Commander supports using Serial Wire Debug (SWD) or Joint Test Action Group (JTAG) as the target interface. All currently supported Silicon Labs hardware works with SWD, while some can also be used with JTAG. Custom hardware may require JTAG to be used.

The maximum clock speed available typically depends on the debug adapter, the target device, and the physical connection between the two. Silicon Labs kits typically support speeds up to 1000 – 8000 kHz, depending on the kit model. If the selected clock speed is higher than what the adapter supports, the clock speed will fall back to using the highest speed it does support. You may want to select a lower clock speed if the debug connection is unstable or not working at all when working with custom hardware with longer debug cables or when the electrical connections are less than ideal.

If the `--tif` and `--speed` options are not used, the default configuration is SWD and 4000 kHz.

Command Line Syntax

```
$ commander <command> [--tif <target interface>] [--speed <speed in kHz>]
```

Command Line Input Example

```
$ commander device info --tif SWD --speed 1000
```

Command Line Output Example

```
Setting debug interface speed to 1000 kHz
Setting debug interface to SWD
Part Number     : EFR32BG1P332F256GJ43
Die Revision     : A2
Production Ver     : 138
Flash Size     : 256 kB
SRAM Size     : 32 kB
Unique ID     : 000b57fffe0934e3
DONE
```

3.3.6 Graphical User Interface

Displays a Graphical User Interface (GUI) for laboratory use of Simplicity Commander. The GUI can be used in the lab for such typical tasks as:

- Flashing device images
- Upgrading Silicon Labs kit firmware and configuration
- Setting device lock features
- Accessing the kit's Admin console
- Communicating with the target device via multiple protocols, including:
  - SEGGER Real Time Transfer (RTT)
  - Serial Wire Output (SWO)
  - Virtual UART (VUART)

Command Line Syntax

```
$ commander
```

3.3.7 Timestamp (`--timestamp`)

Add a timestamp to the Simplicity Commander output.

Command Line Syntax

```
$ commander <command> --timestamp
```

Command Line Usage Output

Display a timestamp to all output from Simplicity Commander.

Command Line Input Example

```
$commander device reset --timestamp
```

Command Line Output Example

Simplicity Commander displays the timestamp for the device reset command.

```
17:00:39.194 Resetting chip...
DONE
```

3.4 Output and Exit Status

The exit status of Simplicity Commander can take on a few different values. Whenever an operation completed successfully, Simplicity Commander's exit status is 0 (zero). Any error will cause the exit status to be non-zero.

Simplicity Commander defines the following exit status codes.

Exit Status	Description
0	No error occured
-1	Input error. For example, this could be a missing command line option, non-existent command, or an invalid filename.
-2	Run time error. Used whenever anything goes wrong when executing the command. Examples include not being able to connect to a debug adapter or flash verification failed.

Note: Some operations systems present the exit status as an unsigned integer. On these systems, -1 will be interpreted as 255, -2 as 254, and so on.

The operating system itself may create other exit codes if the application crashes. These will always be non-zero and are out of the control of Simplicity Commander.

All errors and potential error conditions are indicated in Simplicity Commander's output in addition to the exit status. All errors are displayed with the prefix "ERROR:". All warnings are displayed with the prefix "WARNING:".

Any output from Simplicity Commander will always end with "DONE". This does not indicate that the operation was successful, merely that execution has finished.

Example of an error in Windows follows.

```
C:\>commander device info -s 440000000
ERROR: Unable to connect with device with given serial number
ERROR: Could not open J-Link connection.
DONE

C:\>echo %errorlevel%
-2
```

4. EFR32 Custom Tokens

4.1 Introduction

Simplicity Commander supports defining custom token groups for reading and writing. Custom tokens work just like manufacturing tokens, but the definition and location of the tokens is configurable to suit different requirements.

There are two different ways for Simplicity Commander to find and use custom token definition files. For Simplicity Commander to treat the custom token file in the same way as a regular token group, the file must be placed in a specific location as described in section 4.2 Custom Token Groups.

The other option is to use the `--tokendefs` command line option instead of the `--tokengroup` option. With this method, Simplicity Commander uses a token definition file in an arbitrary location, for example, under revision control. For more information, see section 4.8 Using Custom Token Files in Any Location.

4.2 Custom Token Groups

For Simplicity Commander to treat custom token files like regular token groups, the file must be placed in a specific `tokens` folder and the filename must follow a special syntax.

The location and initialization of the tokens folder depends on the operating system used.

On Windows and Linux, the `tokens` folder is included in the zip file and is placed alongside the executable in the installation directory.

On Mac OS X, the folder named `~/Library/SimplicityCommander/tokens/` is generated automatically in the user's home directory when running
`commander` on the command line for the first time. Running `commander --help`, for example, is enough to ensure that the folder with files is created. Inside this `tokens` folder, there is a file named `tokens-example-efr32.json`. This file provides an example of the token types and locations currently supported by Simplicity Commander.

The syntax of the filename is `tokens-<group name>-<architecture>.json`. `<group name>` is the name of the custom token group and can be any string. `<architecture>` is a string describing which devices the token definitions apply to. The following table lists the supported architecture strings.

Architecture	Devices
`efr32`	All Series 1 EFR32 devices
`efr32xg2`	All Series 2 EFR32 devices
`em3xx`	All EM3xx devices
`efm32`	All EFM32 devices (Series 0 and 1)
`ezr32`	All EZR32 devices

For example, to define the token group myapp for EFR32 Series 1 devices, the filename would be `tokens-myapp-efr32.json`.

4.3 Creating Custom Token Groups

To define a custom token group, copy `tokens-example-efr32.json` to a new file in the same directory using the following naming convention: `tokens-<groupname>-efr32.json`

For example: `tokens-myapp-efr32.json`

To verify that Simplicity Commander sees the new file, run

```
$ commander tokendump --help
```

The name of your token group (for example, "myapp") should be listed as a supported token group like this:

`--tokengroup <tokengroup>` which set of tokens to use. Supported: myapp, znet

4.4 Defining Tokens

Each token in the JSON file has the following properties.

Property	Description
name	The name of the token, which is used as an identifier when dumping or writing tokens.
page	The named memory region to use for the token. For more information, see section 4.5 Memory Regions.
offset	The offset in number of bytes from the start of the memory region at which to place the token.
sizeB	The size of the token in bytes. • A token of size 1 is interpreted as an unsigned 8-bit integer. • A token of size 2 is interpreted as an unsigned 16-bit integer. • A token of size 4 is interpreted as an unsigned 32-bit integer. • Any other size is interpreted as a byte array of the given size.
string	Optional boolean. If this property is `true`, the token is interpreted as a zero terminated ASCII string instead of a byte array. The maximum string length is `sizeB - 1` because one byte is reserved for the zero terminator.
description	A plain text description of the token. This property is currently only used for documentation of the JSON file.

4.5 Memory Regions

The following values are valid data in the "page" option:

USERDATA

The user data page is a separate flash page intended for persistent data and configuration. The user data page is **not** erased when disabling debug lock. It can, however, be erased by a specific page erase.

The user data page is located at address 0x0FE00000. It is 2 kB on Series 1 EFR32 devices and 1 kB on Series 2 EFR32 devices.

LOCKBITSDATA

On Series 1 EFR32 devices, the lock bits page is used by the chip itself to configure flash write locks, debug lock, AAP lock, and so on. However, the last 1.5 kB of this page is unused by the device itself and has the important property that it is erased when disabling debug lock. A regular mass erase by the MSC—typically by executing the `commander device masserase` or `commander flash --masserase` command—does not erase the lock bits page.

The lock bits page is located at address 0x0FE04000 with size 2 kB on Series 1 EFR32 devices. Tokens in this page must use an offset of at least 0x200 on these devices; otherwise, collisions with chip functionality can occur.

On Series 2 EFR32 devices, there is no physical lock bits page. Instead, the LOCKBITSPAGE region is defined to be the first 2 kB of the last flash page in the main flash block. This maintains backwards compatibility, while still ensuring that any data in this region is erased when the device is erased during debug unlock.

4.6 Token File Format Description

A token file declares what values are programmed for manufacturing tokens on the chip. Lines are composed of one of the following forms:

```
<token-name> : <data>
<token-name> : !ERASE!
```

Follow these guidelines when using a token file:

- Omitted tokens are left untouched and not programmed on the chip.
- Token names are case insensitive.
- All integer values are interpreted as hexadecimal numbers in BIG-endian format and must be prefixed with '0x'.
- Blank lines and lines beginning with # (hashtag) are ignored.
- Byte arrays are given in hexadecimal format without a leading '0x'.
- Specifying !ERASE! for the data sets that token to all 0xFF.
- The token data can be in one of three main forms: byte-array, integer, or string.
- Byte arrays are a series of hexadecimal numbers of the required length.
- Integers are BIG-endian hexadecimal numbers that must be prefixed with '0x'.
- String data is a quoted set of ASCII characters.

4.7 Using Custom Token Files

Refer to 4.1 Introduction for a definition of custom token files and where they should be located for Simplicity Commander to find them automatically. To use a custom token file located in the `tokens` folder, run Simplicity Commander with a `--tokengroup` option corresponding to the name of the JSON file. For example, if the file was named `tokens-myapp-efr32.json`, use this option:

```
--tokengroup myapp
```

To create a text file useful as input to the `flash` or `convert` commands, the easiest way is to start by dumping the current data from a device.

For example:

```
$ commander tokendump -s 440050148 --tokengroup myapp --outfile mytokens.txt
```

`mytokens.txt` can then modified to have the desired content, and then used when flashing devices or creating images in this way:

```
$ commander flash -s 440050148 --tokengroup myapp --tokenfile mytokens.txt
```

To be able to read the custom token data from an application, Simplicity Commander provides the `tokenheader` command, which generates a C header file that can be included in an application. See section 6.4.4 Generate C Header Files from Token Groups for details.

4.8 Using Custom Token Files in Any Location

In some cases, it is more convenient to have the custom token defintions file somewhere in the file system (for example, if it is placed under revision control). Simplicity Commander supports this functionality with the `--tokendefs` option which refers to a JSON file anywhere in the file system. Use it instead of the `--tokengroup` option.

For example:

```
$ commander tokendump --tokendefs my_tokens.json --outfile mytokens.txt
$ commander flash --tokendefs my_tokens.json --tokenfile mytokens.txt
```

5. Security Overview

This chapter describes essential security features in Simplicity Commander.

5.1 Security Store

Security Store is the location where all files generated and used by the security commands in Simplicity Commander are stored. You can find the path to Security Store with the `commander security getpath` command. Unless the `--nostore` option is used with security commands, Simplicity Commander will store all keys, certificates, and configuration files seen in Security Store. Descriptions of the files appear below.

- **access_certificate.bin** – certificate delegating permission to unlock debug access of a device.
- **archive folder** – folder used to store all outdated files (for example, all files in the challenge folder are moved here when a challenge is rolled).
- **cert_key.pem** – private key used to sign unlock token.
- **cert_pubkey.pem** – public key used in certificate. Public key corresponding to *cert_key.pem.*
- **certificate_authorization.json** – configuration file used to define authorizations given by access certificate. May be edited.
- **challenge_xxx folder** – folder used to store files related to a challenge.
  - **unlock_payload_xxx.bin** – payload used to unlock secure debug access.
  - **unlock_command_to_be_signed_dd_mm_yyyy.bin** – command token that needs to be signed with *cert_key.pem*
- **command_key.pem** – private command key used to sign access certificate.
- **command_pubkey.pem** – public command key stored on device. Public key corresponding to *command_key.pem.*
- **user_configuration.json** – configuration file used in write config. May be edited.

When running the `commander security unlock` command, Simplicity Commander will use all available files to attempt to unlock the debug access. If anything is missing, you will be asked to provide the file as an option to the command. The file will then be stored in Security Store, unless the `--nostore` option is used.

5.2 Access Certificate

An access certificate is used to delegate access to a single device to another key, which is called a certificate key. This scheme supports security models where the command key is kept in a secure location, while the certificate key can be used with more lenient security practices.

The access certificate contains the serial number of the device it applies to, a description of what actions it gives access to, and the public certificate key. An outline of the access certificate is illustrated in the following figure.

The device serial number uniquely identifies each device. It can be displayed by executing the `commander security status` command. The **certificate_authorizations.json** file sets the authorizations for the certificate. The current version of Simplicity Commander does not support any modifications to the authorization file, but it will be available in future versions. The private certificate key corresponding to the public certificate key in the certificate is used to generate a signature required to unlock debug access. For more information, see 5.3 Challenge and Command Signing. The certificate is authenticated by signing it with the private command key corresponding to the public command key written to the device. The signing of the certificate may be done by passing an unsigned certificate to a Hardware Security Module (HSM) containing the private key or by providing the private key to Simplicity Commander (that is, for development) using the `--command-key` option.

Figure 5.1. Access Certificate

5.3 Challenge and Command Signing

The part of the data that needs to be signed to create a valid unlock command is called the *challenge*. Secure Element generates this random data. It remains unchanged until it is updated to a new random value by the security rollchallenge command.

By updating the challenge, any existing command signatures are effectively invalidated because part of the data the signature encompasses has changed. This allows the owner of the device to give debug access to someone else for a limited amount of time.

A command signature is created by signing a binary containing the data fields in yellow in the following figure; Simplicity Commander sets the unlock command ID, command parameters, and the security challenge using the private key corresponding to the public key in the access certificate.

The `security gencommand` command creates a file containing these elements, but does not include the signature. If the certificate private key is not available to the user, the signature must be obtained from another party—for example, an HSM. If the user possesses the certificate private key, Security Commander can create the signed unlock command using the `security unlock` command. By passing the command signature and the access certificate to the Debug Challenge interface, the debug interface is temporarily unlocked until the next power-on or pin reset.

Figure 5.2. Unlock Command Signature

6. Simplicity Commander Commands

This section includes the following information for using each Simplicity Commander command:

- Command Line Syntax
- Command Line Input Example
- Command Line Output Example

In cases where the Command Line Syntax is the same as the Command Line Input Example, only the former is included.

The Simplicity Commander commands are organized in the following categories:

- 6.1 Device Flashing Commands
- 6.2 Flash Verification Command
- 6.3 Memory Read Commands
- 6.4 Token Commands
- 6.5 Convert and Modify File Commands
- 6.6 EBL Commands
- 6.7 GBL Commands
- 6.8 Kit Utility Commands
- 6.9 Device Erase Commands
- 6.10 Device Lock and Protection Commands
- 6.11 Device Utility Commands
- 6.12 External SPI Flash Commands
- 6.13 Advanced Energy Monitor Commands
- 6.14 Serial Wire Output Read Commands
- 6.15 NVM3 Commands
- 6.16 CTUNE Commands
- 6.17 Security Commands
- 6.18 Util Commands
- 6.19 OTA Commands
- 6.20 Post-Build Command
- 6.21 RPS Commands
- 6.23 RTT Commands
- 6.22 VUART Commands
- 6.24 Serial Commands

6.1 Device Flashing Commands

The commands in this section all require a working debug connection for communicating with the device. You would normally always use one of the J-Link connection options when running the `flash` command, but it is intentionally left out of most of the examples to keep them short and concise.

6.1.1 Flash Image File

Flashes the image in the specified filename to the target device, starting at the specified address. The address value is interpreted as a hexadecimal number. The affected bytes will be erased before writing. If the image contains any partial flash pages, these pages will be read from the device and patched with the image contents before erasing the page and writing back. After writing, the affected flash areas are read back and compared. Finally, the chip is reset using a pin reset, making code execution start. The debugger to connect to is indicated by the J-Link serial number (`--serialno` option). The `--binary` option can be used to interpret all file types as flat binaries, bypassing any parsing of GBL, S-record, or Intel Hex files. For example, you can use this to test firmware upgrade using an internal storage bootloader. The `--include-section` and `--exclude-section` options can be used when flashing an Elf file.

Command Line Syntax

```
$ commander flash <filename> --address <address> --serialno <serial number> [--binary --include-section
<section> --exclude-section <section>]
```

Command Line Input Example

```
$ commander flash blink.bin --address 0x0 --serialno 440012345
```

Connects to the J-Link debugger with serial number 440012345 and flashes the image in blink.bin to the target device, starting at address 0.

Command Line Output Example

```
Flashing blink.s37.
Flashing 2812 bytes, starting at address 0x00000000
Resetting...
Uploading flash loader...
Waiting for flashloader to become ready...
Erasing flash...
Flashing...
Verifying written data...
Resetting...
Finished!
DONE
```

6.1.2 Flash Using IP Address without Verification and Reset

Flashes the image in the specified filename to the target device, using the IP address specified. The data in flash is not verified after flashing, and the device is left halted after flashing.

Command Line Syntax

```
$ commander flash <filename> --ip <IP> --halt --noverify>
```

Command Line Input Example

```
$ commander flash blink.s37 --ip 10.7.1.27 --halt --noverify
```

Flashes the image in blink.s37 to the target device, using the IP address 10.7.1.27. The data in flash is not verified after flashing, and the device is left halted after flashing.

Command Line Output Example

```
Flashing blink.s37.
Flashing 2812 bytes, starting at address 0x00000000
Resetting...
Uploading flash loader...
Waiting for flashloader to become ready...
Erasing flash...
Flashing...
Finished!
DONE
```

6.1.3 Flash Several Files

Flashes the images to the target device. Any overlapping data is considered an error.

Command Line Syntax

```
$ commander flash <filename> <filename>
```

Command Line Input Example

```
$ commander flash blink.s37 userpage.hex
```

Flashes the images in blink.s37 and userpage.hex to the target device.

Command Line Output Example

```
Adding file blink.s37...
Adding file userpage.hex...
Flashing 2812 bytes, starting at address 0x00000000
Resetting...
Uploading flash loader...
Waiting for flashloader to become ready...
Erasing flash...
Flashing...
Verifying written data...
Finished!
Flashing 2048 bytes, starting at address 0x0fe00000
Resetting...
Uploading flash loader...
Waiting for flashloader to become ready...
Erasing flash...
Flashing...
Verifying written data...
Resetting...
Finished!
DONE
```

6.1.4 Patch Flash

Writes the specified byte(s) to the flash. The affected pages will be read from the device and patched with this data before erasing the page and writing back. When you use the `--patch` option, the patch memory data is interpreted as an unsigned integer. The optional `length` argument can be used to define the number of bytes, up to 8 bytes. If no length is specified, the default is to patch 1 byte.

Command Line Syntax

```
$ commander flash --patch <address>:<data>[:length]
```

Command Line Input Example

```
$ commander flash --patch 0x120:0xAB --patch 0x3200:0xA5A5:2
```

Writes the specified bytes 0xAB to address 0x120 and 0xA5A5 to address 0x3200. The affected pages will be read from the device and patched with this data before erasing the page and writing back.

Command Line Output Example

```
Patching 0x00000120 = 0xAB...
Patching 0x00003200 = 0xA5A5...
Flashing 2048 bytes, starting at address 0x00000000
Resetting...
Uploading flash loader...
Waiting for flashloader to become ready...
Erasing flash...
Flashing...
Verifying written data...
Finished!
Flashing 2048 bytes, starting at address 0x00003000
Resetting...
Uploading flash loader...
Waiting for flashloader to become ready...
Erasing flash...
Flashing...
Verifying written data...
Resetting...
Finished!
DONE
```

6.1.5 Patch Using Input File

Flashes the specified application while simultaneously patching the image file and the flash of the device. If a filename is inside the file, these bytes are patched before writing the image

Command Line Syntax

```
$ commander flash <filename> --patch <address>:<data>[:length] --patch <address>:<data>[:length]
```

Command Line Input Example

```
$ commander flash blink.s37 --patch 0x123:0x00FF0001:4 --patch 0x0FE00004:0x00
```

Flashes the blink application while simultaneously patching the image file and the flash of the device. Because 0x123 is inside the file, these bytes are patched before writing the image. Additionally, the user page will be read from the device and patched with this data before erasing the page and writing back.

Command Line Output Example

```
Flashing blink.s37.
Patching 0x00000123 = 00FF0001...
Patching 0x0FE00004 = 00...
Flashing 4096 bytes, starting at address 0x00000000
Resetting...
Uploading flash loader...
Waiting for flashloader to become ready...
Erasing flash...
Flashing...
Verifying written data...
Finished!
Flashing 2048 bytes, starting at address 0x0fe00000
Resetting...
Uploading flash loader...
Waiting for flashloader to become ready...
Erasing flash...
Flashing...
Verifying written data...
Finished!
DONE
```

6.1.6 Flash Tokens

This section describes how to flash one or more tokens from text file(s) and/or command line options with their new values. Manufacturing tokens are the only token type supported by Simplicity Commander; simulated EEPROM tokens are not supported. For more information on manufacturing tokens, see *AN961*: *Bringing Up Custom Nodes for the EFR32MG and EFR32FG Families*.

The `--tokengroup` option defines which group of tokens is used. Simplicity Commander currently has built-in support for the `znet` token group.

Silicon Labs recommends generating a token file from a device or image file using the `tokendump` command and then making modifications to this file for use with the `--tokenfile` option.

Command Line Syntax

```
$ commander flash --tokengroup <token group> --token <TOKEN_NAME:value> --tokenfile <filename>
```

Command Line Input Example

```
$ commander flash --tokengroup znet --token TOKEN_MFG_STRING:"IoT Inc"
```

Set the token `MFG_STRING` to have the value `IoT Inc`. The `TOKEN_` prefix is optional, that is, `TOKEN_MFG_STRING` and `MFG_STRING` are equivalent.

Command Line Input Example

```
$ commander flash --tokengroup znet --tokenfile tokens.txt
```

Sets the tokens specified in tokens.txt. All tokens in the file are processed, and if a duplicate is found, it will be treated as an error.

Command Line Input Example

```
$ commander flash --tokengroup znet --tokenfile tokens.txt --token TOKEN_MFG_STRING:”IoT Inc”
```

Sets the tokens specified in tokens.txt. Additionally, sets the `MFG_STRING` to the value given. All files and tokens specified on the command line are processed, and if a duplicate is found, it will be treated as an error.

Depending on the operating system and shell being used, some escapes may be needed to correctly specify a string. For example, on the command line in a Windows 7 Professional Command Prompt window, execute the following command:

```
$ commander flash --tokengroup znet --token "TOKEN_MFG_STRING:\"IoT Inc\""
```

Command Line Output Example

```
Flashing 2048 bytes to 0x0fe00000
Resetting...
Uploading flash loader...
Waiting for flashloader to become ready...
Erasing flash...
Flashing...
Verifying written data...
Resetting...
Finished!
DONE
```

6.2 Flash Verification Command

The `verify` command verifies the contents of a device against a set of files, tokens, and/or patch options without writing anything to the flash. It works just like the verification step of the `flash` command, but without actually flashing first. For example, the `verify` command can be used to verify that the application on a microcontroller is what you expect it to be.

Command Line Syntax

All options and examples for the `flash` command also apply to the `verify` command. The exceptions are the `--halt`, `--masserase`, and `--noverify` options that do not apply to the verify command.

```
$ commander verify [filename] [filename ...] [patch options] [token options]
```

Command Line Input Example

```
$ commander verify myimage.hex
```

Command Line Output Example

```
Parsing file myimage.hex...
Verifying 52000 bytes at address 0x00000000...OK!
Verifying 2048 bytes at address 0x0fe00000...OK!
DONE
```

6.3 Memory Read Commands

The `readmem` command reads data from a device and can either store it to file or print it in human-readable format. The location and length to be read from the device is defined by the `--range` and `--region` options. You can combine one or more ranges and regions to read and combine several different areas in flash to one file.

Note: Like `flash`, the commands in this section all require a working debug connection for communicating with the device. You would normally always use one of the J-Link connection options when running `readmem`, but this is left out of the examples to keep them short and concise.

The `--range` option supports two different range formats:

- The first is `<startaddress>:<endaddress>`, for example, `--range 0x4000:0x6000`. The range is non-inclusive, meaning that all bytes from 0x4000 up to and including 0x5FFF are read out.
- The second is `<startaddress>:+<length>`, which takes an address to start reading from, and a number of bytes to read. For example, the equivalent command line input to the previous example is `--range 0x4000:+0x2000`.

The `--region` option takes a named flash region with an @ prefix. Valid regions for use with the `--region` option are listed below.

Series 0 **EFM32, EZR32, EFR32:** `@mainflash`, `@userdata`, `@lockbits, @devinfo`

Series 1 **EFM32, EFR32**: `@mainflash`, `@userdata`, `@lockbits`, `@devinfo`, `@bootloader`

Series 2 **EFM32, EFR32**: `@mainflash`, `@userdata`, `@devinfo`

EM3xx: `@mfb`, `@cib`, `@fib`

6.3.1 Print Flash Contents

Specifies the range of memory to read from flash and prints data.

Command Line Syntax

```
$ commander readmem --range <startaddress>:<endaddress>
```

OR

Command Line Syntax

```
$ commander readmem --range <startaddress>:+<length>
```

Command Line Input Example

```
$ commander readmem --range 0x100:+128
```

Reads 128 bytes from flash starting at address 0x100 and prints it to standard out.

Command Line Output Example

```
Reading 128 bytes from 0x00000100...
{address:  0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F}
00000100: 12 F0 40 72 11 00 DF F8 C0 24 90 42 07 D2 DF F8
00000110: BC 24 90 42 03 D3 5F F0 80 72 11 00 01 E0 00 22
00000120: 11 00 DF F8 84 26 12 68 32 F0 40 72 0A 43 DF F8
00000130: 78 36 1A 60 70 47 80 B5 00 F0 90 FC FF F7 DD FF
00000140: 01 BD DF F8 70 16 09 68 08 00 70 47 38 B5 DF F8
00000150: 4C 06 00 F0 9F F9 05 00 ED B2 28 00 07 28 05 D0
00000160: 08 28 07 D1 00 F0 7C FC 04 00 0B E0 FF F7 E9 FF
00000170: 04 00 07 E0 40 F2 25 11 DF F8 3C 06 00 F0 B0 FC
DONE
```

6.3.2 Dump Flash Contents to File

Reads the contents of the specified user page and stores it in the specified filename. File format will be auto-detected based on file extension (.bin, .hex, or .s37). (See 2. File Format Overview for more information on file formats.)

Command Line Syntax

```
$ commander readmem --region <@region> --outfile <filename>
```

Command Line Input Example

```
$ commander readmem --region @userdata --outfile userpage.hex
```

Reads the contents of the region named userdata and stores it in an output file named userpage.hex.

Command Line Output Example

```
Reading 2048 bytes from 0x0fe00000...
Writing to userpage.hex...
DONE
```

6.4 Token Commands

The `tokendump` command generates a text dump of token data. It can take as input either a (set of) files using the same command line options as the `convert` command, or a microcontroller using the same command line options as the `readmem` command.

The output of `tokendump` can either be printed to standard output or written to an output file using the `--outfile` option. The file written when using the `--outfile` option is suitable for modification and re-use as input to the `flash`, `verify`, or `convert` commands using the `--tokenfile` option.

`tokendump` always requires a token group to be selected with the `--tokengroup` option. A token group is a defined set of tokens for a specific stack or application. Simplicity Commander only supports the `znet` token group.

Manufacturing tokens are the only token type supported by Simplicity Commander; simulated EEPROM tokens are not supported. For more information on manufacturing tokens, see *AN961*: *Bringing Up Custom Nodes for the EFR32MG and EFR32FG Families*.

6.4.1 Print Tokens

Command Line Syntax

```
$ commander tokendump --tokengroup <token group> [--token <token name>]
```

Command Line Input Example

```
$ commander tokendump --tokengroup znet --token TOKEN_MFG_STRING --token TOKEN_MFG_EMBER_EUI_64
```

Reads the selected tokens from the device and prints it to stdout.

Command Line Output Example

```
#
# The token data can be in one of three main forms: byte-array, integer, or string.
# Byte-arrays are a series of hexadecimal numbers of the required length.
# Integers are BIG endian hexadecimal numbers.
# String data is a quoted set of ASCII characters.
#
MFG_STRING      : "IoT_Inc"
# MFG_EMBER_EUI_64: F0B2030000570B00
DONE
```

6.4.2 Dump Tokens to File

This example works just like section 6.4.1 Print Tokens, except that the output is written to a file suitable for use with the `--tokenfile` option (`flash`, `verify`, and `convert` commands).

Command Line Syntax

```
$ commander tokendump --tokengroup <token group> [--token <token name>] --outfile <filename>
```

Command Line Input Example

```
$ commander tokendump --tokengroup znet --outfile tokens.txt
```

Reads all tokens from the device and outputs it to the file named tokens.txt.

Command Line Output Example

```
Writing tokens to tokens.txt...
DONE
```

6.4.3 Dump Tokens from Image File

If an input file is given to the `tokendump` command, the input is read from one or more files instead of reading from a device.

In this case, the `--device` option must be provided, because token locations can be different from one device family to another.

Command Line Syntax

```
$ commander tokendump <filename> --tokengroup <token group> --device <device> [--outfile <filename>]
```

Command Line Input Example

```
$ commander tokendump blink.hex --tokengroup znet --device EFR32MG1P --outfile tokens.txt
```

Command Line Output Example

```
Parsing file blink.hex...
DONE
```

6.4.4 Generate C Header Files from Token Groups

The `tokenheader` command generates a simple header file based on a custom token group. The generated header file contains pre-processor defines that specify the location and size of each token.

See section 4. EFR32 Custom Tokens for details on custom tokens.

Command Line Syntax

```
$ commander tokenheader --tokengroup <group name> --device <target device> <filename>
```

Command Line Input Example

```
$ commander tokenheader --tokengroup myapp --device EFR32MG1P233F256 my_tokens.h
```

Command Line Output Example

```
Writing token header file: my_tokens.h
DONE
```

6.5 Convert and Modify File Commands

The `convert` command performs image file conversion and manipulation. It supports the following actions:

- Converting between file formats.
- Merging several image files.
- Extracting subsets of images.
- Patching bytes.
- Setting token data.

The `convert` command can either write its output to a file or print it to standard out in human-readable format similar to the `readmem` command. When writing to a file, the file format is auto-detected based on the file extension used.

The `convert` command works off-line without any J-Link/debug connection. The command is device-agnostic, except when working with tokens or Ember Bootloader (EBL) files. In this case, you must use the `--device` option.

Command Line Syntax

```
$ commander convert [infile1] [infile2 …] [options]
```

6.5.1 Combine Two Files

Converts two files with different file formats into one specified output file.

Command Line Syntax

```
$ commander convert <filename> <filename> [--address <address>] --outfile <filename>
```

Command Line Input Example

```
$ commander convert blink.bin userpage.hex --address 0x0 --outfile blinkapp.s37
```

Combines blink.bin and userpage.hex to blinkapp.s37. The address option is used to set the start address of the .bin file, since bin files doesn't contain any addressing information. The address value is interpreted as a hexadecimal number. If more than one .bin file is supplied, the same start address is used for all. If this is not desirable, consider converting the bin files to s37 or hex in a separate preparation step.

Command Line Output Example

```
Parsing file blink.bin...
Parsing file userpage.hex...
Writing to blinkapp.s37...
DONE
```

6.5.2 Define Specific Bytes

Like the `flash` command, the `convert` command supports the `--patch` option for setting arbitrary unsigned integers at any address.

Command Line Syntax

```
$ commander convert [filename] --patch <address>:<data>[:length] [--outfile <filename>]
```

Command Line Input Example

```
$ commander convert blink.s37 --patch 0x0FE00000:0x12345:4 --outfile blink.hex
```

Converts blink.s37 to hex format, while simultaneously defining the first four bytes of the user page to 0x00012345. This works just like `flash blink.s37 --patch 0x0FE00000:0x12345:4`, but works against a file instead of writing to a device flash.

Command Line Output Example

```
Parsing file blink.s37...
Patching 0x0FE00000 = 0x00012345...
Writing to blink.hex...
DONE
```

6.5.3 Define Tokens

Like the `flash` command, the `convert` command supports the `--tokengroup`, `--token`, and `--tokenfile` options for setting token data while executing file conversion.

Command Line Syntax

```
$ commander convert [filename] --tokengroup <token group> [--tokenfile <filename>]
[--token <token name>
```

:<token data>] [--device <device>] [--outfile <filename>]

Command Line Input Example

```
$ commander convert blink.s37 --tokengroup znet --tokenfile tokens.txt --device EFR32MG1P --outfile blink.hex
```

Converts blink.s37 to hex format, while simultaneously defining the tokens defined in tokens.txt and on the command line. Works just like the corresponding options with `flash`, but writes to a file instead of flash.

Command Line Output Example

```
Parsing file blink.s37...
Writing to blink.hex...
DONE
```

6.5.4 Dump File Contents

Like the `readmem` command, the `convert` command will print its output in human-readable format to standard out if no output file is given. The value of the address option is interpreted as a hexadecimal number.

Command Line Syntax

```
$ commander convert <filename> [--address <bin file start address>]
```

Command Line Input Example

```
$ commander convert blink.bin --address 0x0 userpage.hex
```

If the `--outfile` option is not used, the data is printed to stdout instead of writing to file.

Command Line Output Example

```
Parsing file blink.bin...
Parsing file userpage.hex...
{address:  0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F}
00000000: 10 04 00 20 B5 0A 00 00 57 08 00 00 8B 0A 00 00
00000010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000020: 00 00 00 00 00 00 00 00 00 00 00 00 97 0A 00 00
00000030: 00 00 00 00 00 00 00 00 D1 0A 00 00 13 06 00 00
00000040: D3 0A 00 00 D5 0A 00 00 D7 0A 00 00 D9 0A 00 00
00000050: DB 0A 00 00 DD 0A 00 00 DF 0A 00 00 E1 0A 00 00
00000060: E3 0A 00 00 E5 0A 00 00 E7 0A 00 00 E9 0A 00 00
00000070: EB 0A 00 00 ED 0A 00 00 EF 0A 00 00 F1 0A 00 00
<shortened data for documentation>
00000ac0: C5 0A 00 00 C0 46 C0 46 C0 46 C0 46 FF F7 CA FF
00000ad0: FE E7 FE E7 FE E7 FE E7 FE E7 FE E7 FE E7 FE E7
00000ae0: FE E7 FE E7 FE E7 FE E7 FE E7 FE E7 FE E7 FE E7
00000af0: FE E7 FE E7 00 36 6E 01 00 80 00 00
{address:  0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F}
0fe00000: 45 23 01 00 FF FF FF FF FF FF FF FF FF FF FF FF
0fe00010: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
0fe00020: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
<shortened data for documentation>
0fe007e0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
0fe007f0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
DONE
```

6.5.5 Signing an Application for Secure Boot

Signs an application for use with a Secure Boot bootloader. For more information, see *UG266: Silicon Labs Gecko Bootloader User's Guide for GSDK 3.x and Lower* or *UG489: Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher*.

Command Line Syntax

```
$ commander convert <image file> --secureboot --keyfile <signing key> --outfile <signed image file>
```

Command Line Input Example

```
$ commander convert example.s37 --secureboot --keyfile mykey --outfile example-signed.s37
```

This example signs the image file named example.s37.

Command Line Output Example

```
Parsing file example.s37...
Image SHA256: 4591da45b6c40a424b81753001708061d5319197adec5188f4acc512cfb88e65
R = 8E417EB4CBC584218A8605FCF3E778F2A7810F2CAE190CB2EF4D0DF842829CC1
S = 5B095025FFD571699725107C4666C0B8B867370E990B73E74A0502CB9788DCA8
Writing to example-signed.s37...
DONE
```

6.5.6 Signing an Application for Secure Boot using a Hardware Security Module

Prepares an application for signing for use with a Secure Boot enabled bootloader using a Hardware Security Module (HSM). For more information, see *UG266: Silicon Labs Gecko Bootloader User's Guide for GSDK 3.x and Lower* or *UG489: Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher*.

Command Line Syntax

```
$ commander convert <image file> --secureboot --extsign --outfile <image file for external signing>
```

Command Line Input Example

```
$ commander convert example.s37 --secureboot --extsign --outfile example.s37.extsign
```

This example creates an output in the form that an HSM can create a signature over of the entire file. This signature can again be written to the file using the command described in 6.5.7 Signing an Application for Secure Boot Signing using a Signature Created by a Hardware Security Module.

Command Line Output Example

```
Parsing file example.s37...
Writing to example.s37.extsign...
DONE
```

6.5.7 Signing an Application for Secure Boot Signing using a Signature Created by a Hardware Security Module

Signs an application for use with a Secure Boot bootloader using a signature created by a Hardware Security Module (HSM). For more information, see *UG266: Silicon Labs Gecko Bootloader User's Guide for GSDK 3.x and Lower* or *UG489: Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher*.

Command Line Syntax

```
$ commander convert <image file> --secureboot --signature <signature from external signing> --outfile <signed image file>
```

Command Line Input Example

```
$ commander convert example.s37 --secureboot --signature example.s37.extsign.sig --outfile example-signed.s37
```

This example signs the image file example.s37 using a signature obtained from an HSM using the .extsign file generated in 6.5.6 Signing an Application for Secure Boot using a Hardware Security Module. The input file (example.s37) used with this function must be the same file as was used when generating the .extsign file in 6.5.6 Signing an Application for Secure Boot using a Hardware Security Module.

Command Line Output Example

```
Parsing file example.s37...
Parsing signature file example.s37.extsign.sig...
R = 8E417EB4CBC584218A8605FCF3E778F2A7810F2CAE190CB2EF4D0DF842829CC1
S = 5B095025FFD571699725107C4666C0B8B867370E990B73E74A0502CB9788DCA8
Writing to example-signed.s37...
Overwriting file: example-signed.s37...
DONE
```

6.5.8 Adding a CRC32 for Gecko Bootloader

This option adds a CRC32 (32-bit cyclic redundancy check) of the image that the Gecko Bootloader can use to ensure image integrity when Secure Boot is not used. This feature requires that an `ApplicationProperties_t` struct is present in the image. For more details on the `ApplicationProperties_t` struct, see *UG266: Silicon Labs Gecko Bootloader User's Guide for GSDK 3.x and Lower* or *UG489: Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher*.

Command Line Syntax

```
$ commander convert <image file> --crc --outfile <image file with CRC>
```

Command Line Input Example

```
$ commander convert example.s37 --crc --outfile example-crc.s37
```

This example adds a checksum to the image file named example.s37.

Command Line Output Example

```
Parsing file example.s37...
Appending CRC32 checksum...
Writing to example-crc.s37...
DONE
```

6.5.9 Signing an Application for Secure Boot using an Intermediary Certificate

Signs an application for use with a Secure Boot bootloader using an intermediary certificate. When using an intermediary certificate, the `ApplicationProperties_t` struct must be present in the image. For more information on the `ApplicationProperties_t` struct, see *UG266: Silicon Labs Gecko Bootloader User's Guide for GSDK 3.x and Lower* or *UG489: Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher*.

Secure Boot verification via an intermediary certificate is only supported on Series 2 EFR32 devices. Secure Boot must be enabled before signing a bootloader with an intermediary certificate. For more information about enabling Secure Boot, see 6.17.16 Write User Configuration.

There are two ways of signing the application:

- Providing the private keyfile corresponding to the public key embedded in the certificate directly.
- Preparing an application for signing with a Hardware Security Module (HSM) by generating an output in the form that an HSM can create a signature over the entire file. The signature can then be written to the file by passing it to Simplicity Commander as described below.

Note: Simplicity Commander does currently not support the generation of certificates for Secure Boot signing. This will be available in a future version of Simplicity Commander.

Command Line Syntax

```
$ commander convert <image file> --secureboot --certificate <certificate file> --keyfile <keyfile> --outfile <signed image file>
```

```
$ commander convert <image file> --secureboot --certificate <certificate file> --extsign --outfile <image file for external signing>
```

```
$ commander convert <image file> --secureboot --certificate <certificate file> --signature <signature> --outfile <signed image file>
```

Command Line Input Example

```
$ commander convert example.s37 --secureboot --certificate example_certificate.bin --keyfile public_certificate_key.pem --outfile example-signed.s37
```

This example signs the image file example.s37 using an intermediary certificate. The keyfile used to sign the application corresponds to the public key embedded in the certificate. Simplicity Commander always validates the key before signing the application.

Command Line Output Example

```
Parsing file example.s37...
Private key matches public key in certificate.
R = 137EA7A19F6100E1EFA5C185CA952B67137D0597F4A89C7543BC5A49A7A6681E
S = C537A833018C3A23CF1EBDBAB04559482B0B5333A7C21556E6B42EDA1D1A5102
Writing to example-signed.s37...
DONE
```

6.5.10 Add a Trust Zone Decryption Key

Adds an Advanced Encryption Standard (AES) encryption/decryption key to a bootloader image for TrustZone secure key storage. Requires Application Properties struct version 1.2 or higher.

Command Line Syntax

```
$ commander convert <image file> --aeskey <keyfile> --outfile <bootloader with decryption key>
```

Command Line Input Example

```
$ commander convert my_bootloader.s37 --aeskey my_key.txt --outfile my_bootloader_with_key.s37
```

Adds the decryption key *my_key.txt* to the bootloader image named *my_bootloader_with_key.s37*.

Command Line Output Example

```
Parsing file my_bootloader.s37...
Writing to my_bootloader_with_key.s37...
DONE
```

6.5.11 Extract Sections from ELF Files

Extract sections from an Executable and Linkable Format (ELF) file and convert into the specified output file. If neither the `--include-section` nor the `--exclude-section` option is used, Simplicity Commander will extract all `.text` sections, as well as sections of type `progbits` with address not equal to 0x0.

Command Line Syntax

```
$ commander convert <filename>  [--include-section <section> --exclude-section <section>] --outfile <filename>
```

Command Line Input Example

```
$ commander convert application.out --include-section text_apploader  --outfile apploader.s37
```

Creates an S-record file from the `text_apploader` section of an ELF file.

Command Line Output Example

```
Parsing file application.out...
Writing to apploader.s37...
DONE
```

6.6 EBL Commands

6.6.1 Print EBL Information

Parses and prints EBL information from the specified .ebl file.

Command Line Syntax

```
$ commander ebl print <filename>
```

Command Line Input Example

```
$ commander ebl print example.ebl
```

Command Line Output Example

```
Found EBL Tag = 0x0000, length  140, [EBL Header]
  Version:      0x0201
  Signature:    0xE350 (Correct)
  Flash Addr:   0x00004000
  AAT CRC:      0x53BC1F4E
  AAT Size:     128 bytes
    HalAppBaseAddressTableType
      Top of Stack:        0x20006980
      Reset Vector:        0x000121F9
      Hard Fault Handler: 0x00012125
      Type:                0x0AA7
      HalVectorTable:      0x00004100
    Full AAT Size:     172
    Ember Version:     5.7.0.0
    Ember Build:       0
    Timestamp:         0x561E581F (Wed Oct 14, 2015 13:26:55 UTC [+0100])
    Image Info String:''
    Image CRC:         0x2ACE0C5B
    Customer Version: 0x00000000
    Image Stamp:       0xF4271F50BA2E2FBA
Found EBL Tag = 0xFD03, length 1924, [Erase then Program Data]
  Flash Addr: 0x00004080
Found EBL Tag = 0xFD03, length 2052, [Erase then Program Data]
  Flash Addr: 0x00004800
(32 additional tags of the same type and length.)
Found EBL Tag = 0xFD03, length 1772, [Erase then Program Data]
  Flash Addr: 0x00015000
Found EBL Tag = 0xFC04, length    4, [EBL End Tag]
  CRC: 0xDBC82DA5
The CRC of this EBL file is valid (0xdebb20e3)
File has 0 bytes of end padding.
Calculated image stamp matches value found in AAT.
DONE
```

6.6.2 EBL Key Generation

Generates a keyfile to be used for encryption or decryption and outputs the keyfile to the specified filename.

Command Line Syntax

```
$ commander ebl keygen --type aes-ccm --outfile <filename>
```

Command Line Input Example

```
$ commander ebl keygen --type aes-ccm --outfile key.txt
```

Command Line Output Example

```
Using /dev/random for random number generation
Gathering sufficient entropy... (may take up to a minute)...
DONE
```

6.6.3 EBL File Creation

Creates an EBL file from an application image and writes the output to the specified filename. Can optionally encrypt the EBL file using a keyfile generated by the `ebl keygen` command.

Command Line Syntax

```
$ commander ebl create <eblfile> --app <filename> --device <part number> [--encrypt <keyfile>]
```

Command Line Input Example

```
$ commander ebl create app.ebl.encrypted --app example.s37 --device EFR32F256 --encrypt key.txt
```

Command Line Output Example

```
Parsing file example  .s37...
Parse .s37 format for flash
Flash Usage:
  Reserved for Bootloader:                0x00000000-0x00003fff (16384 bytes)
  CODE and Tables:                        0x00004000-0x00014ddb (69084 bytes)
  CONST and INITC:                        0x00014ddc-0x000184ab (14032 bytes)
  Available for future use:               0x000184ac-0x0003dfff (154452 bytes)
  Reserved for SIMEE:                     0x0003e000-0x0003ffff (8192 bytes)

Usage Summary:
  262144 total bytes Flash, 107692 used, 154452 available

Setting AAT timestamp to current time: 0x586e1ec9
Create ebl image file
Wrote image stamp into AAT.
Encrypting EBL...
Unencrypted input file: ebl_plaintext_ux8544.ebl
Encrypt output file:     app.ebl.encrypted
Randomly generating nonce
Using /dev/random for random number generation
Gathering sufficient entropy... (may take up to a minute)...
Created ENCRYPTED ebl image file
DONE
```

6.6.4 EBL File Parsing

Parses an EBL file and writes the application image to the specified filename. Optionally decrypts an encrypted EBL file. The keyfile must be the same as was used for encrypting the encrypted EBL file.

Command Line Syntax

```
$ commander ebl parse <ebl filename> --app < filename> --device <part number> [--decrypt <key filename>]
```

Command Line Input Example

```
$ commander ebl parse example.ebl.encrypted --app app.s37 --device EFR32F256 --decrypt ../aeskey
```

Command Line Output Example

```
Unencrypted output file:  ebl_plaintext_L10567.ebl
Encrypt input file:      example.ebl.encrypted
MAC matches. Decryption successful.
Created DECRYPTED ebl image file
Parse .ebl format for flash
Create image file
Writing application to app.s37...
DONE
```

6.6.5 Memory Usage Information from AAT

For applications containing an Application Address Table (AAT), Simplicity Commander can analyze the memory usage of the application. The AAT is included in Zigbee applications.

RAM usage is only available for EM3xx applications. Applications built for EFR32 can only be analyzed for flash usage.

Command Line Syntax

```
$ commander ebl aat-usageinfo <filename> --device <part number>
```

Command Line Input Example

```
$ commander ebl aat-usageinfo example.s37 --device EM357
```

Command Line Output Example

```
Parse .s37 format for flash

Approximate Usage Information:
RAM Usage:
  APPLICATION_CONFIGURATION_HEADER usage: 0x20000000-0x20000fc3 (4036 bytes)
  Available for future use:               0x20000fc4-0x2000195f (2460 bytes)
  Call Stack:                             0x20001960-0x200022bf (2400 bytes)
  Globals and Statics:                    0x200022c0-0x20002fe8 (3369 bytes)
  Alignment Overhead:                     0x20002fe9-0x20002fef (7 bytes)
  NO_INIT and Debug Channel:              0x20002ff0-0x20002fff (16 bytes)
Flash Usage:
  Reserved for Bootloader:                0x08000000-0x08001fff (8192 bytes)
  CODE and Tables:                        0x08002000-0x08011cdf (64736 bytes)
  CONST and INITC:                        0x08011ce0-0x08014263 (9604 bytes)
  Available for future use:               0x08014264-0x0802dfff (105884 bytes)
  Reserved for SIMEE:                     0x0802e000-0x0802ffff (8192 bytes)

Usage Summary:
  12288 total bytes RAM, 9828 used, 2460 available
  196608 total bytes Flash, 90724 used, 105884 available

DONE
```

6.7 GBL Commands

6.7.1 GBL File Creation

Creates a Gecko Bootloader (GBL) file from an application image and writes the output to the specified filename. Can optionally encrypt the GBL file using a keyfile generated by the `gbl keygen` command.

Command Line Syntax

```
$ commander gbl create <gblfile> --app <filename> [--encrypt <keyfile>]
```

Command Line Input Example

```
$ commander gbl create app.gbl.encrypted --app example.s37 --encrypt key.txt
```

Command Line Output Example

```
Parsing file example.s37...
Initializing GBL file...
Adding application to GBL...
Encrypting GBL...
Writing GBL file app.gbl.encrypted...
DONE
```

6.7.2 GBL File Creation with Compression

Creates a compressed Gecko Bootloader (GBL) file from an application image and writes the output to the specified filename. Can optionally encrypt the GBL file using a keyfile generated by the `gbl keygen` command.

The currently supported compression algorithms are `lz4` and `lzma`. The bootloader on the targeted devices must support decompressing the selected compression type.

Command Line Syntax

```
$ commander gbl create <gblfile> --app <filename> --compress <compression algorithm> [--encrypt <keyfile>]
```

Command Line Input Example

```
$ commander gbl create app.gbl --app example.s37 --compress lz4
```

Command Line Output Example

```
Parsing file example.s37...
Initializing GBL file...
Adding application to GBL...
Compressing using lz4...
Writing GBL file app.gbl...
DONE
```

6.7.3 Create a GBL File for Bootloader Upgrade

Creates a Gecko Bootloader (GBL) file from a bootloader image and writes the output to the specified bootloader image filename. For more information, see *UG266: Silicon Labs Gecko Bootloader User's Guide for GSDK 3.x and Lower* or *UG489: Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher*.

Command Line Syntax

```
$ commander gbl create <gblfile> --bootloader <bootloader image file> [--encrypt <keyfile>]
```

Command Line Input Example

```
$ commander gbl create bootloader.gbl --bootloader bootloader.s37
```

Command Line Output Example

```
Initializing GBL file...
Adding bootloader to GBL...
Writing GBL file bootloader.gbl...
DONE
```

6.7.4 Creating a GBL File for Secure Element Upgrade

The Secure Element on EFR32xG21 devices can be upgraded using a Secure Element upgrade binary provided by Silicon Labs. This command creates a GBL file containing a Secure Element upgrade file and writes the output to the specified GBL filename. For more information, see *UG266: Silicon Labs Gecko Bootloader User's Guide for GSDK 3.x and Lower* or *UG489: Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher*.

Command Line Syntax

```
$ commander gbl create <gblfile> --seupgrade <secure element upgrade file> --app <application image>
```

Command Line Input Example

```
$ commander gbl create se-upgrade.gbl --seupgrade secure-element-1.0.0.seu --app myapp.s37
```

Command Line Output Example

```
Parsing file myapp.s37...
Initializing GBL file...
Adding application to GBL...
Adding Secure Element upgrade image to GBL...
Writing GBL file se-upgrade.gbl...
DONE
```

6.7.5 Creating a Signed and Encrypted GBL Upgrade Image File from an Application

Creates a GBL file, signs the GBL file, and encrypts the GBL file. For more information, see *UG266: Silicon Labs Gecko Bootloader User's Guide for GSDK 3.x and Lower* or *UG489: Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher*.

Command Line Syntax

```
$ commander gbl create <gblfile> --app <app image file> --sign <signing key> [--encrypt <encryption key>]
```

Command Line Input Example

```
$ commander gbl create example.gbl --app example.s37 --sign ecdsakey --encrypt aeskey
```

Command Line Output Example

```
Parsing file example.s37...
Initializing GBL file...
Adding application to GBL...
Encrypting GBL...
Signing GBL...
Image SHA256: 74b126bdbad680470487e32d7d7b3ec7f12b15d9988e028b26c2dd54f81dcfb7
R = 055A23A44CDEDA34506EE72F4530FE174CFC85F48933C1379C1360F8BC1AA75B
S = 1C9EF6C3F5CAA0D5B92ECC2569E4A8251F8561DAF52DE54D3E59591A5001B9EA
Writing GBL file example.gbl...
DONE
```

6.7.6 Creating a Partial Signed and Encrypted GBL Upgrade File for Use with a Hardware Security Module

It is often not desirable to keep the private key used for signing locally on the computer that creates the GBL images. A good way to increase security is to use a Hardware Security Module (HSM) to generate the actual signatures. Simplicity Commander supports using a three-step process:

1. Create a partial GBL file for external signing using Simplicity Commander.
2. Create an Elliptic Curve Digital Signature Algorithm (ECDSA) signature of the partial GBL file using an HSM.
3. Use Simplicity Commander to sign the partial GBL file using the signature from the HSM, and create a complete GBL file.

Step 1 is described in this section. Step 2 is specific to the HSM you are using. Step 3 is described in 6.7.7 Creating a Signed GBL File Using a Hardware Security Module. For more information, see *UG266: Silicon Labs Gecko Bootloader User's Guide for GSDK 3.x and Lower* or *UG489: Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher*.

Command Line Syntax

```
$ commander gbl create <output partial GBL file for external signing> --app <app image file>
--extsign [--encrypt <encryption key>]
```

Command Line Input Example

```
$ commander gbl create example.gbl.extsign --app example.s37 --extsign --encrypt aeskey
```

Command Line Output Example

```
Parsing file example.s37...
Initializing GBL file...
Adding application to GBL...
Encrypting GBL...
Preparing GBL for external signing...
Writing GBL file example.gbl.extsign...
DONE
```

6.7.7 Creating a Signed GBL File Using a Hardware Security Module

Creates a signed GBL file from a partial GBL file and an ECDSA signature file in Distinguished Encoding Rules (DER) format generated as described in 6.7.6 Creating a Partial Signed and Encrypted GBL Upgrade File for Use with a Hardware Security Module . For more information, see *UG266: Silicon Labs Gecko Bootloader User's Guide for GSDK 3.x and Lower* or *UG489: Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher*.

Silicon Labs recommends that you use the `--verify` option with the public key corresponding to the private key used by the HSM to ensure the integrity of the generated GBL file.

Command Line Syntax

```
$ commander gbl sign <partial GBL file for external signing> --signature <signature from HSM>
[--verify <public key file>] --outfile <signed GBL file>
```

Command Line Input Example

```
$ commander gbl sign example.gbl.extsign --signature example.gbl.extsign.sig --verify ecdsakey.pub
--outfile example-signed.gbl
```

Command Line Output Example

```
Reading GBL data from example.gbl.extsign...
Parsing signature file example.gbl.extsign.sig...
R = 2E73426A1052E12BFFFEFBA9BE2AA50CEA815B630C3CA878494EEF26088A5673
S = C218596DB9958AB30924B516953D2E5107644963B4CA128072AC965BE5C2992D
Writing signature to GBL...
Verifying GBL...
Image SHA256: 4d7325b09ade0ea272eb9895096c8137b18451f694a4eca9a5782f5c08dea03a
Q_X: 60BA97B850291456217C2149061AA344B32BBFB69A91A94BBF2F274744308D39
Q_Y: 41927DA5DB171E1C723C6B59C2BC88EDFF5A37014B0473775BA5B15921686ECA
R = 2E73426A1052E12BFFFEFBA9BE2AA50CEA815B630C3CA878494EEF26088A5673
S = C218596DB9958AB30924B516953D2E5107644963B4CA128072AC965BE5C2992D
Writing GBL file example-signed.gbl...
DONE
```

6.7.8 GBL File Parsing

Parses a Gecko Bootloader (GBL) file and writes the application image to the specified filename. Optionally decrypts an encrypted GBL file. The keyfile must be the same as was used for encrypting the encrypted GBL file.

Command Line Syntax

```
$ commander gbl parse <gbl filename> --app < filename> [--decrypt <key filename>]
```

Command Line Input Example

```
$ commander gbl parse example.gbl.encrypted --app app.s37 --decrypt key.txt
```

Command Line Output Example

```
Reading GBL data...
Decrypting GBL...
Reading application...
Writing application to app.s37...
DONE
```

6.7.9 GBL Key Generation

This command is deprecated. Please see 6.18.1 Key Generation for more information about key generation.

6.7.10 Generating a Signing Key

This command is deprecated. Please see 6.18.2 Generating a Signing Key for more information about generating a signing key.

6.7.11 Generate a Signing Key Using a Hardware Security Module

This command is deprecated. Please see 6.18.3 Key to Token for more information about generating a signing key using a hardware security module.

6.7.12 Creating a Signed GBL File Using a Hardware Security Module

Creates a signed GBL file from a partial GBL file and an ECDSA signature file in Distinguished Encoding Rules (DER) format generated as described in 6.7.6 Creating a Partial Signed and Encrypted GBL Upgrade File for Use with a Hardware Security Module . For more information, see *UG266: Silicon Labs Gecko Bootloader User's Guide for GSDK 3.x and Lower* or *UG489: Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher*.

Silicon Labs recommends that you use the `--verify` option with the public key corresponding to the private key used by the HSM to ensure the integrity of the generated GBL file.

Command Line Syntax

```
$ commander gbl sign <partial GBL file for external signing> --signature <signature from HSM>
[--verify <public key file>] --outfile <signed GBL file>
```

Command Line Input Example

```
$ commander gbl sign example.gbl.extsign --signature example.gbl.extsign.sig --verify ecdsakey.pub
--outfile example-signed.gbl
```

Command Line Output Example

```
Reading GBL data from example.gbl.extsign...
Parsing signature file example.gbl.extsign.sig...
R = 2E73426A1052E12BFFFEFBA9BE2AA50CEA815B630C3CA878494EEF26088A5673
S = C218596DB9958AB30924B516953D2E5107644963B4CA128072AC965BE5C2992D
Writing signature to GBL...
Verifying GBL...
Image SHA256: 4d7325b09ade0ea272eb9895096c8137b18451f694a4eca9a5782f5c08dea03a
Q_X: 60BA97B850291456217C2149061AA344B32BBFB69A91A94BBF2F274744308D39
Q_Y: 41927DA5DB171E1C723C6B59C2BC88EDFF5A37014B0473775BA5B15921686ECA
R = 2E73426A1052E12BFFFEFBA9BE2AA50CEA815B630C3CA878494EEF26088A5673
S = C218596DB9958AB30924B516953D2E5107644963B4CA128072AC965BE5C2992D
Writing GBL file example-signed.gbl...
DONE
```

6.7.13 Create a GBL File from an ELF File

Creates a Gecko Bootloader (GBL) file from an Executable and Linkable Format (ELF) file and writes the output to the specified file. If neither the `--include-section` nor the `--exclude-section` option is used, Simplicity Commander will include all sections that appear to be part of the application.

Command Line Syntax

```
$ commander gbl create <gblfile> --app <application image file> [--include-section <section> --exclude-section
<section>]
```

Command Line Input Example

```
$ commander gbl create app.gbl --app app.out --exclude-section text_apploader --exclude-section text_signature
```

Creates a GBL file containing an ELF application, excluding the `text_apploader` and `text_signature` sections from the application.

Command Line Output Example

```
Parsing file app.out...
Initializing GBL file...
Adding application to GBL...
Encrypting GBL...
Writing GBL file app.gbl.encrypted...
DONE
```

6.7.14 Create an Encrypted GBL File with an Unencrypted Secure Element Upgrade File

Creates an encrypted Gecko Bootloader (GBL) file containing an unencrypted Secure Element upgrade file and then writes the output to the specified GBL file.

Command Line Syntax

```
$ commander gbl create <gblfile> --seupgrade <secure element upgrade file> --seunencrypted --app <application
image> --encrypt <AES key file>
```

Command Line Input Example

```
$ commander gbl create se-upgrade.gbl --seupgrade secure-element.seu --seunencrypted --app myapp.s37 --encrypt
aes-key.txt
```

Creates an encrypted GBL file with a Secure Element upgrade file outside the encrypted area of the file.

Command Line Output Example

```
Parsing file myapp.s37...
Initializing GBL file...
Adding application to GBL...
Adding Secure Element upgrade image to GBL...
Encrypting GBL...
Writing GBL file se-upgrade.gbl...
DONE
```

6.7.15 Create a GBL File with Version Dependencies

Any version dependencies between application, bootloader, and secure element upgrade files in a Gecko Bootloader (GBL) file may be resolved using the `--dep-app`, `--dep-boot`, and `--dep-se` options.

Command Line Syntax

```
$ commander gbl create <gblfile> --seupgrade <secure element upgrade file> --app <application image> --dep-app
<statement:version> --dep-se <staetment:version> --dep-boot <statement:version>
```

Dependency Statement

The dependency statement may be one of the following:

Simplicity Commander Input	Statement
g	Greater than
geq	Greater than or equal
eq	Equal
leq	Less than or equal
l	Less than

The `--dep-app` option takes an uint32 as version input, while the `--dep-se` and `--dep-boot` options take the version input in the format major.minor.patch.

Command Line Input Example

```
$ commander gbl create se-upgrade.gbl --app myapp.s37 --seupgrade secure-element.seu --bootloader my-
bootloader.s37 --dep-app geq:0x01020002 --dep-boot l:0.5.7 --dep-se g:1.2.3
```

Creates a GBL where the application version must be greater than or equal to version 0x01020002, bootloader version must be less than version 0.5.7, and secure element upgrade version must be greater than version 1.2.3.

Command Line Output Example

```
Parsing file myapp.s37
Initializing GBL file...
Setting version dependency of Application to >= 0x00120002
Setting version dependency of Bootloader to < 0x00050007
Setting version dependency of SE upgrade image to > 0x00010203
Adding version dependencies to GBL...
Adding application to GBL...
Adding bootloader to GBL...
Adding Secure Element upgrade image to GBL...
Writing GBL file se-upgrade.gbl
DONE
```

6.8 Kit Utility Commands

6.8.1 Firmware Upgrade

Updates the application running on the board controller on the kit to a new version provided in an .emz file by Silicon Labs.

Command Line Syntax

```
$ commander adapter fwupgrade --serialno <J-Link serial number> <filename>
```

Command Line Input Example

```
$ commander adapter fwupgrade -s 440050184 S1015B_wireless_stk_firmware_package_0v14p0b435.emz
```

Command Line Usage Output

```
Checking manifest...
Checking if target is in bootloader...
Waiting for kit to restart...
Package is usable
Deleting previous firmware...
Installing files...
Resetting target...
Waiting for kit to restart...
Finished!
DONE
```

6.8.2 Kit Information Probe

Retrieves information about a connected kit. Lists information about the kit part number and name, connected boards, and firmware version.

The options `--kit`, `--boards`, and `--firmware` limit the output to just kit information, board list, or firmware information, respectively.

The `VCOM Port` line informs which virtual COM port name the kit has been assigned by the operating system. On Windows this is on the form COM<number>. On Linux and macOS, the name corresponds to a special file in the `/dev/` folder. E.g. `VCOM Port: ttyACM0` indicates that the serial port is available at `/dev/ttyACM0`. This line is not always available, and may be omitted from the output.

Command Line Syntax

```
$ commander adapter probe --serialno <J-Link serial number> [--kit] [--boards] [--firmware]
```

Command Line Input Example

```
$ commander adapter probe --serialno 440050184
```

Command Line Usage Output

```
Kit Information:
=======================================
Kit Name        : EFR32 Mighty Gecko 2400/915 MHz Dual Band Wireless Starter Kit
Kit Part Number : WSTK6002A Rev. A00
J-Link Serial   : 440050184
Debug Mode      : MCU
AEM Supported   : 1
VCOM Supported  : 1
IP Supported    : 1
VCOM Port       : COM3
Firmware Information:
=======================================
FW Version      : 0v14p0b435
Board List:
=======================================
Name            : Wireless Starter Kit Mainboard
Part Number     : BRD4001A Rev. A01
Serial Number   : 152607557
Name            : EFR32MG 2400/915 MHz 19.5 dBm Dual Band Radio Board
Part Number     : BRD4150B Rev. B00
Serial Number   : 151300035
DONE
```

6.8.3 Adapter Reset Command

This command resets the adapter itself, causing a restart. The `adapter reset` command is usually not required during normal operation.

An error about “Communication timed out” may occur because the adapter sometimes restarts before it has time to reply to the command.

Command Line Syntax

```
$ commander adapter reset
```

Command Line Input Example

```
$ commander adapter reset
```

Command Line Output Example

```
Communication timed out: Requested 76 bytes, received 0 bytes !
DONE
```

6.8.4 Adapter Debug Mode Command

This command sets or reads the current debug mode of the adapter. The supported debug modes are typically IN, OUT, MCU, and OFF in addition to MINI for Wireless Pro Kits and TARGET for Development Kits. See the *Quick Start Guide* for your kit for a description of the debug modes it supports.

Command Line Syntax

```
$ commander adapter dbgmode [mode]
```

Command Line Input Example

```
$ commander adapter dbgmode MCU
```

Command Line Output Example

```
Setting debug mode to MCU...
DONE
```

6.8.5 List Adapter IP Configuration Command

The `adapter ip` command gets or sets the IP configuration of the adapter. With no options, the current configuration is retrieved and displayed.

Command Line Syntax

```
$ commander adapter ip
```

Command Line Input Example

```
$ commander adapter ip
```

Command Line Output Example

```
IP Address: 192.168.0.5/24
Gateway   : 192.168.0.1
DNS Server: 192.168.0.1
DONE
```

6.8.6 Adapter DHCP Command

This command sets up the adapter to use DHCP to automatically retrieve IP, gateway and DNS addresses. This is the default configuration. After enabling DHCP, the adapter must be restarted for the change to take effect.

Command Line Syntax

```
$ commander adapter ip --dhcp
```

Command Line Input Example

```
$ commander adapter ip --dhcp
```

Command Line Output Example

```
Enabling DHCP. The adapter must be restarted to acquire a new IP address.
DONE
```

6.8.7 Set Static IP Configuration Command

This command sets the IP address of the adapter in Classless Inter-Domain (CIDR) notation.

Command Line Syntax

```
$ commander adapter ip --addr <IP address/prefix> [--gw <gateway address>] [--dns <dns server address>]
```

Command Line Input Example

```
$ commander adapter ip --addr 192.168.1.5/24 --gw 192.168.1.1 --dns 192.168.1.1
```

Command Line Output Example

```
Setting IP Address: 192.168.1.5/24
Setting gateway: 192.168.1.1
Setting DNS server: 192.168.1.1
DONE
```

6.9 Device Erase Commands

6.9.1 Erase Chip

Executes a mass erase for devices where it is supported. On EFM32G and EFM32TG, all pages are erased instead, which is significantly slower.

Command Line Syntax

```
$ commander device masserase
```

Command Line Usage Output

```
Erasing chip...
DONE
```

6.9.2 Erase Region

Erases a named region. For more information on the `--region` option, see section 6.2 Flash Verification Command.

Command Line Syntax

```
$ commander device pageerase --region <@region>
```

Command Line Input Example

```
$ commander device pageerase --region @userdata
```

Command Line Output Example

```
Erasing range 0x0fe00000 - 0x0fe00800
DONE
```

6.9.3 Erase Pages in Address Range

Erases all flash pages affected by the given memory range. If the given range doesn't match page boundaries, it will be extended to always erase entire pages.

Command Line Syntax

```
$ commander device pageerase --range <startaddress>:<endaddress>
```

Command Line Input Example

```
$ commander device pageerase --range 0x200:0x6000
```

Erases all flash pages 0 to 11 or 0x0000 to 0x5FFF (assuming a page size of 2 kB).

Command Line Output Example

```
Erasing range 0x00000000 - 0x00006000
DONE
```

6.10 Device Lock and Protection Commands

6.10.1 Debug Lock

Locks access to the debug interface of the device. This feature is only supported on EFM32 and EFR32 devices. The `--debug enable` option is no longer required as of Simplicity Commander version 1.8.

Command Line Syntax

```
$ commander device lock [--debug enable]
```

Command Line Usage Output

```
Locking debug access...
DONE
```

6.10.2 Debug Unlock

Unlocks access to the debug interface of the device. This triggers a mass erase if the device was locked before.

This feature is only supported on EFM32 and EFR32 devices.

Command Line Syntax

```
$ commander device lock --debug disable
```

Command Line Usage Output

```
ERROR: Could not get MCU information
Removing all locks/protection...
Unlocking debug access (triggers a mass erase)...
DONE
```

In Simplicity Commander version 1.8 an alternative command syntax was introduced.

Command Line Syntax

```
$ commander device unlock
```

Command Line Usage Output

```
Unlocking debug access (triggers a mass erase)...
Chip successfully unlocked.
DONE
```

6.10.3 Write Protect Flash Ranges

Protects all flash pages affected by the given memory range from any writes or erases. The available granularity of flash write protection is device-dependent. Consult the device reference manual for details. For EFM32 and EFR32 devices, for example, the write protect feature operates on flash pages. On EM3xx devices, this works on 8 kB or 16 kB blocks.

For all devices, if the given range doesn't match the block size supported by the device, it will be extended to always protect entire regions.

Command Line Syntax

```
$ commander device protect --write --range <startaddress>:<endaddress>
```

Command Line Input Example

```
$ commander device protect --write --range 0x0:0x4000
```

Protects all flash pages in the first 16 kB from being erased or written to. Useful for protecting a bootloader from being modified by buggy application code, for example.

Command Line Output Example

```
Write protecting range 0x00000000 - 0x00004000
DONE
```

6.10.4 Write Protect Flash Region

Protects all flash pages in the named region from being written to or erased.

Command Line Syntax

```
$ commander device protect --write --region @<region>
```

Command Line Input Example

```
$ commander device protect --write --region @mainflash
```

Protects the entire main flash from being written to or erased.

Command Line Output Example

```
Write-protecting all pages in main flash.
DONE
```

6.10.5 Disable Write Protection

Disables write protection for all pages.

Command Line Syntax

```
$ commander device protect --write --disable
```

Command Line Output Example

```
Disabling all write protection...
DONE
```

6.11 Device Utility Commands

6.11.1 Device Information Command

Shows detailed information about the target device.

Command Line Syntax

```
$ commander device info
```

Command Line Usage Output

```
Part Number     : EFR32MG1P233F256GM48
Die Revision    : A0
Production Ver : 0
Flash Size      : 256 kB
SRAM Size       : 32 kB
Unique ID       : 000b57000003b2f0
DONE
```

6.11.2 Device Reset Command

Resets a device using a pin reset.

Command Line Syntax

```
$ commander device reset
```

Command Line Usage Output

```
Resetting chip...
DONE
```

6.11.3 Device Recovery Command

On EFM32 and EFR32 devices, this command tries to recover a device that has lost debug access due to misconfiguration of clocks, GPIO pins, or similar. Recovery is not supported on all devices, and in some cases requires the kit corresponding to the device you want to recover, for example, an EFM32TG STK to recover an EFM32TG device.

On EM3xx devices, this command can be used to recover from option byte failure.

Command Line Syntax

```
$ commander device recover
```

Command Line Usage Output

```
Recovering "bricked" device...
DONE
```

6.11.4 Device Z-Wave QR Code Command

The Z-Wave QR code command is used to read out the QR code from all Z-Wave devices. The QR code is 90 bytes, displayed as ASCII characters, and stored in the TOKEN_MFG_ZW_QR_CODE manufacturing token.

The QR code is generated in the chip during initialization. When the QR code is correctly initialized, the value of the manufacturing token TOKEN_MFG_ZW_INITIALIZED is changed from 0xFF to 0x00. The optional `--timeout` option is used to indicate how long Simplicity Commander should wait for the QR code to be initialized. If no time is given, the default is 5000 ms.

Command Line Syntax

```
$ commander device zwave-qrcode [--timeout <timeout in ms>]
```

Command Line Input Example

```
commander device zwave-qrcode --timeout 5000
```

Command Line Usage Output

```
QR code: 900132782003515253545541424344453132333435212223242500100435301537022065520001000000300578
DONE
```

6.12 External SPI Flash Commands

Simplicity Commander supports reading, writing, and erasing data on an external SPI flash on a limited selection of boards and devices. The following configurations are currently supported:

- The integrated SPI flash on EFR32MG1x632 and EFR32MG1x732 devices
- The MX25 SPI flash on EFR32 radio boards

6.12.1 Erase External SPI Flash Command

Use this command to erase data on an external flash. By default, the erased range is read back to verify that it was actually erased. This blank check can be disabled by including the `--noverify option`.

The `extflash erase` command always erases complete sectors. Any sector overlapping with the range provided will be erased. All currently supported flash devices have a sector size of 4096 bytes. For example, erasing with option `--range 0xE00:0x1100` will effectively erase the first two sectors (equivalent to `--range 0x0:0x2000`).

Command Line Syntax

```
$ commander extflash erase --range <range expression> [--noverify]
```

Command Line Input Example

```
$ commander extflash erase --range 0x1000:0x3000
```

Command Line Output Example

```
Erasing 8192 bytes from 0x00001000 on external flash.
Resetting target...
Uploading flashloader...
Erasing external flash...
Verifying written data...
Waiting for flashloader to become ready...
Reading from external flash...
DONE
```

6.12.2 Read External SPI Flash Command

Use this command to read from external flash.

Command Line Syntax

```
$ commander extflash read --range <range expression>
```

Command Line Input Example

```
$ commander extflash read --range 0x0:+0x20
```

Command Line Output Example

```
Reading 32 bytes from 0x00002000 on external flash.
Resetting target...
Uploading flashloader...
Waiting for flashloader to become ready...
Reading from external flash...
{address:  0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F}
00002000: 48 65 6C 6C 6F 20 57 6F 72 6C 64 21 0A FF FF FF
00002010: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
DONE
```

6.12.3 Write External SPI Flash Command

Use this command to write to external flash.

Any existing content in the affected flash sectors will be erased before writing.

In contrast to the `flash` command for internal flash, the `extflash write` command always flashes the raw content of the given file. If the address option is given the value is interpreted as a hexadecimal number. If, for example, an S-record file is provided, the ASCII content of the file is written; the S-record format is not parsed and written to the addresses specified in the file.

Command Line Syntax

```
$ commander extflash write <filename> --address <start address>
```

Command Line Input Example

```
$ commander extflash write myfile.txt --address 0x2000
```

Command Line Output Example

```
Flashing 13 bytes to 0x00002000 on external flash.
Resetting target...
Uploading flashloader...
Waiting for flashloader to become ready...
Erasing external flash...
Writing to external flash...
Verifying written data...
Waiting for flashloader to become ready...
Reading from external flash...
DONE
```

6.13 Advanced Energy Monitor Commands

Simplicity Commander supports reading and logging current measurement data from the Advanced Energy Monitor (AEM) of the adapter.

6.13.1 Measure Average Current in a Time Window

The `aem measure` command measures the average current in a time window. The `--windowlength` is in milliseconds (ms) and is defined as the duration where current samples will be measured and averaged. The default is 100 ms if no time is given. Ongoing measurements can be terminated by pressing CTRL+C.

Command Line Syntax

```
$ commander aem measure [--windowlength <time in ms>]
```

Command Line Input Example

```
$ commander aem measure --windowlength 200
```

Command Line Output Example

```
Averaged over 200 ms:
Current [mA]: 5.359
Power [mW]  : 17.763
Voltage [V] : 3.314
DONE
```

6.13.2 Log Current Measurements as Time Series Data

The `aem dump` command continuously measures the current and logs the measurement data (voltage and current). If `--outfile` is provided, the data is logged in the specified output file, otherwise the data is streamed to the terminal window. If `--duration` is provided, the logging will stop after the specified time, otherwise the logging will continue indefinitely. In both cases, the logging may be terminated at any time by pressing CTRL+C. The `--noheader` option can be passed to omit the column header from being included in the output file. The options described here are also applicable for use with the other options related to the aem dump command, but will be omitted in the next sections for the sake of brevity.

If the `--datarate` option is provided, the rate of which the current measurements are logged is set to the specified rate. This will collect samples over a time equal to the reciprocal of the provided data rate, and average these samples before storing the data in the output log (terminal or specified output file). The data rate must be equal to or larger than 1 Hz, and also equal to or less than the AEM sampling rate of the adapter in question. If `--datarate` is not provided, the command will default to the AEM sampling rate of the adapter.

The output file must be of either .txt or .csv format.

Command Line Syntax

```
$ commander aem dump [--outfile <filename> --datarate <rate in Hz> --duration <time in s> --noheader]
```

Command Line Input Example

```
$ commander aem dump --outfile output.csv --duration 10
```

This command will log AEM measurements for 10 seconds and store the data in 'output.csv', including a column header.

Command Line Output Example

```
Logging...
Send CTRL+C to abort.
Measurements written to file: 10000
Measurements written to file: 20000
<shortened data for documentation>
Measurements written to file: 90000
Measurements written to file: 100000

Closed file 'output.csv',
100000 measurements written to file.
DONE
```

6.13.3 Start Logging on Trigger Event

Trigger parameters can be set up to start logging when either the current is above a certain current threshold (in mA) or below a certain threshold. The `--triggerabove` and `--triggerbelow` options can be included to specify the type of triggering event. Only one of these options can be used when issuing this command.

Using the `--triggertimeout` option, you can specify how long the command should wait for the trigger event before timing out. Additionally, the `--pretrigger` option may be included to allow for logging of the measurements in the specified time window leading up to the trigger event. If the actual trigger event occurs before the length of the pre-trigger time has passed (after the command was executed), the actual included pre-trigger data will be the data that was collected from the execution of the command until the trigger event has occurred.

Note: Pay attention to the units used with the following options.

Command Line Syntax

```
$ commander aem dump [--triggerabove <current in mA> --triggerbelow <current in mA> --triggertimeout <timeout
in s> --pretrigger <time in ms>]
```

Command Line Input Example

```
$ commander aem dump --triggerabove 2 --triggertimeout 10 --pretrigger 250 --datarate 1000
```

This command logs AEM measurements at a rate of 1000 Hz to the console output (indefinitely), starting from when the measured current is greater than 2 mA (milliampere). Data recorded up to 250 ms (milliseconds) before the actual trigger event will also be logged. If no trigger condition is met, the command will time out after 10 seconds.

Command Line Output Example

```
Logging...
Send CTRL+C to abort.
Waiting for trigger (current above 2 mA)...
Triggered at timestamp: 155119217 [us], 3.32649 seconds after sampling started.
154869217,1.00552,3.30057
154870217,1.00447,3.30057
154871217,1.00568,3.30057
<shortened data for documentation>
155117217,1.0006,3.3007
155118217,1.00196,3.30029
155119217,4.96464,3.29997
155120217,4.96765,3.29997
155121217,4.96612,3.30016
^C
Sampling was stopped by user.
DONE
```

6.14 Serial Wire Output Read Commands

Simplicity Commander supports reading and dumping data received over Serial Wire Output (SWO) using the `swo read` command. When the command is executed, the target device is reset. The command will then read and dump SWO data until the application is terminated by pressing Ctrl+C, or one of the conditions described below is met.

By default, the target will be reset during initialization of the SWO connection. Providing the `--noreset` option will prevent this.

6.14.1 Configure SWO Speed

This command sets the SWO speed frequency in Hz. The default SWO speed is 875000 Hz. The SWO speed must match the frequency used by the target application.

Command Line Syntax

```
$ commander swo read [--swospeed <frequency in Hz>]
```

Command Line Input Example

```
$ commander swo read --swospeed 1000000
```

Command Line Output Example

```
<data written by the target application at 1 MHz>
CTRL+C entered, terminating SWO connection...
DONE
```

6.14.2 Read SWO Until Timeout

This command sets the number of seconds for the adapter to wait without receiving data before it times out. The default is to never time out.

Command Line Syntax

```
$ commander swo read [--timeout <timeout in s>]
```

Command Line Input Example

```
$ commander swo read --timeout 1
```

Command Line Output Example

```
<data written by the target application>
Timeout: No SWO output for 1 seconds.
DONE
```

6.14.3 Read SWO Until a Marker Is Found

If the `--endmarker` option is used, the command will terminate after finding the specified string in the SWO stream.

Command Line Syntax

```
$ commander swo read [--endmarker <end marker>]
```

Command Line Input Example

```
$ commander swo read [--endmarker "--finished--"]
```

Command Line Output Example

```
<data written by the target application>
--finished--
SWO connection terminated.
End marker '--finished--' found.
DONE
```

6.14.4 Dump Hex Encoded SWO Output

If the `--hex` option is used, all input and output is converted to a hexadecimal string. This is useful if the target dumps binary data. If the `--hex` option is used, `--endmarker` must also be hex-encoded.

Command Line Syntax

```
$ commander swo read [--hex] [--endmarker <hex encoded end marker>]
```

Command Line Input Example

```
$ commander swo read --hex --endmarker 50415353
```

Command Line Output Example

```
0a5374617274696e6720746573742067726f757020434d550a434d553a333836323a546573745f434d555f4275675f363639393a50415353
SWO connection terminated.
End marker '50415353' found.
DONE
```

6.15 NVM3 Commands

The Third Generation Non-Volatile Memory (NVM3) module in the Gecko SDK provides a way to store data in non-volatile memory (flash) on EFM32 and EFR32 devices. Refer to *UG103.7: Non-Volatile Memory Fundamentals* or *AN1135: Using Third Generation Non-Volatile Memory (NVM3) Data Storage in Dynamic Multiprotocol Applications* for more details on NVM3.

Simplicity Commander supports reading out the NVM3 data area from a device and parsing the NVM3 data to extract stored values. This can be useful in a debugging scenario where you may need to find out the stored state of an application that has been running for some time.

6.15.1 Read NVM3 Data From a Device

This command searches for an NVM3 area in the device's flash and dumps the content to a file in .bin, .s37 or .hex format.

The optional `--range` parameter can be used to specify the memory range where Simplicity Commander should search for NVM3 data. If no range is given, the entire flash is searched.

Command Line Syntax

```
$ commander nvm3 read -o <outfile> [--range <startaddress>:<endaddress>]
```

Command Line Input Example

```
$ commander nvm3 read -o my_nvm3_data.s37
```

Scans through the device flash and searches for a valid NVM3 area. When it is found, the NVM3 area is written to the file named `my_nvm3_data.s37`.

Command Line Output Example

```
Reading 24576 bytes from 0x000fa000...
Writing to my_nvm3_data.s37...
DONE
```

6.15.2 Parse NVM3 Data

This command takes an image file containing NVM3 data and parses the contents. The parsed NVM3 objects are printed to standard out.

The optional `--range` parameter can be used to specify the memory range where Simplicity Commander should search for NVM3 data. If no range is given, the entire file is searched.

The optional `--key` parameter can be used to specify specific NVM3 keys to look up. It can be used multiple times to look up more than one key at a time. Objects with more than eight bytes of data will be truncated when listing all objects. Use the `--key` parameter to select objects whose data should be displayed.

Command Line Syntax

```
$ commander nvm3 parse <file> [--range <startaddress>:<endaddress>] [--key <object key>]
```

Command Line Input Example

```
$ commander nvm3 parse my_nvm3_data.s37
```

Scans through the given file and searches for valid NVM3 data. When it is found, the data is parsed and printed to standard out.

Command Line Output Example

```
Parsing file my_nvm3_data.s37...
Found NVM3 range: 0x000FA000 - 0x00100000
All NVM3 objects:
    KEY -       TYPE -      SIZE - DATA
0x00001 -       Data -       4 B - 2A 00 00 00
0x00002 -       Data -      16 B - 73 36 57 CA 6B CE CF E2 (+ 8 more bytes)
0x00003 -    Counter -       4 B - 2

NVM3 erase count: 1

DONE
```

6.15.3 Initialize NVM3 Area in a File

The `nvm3 initfile` command creates a blank NVM3 area in an image file. For example, this feature is useful to create a file that the `nvm3 set` command can work on to create a default set of NVM3 data that can be written during production.

The size and location of the NVM3 area must be given and must match the size and location used in the embedded application using the NVM3 area.

Command Line Syntax

```
$ commander nvm3 initfile --address <location> --size <size in bytes> --device <target device part number> --
outfile <image file>
```

Command Line Input Example

```
$ commander nvm3 initfile --address 0xfa000 --size 0x6000 --device EFR32MG12P233F1024 --outfile
my_nvm3_data.s37
```

This creates a 24 kB NVM3 area spanning the flash address range 0xfa000 - 0x100000.

Command Line Output Example

```
Placing NVM3 area at address 0x000fa000
Writing to my_nvm3_data.s37...
DONE
```

6.15.4 Write NVM3 Data Using a Text File

The `nvm3 set` command takes an image file containing an NVM3 data region and sets the value of one or more NVM3 objects. The objects may already exist, in which case the value is updated. If the object does not already exist, it is created. The definition of the data to write can be passed either as a text file (`--nvm3file`) or as command line parameters (`--object` and `--counter`).

The text file passed by the `--nvm3file` option must have the following format:

- Each line defines a single object or counter.
- Empty lines are ignored.
- Lines starting with # are ignored.

Each line in the file must have the following syntax:

`<key>:<type>:<data>`

`<key>` is the NVM3 object key which is the unique identifer used by the embedded application. It has a maximum size of 20 bits (maximum value 0xFFFFF).

`<type>` is the NVM3 object type. It can be one of two values: `OBJ` or `CNT`. `OBJ` indicates a plain byte array. `CNT` indicates an NVM3 counter type (32-bit unsigned integer).

`<data>` is the value the object should be set to. For counter types, the value is interpreted as an unsigned integer which can be prefixed with 0x to indicate a hexadecimal value. Byte arrays are always parsed as hexadecimal and should not be prefixed with 0x.

Example File

```
0x00001 : OBJ : 01020304AABBCCDD
0x01000 : CNT : 0x80
0x01001 : CNT : 42
```

This file sets the object with ID 0x1 to be a byte array of eight bytes in length with the contents above.

The object with ID 0x1000 is a counter with value 0x80 (128). The object with ID 0x1001 is a counter with value 42.

Command Line Syntax

```
$ commander nvm3 set <input image file> --nvm3file <filename> --outfile <image file>
```

Command Line Input Example

```
$ commander nvm3 set my_nvm3_data.s37 --nvm3file nvm3_objects.txt --outfile my_modified_nvm3_data.s37
```

`nvm3_objects.txt` is parsed for NVM3 objects following the format described above. The given input image file is scanned for a valid NVM3 region. The objects defined in the text file are written into the NVM3 region and the modified output is written to the output image file.

Command Line Output Example

```
Parsing file my_nvm3_data.s37...
Found NVM3 range: 0x000FA000 - 0x00100000
Setting NVM3 object: 0x00001 = 01020304AABBCCDD
Setting NVM3 counter: 0x01000 = 128 (0x00000080)
Setting NVM3 counter: 0x01001 = 42 (0x0000002a)
Writing to my_modified_nvm3_data.s37...
DONE
```

6.15.5 Write NVM3 Data Using CLI Options

In some cases, it may be more convenient to set the NVM3 object data directly from the command line without using a text file. In this instance, use the command line options `--object` and `--counter`.

The two options both use the same syntax: `<key>:<data>`. The definitions of `<key>` and `<data>` are the same as in 6.15.4 Write NVM3 Data Using a Text File. The only difference between the two formats is that the `<type>` field has been removed because it is given by the command line option name instead.

Simplicity Commander automatically finds the correct NVM3_MAX_OBJECT_SIZE based on the given size of NVM3 area.

Command Line Syntax

```
$ commander nvm3 set <input image file> --object <key>:<data> --counter <key>:<data> --outfile <image file>
```

Command Line Input Example

```
$ commander nvm3 set my_nvm3_data.s37 --object 0x1:01020304AABBCCDD --counter 0x1000:0x80 --counter 0x01001:42
--outfile my_modified_nvm3_data.s37
```

All `--object` and `--counter` parameters are parsed according to the format above. The given input image file is scanned for a valid NVM3 region. The objects defined in the text file are written into the NVM3 region and the modified output is written to the output image file.

Command Line Output Example

```
Parsing file my_nvm3_data.s37...
Setting NVM3 object: 0x00001 = 01020304AABBCCDD
Setting NVM3 counter: 0x01000 = 128 (0x00000080)
Setting NVM3 counter: 0x01001 = 42 (0x0000002a)
Writing to my_modified_nvm3_data.s37...
DONE
```

6.16 CTUNE Commands

Wireless Gecko (EFR32™) portfolio devices support configuring the crystal oscillator load capacitance in software. The crystal oscillator load capacitor tuning (CTUNE) values are tuned during the production test of both Wireless Gecko-based modules and Silicon Labs Wireless Starter Kit (WSTK) radio boards. For modules, the optimal value for each device is written to the Device Information (DI) page in flash. For radio boards, the optimal value for each board is written to an EEPROM that is inaccessible to the software running on the target device, but readable by Simplicity Commander. The `ctune` commands support reading out the stored CTUNE values from these locations, and writing and reading the CTUNE manufacturing token.

6.16.1 CTUNE Get Command

This command retrieves the CTUNE value stored in the Device Info page, the value stored in EEPROM on the board, and the value written to the CTUNE manufacturing token. The values are displayed.

Command Line Syntax

```
$ commander ctune get
```

Command Line Input Example

```
$ commander ctune get
```

Command Line Output Example

```
Getting CTUNE values from the Device Info page, stored in EEPROM on the board, and the MFG token.
DI:    Not set
Board: 346
Token: 346
DONE
```

Note: Not all devices have the CTUNE value stored in both the Device Info page and in EEPROM on the board. If this is the case, the value is displayed as "Not set".

6.16.2 CTUNE Set Command

This command sets the CTUNE manufacturing token to the value specified by the value option.

Command Line Syntax

```
$ commander ctune set <value>
```

Command Line Input Example

```
$ commander ctune set --value 346
```

Command Line Output Example

```
Setting CTUNE token to 346
DONE
```

6.16.3 CTUNE Autoset Command

This command retrieves the CTUNE value from EEPROM on the board and sets the CTUNE manufacturing token to this value.

Command Line Syntax

```
$ commander ctune autoset
```

Command Line Input Example

```
$ commander ctune autoset
```

Command Line Output Example

```
Getting CTUNE value stored on the board...
Board: 346
Setting the CTUNE value...
```

6.17 Security Commands

6.17.1 Get Device Status

This command prints Secure Element device information status, including:

- Firmware version
- Serial number
- Device erase status
- Secure debug unlock status
- Tamper status
- Secure boot status

Command Line Syntax

```
$ commander security status [--trustzone --verbose]
```

Command Line Input Example

```
$ commander security status
```

Command Line Output Example

```
SE Firmware version : 1.1.3
Serial number       : 0000000000000000d0cf5efffe68a68b
Debug lock          : Disabled
Device erase        : Enabled
Secure debug unlock : Disabled
Tamper status       : OK
Secure boot         : Disabled
Boot status         : 0x20 - OK
DONE
```

`Debug lock` indicates whether debug access is Enabled (locked) or Disabled (unlocked).

`Device erase` indicates whether or not it is possible to regain debug access using the `device erase` command. If the device erase is enabled, this is possible. If the device is disabled, it is not possible.

`Security debug unlock` Enabled means that if the device is locked, debug access can be regained using the `security unlock` command. If both `device erase` and `secure debug unlock` are Disabled, it is not possible to regain debug access if the device is locked.

`Tamper status` indicates whether or not a tamper event is detected by the device. OK means no tamper event is detected.

`Secure boot` Enabled means that all images running on the device must be signed with the private sign key corresponding to the public sign key written to the device.`Secure boot` Disabled means that images do not have to be signed with the sign key.

`Boot status` shows if secure boot failed or if the secure boot is OK.

Command Line Input Example

```
$ commander security status --trustzone
```

Show the TrustZone status of the device.

Command Line Output Example

```
SE Firmware version : 1.1.3
Serial number       : 0000000000000000d0cf5efffe68a68b
Debug lock          : Disabled
Device erase        : Enabled
Secure debug unlock : Disabled

Debug lock state: Unlocked

Non-secure, invasive debug lock     (DBGLOCK)   : Unlocked
Non-secure, non-invasive debug lock (NIDLOCK)   : Unlocked
Secure, invasive debug lock         (SPIDLOCK)  : Unlocked
Secure, non-invasive debug lock     (SPNIDLOCK): Unlocked
```

```
Non-secure, invasive debug lock state      (DBGLOCK)  : Unlocked
Non-secure, non-invasive debug lock state (NIDLOCK)   : Unlocked
Secure, invasive debug lock state          (SPIDLOCK) : Unlocked
Secure, non-invasive debug lock state      (SPNIDLOCK): Unlocked

Tamper status        : OK
Secure boot          : Disabled
Boot status          : 0x20 - OK
DONE
```

`Debug lock state` indicates whether the debug port is locked or unlocked.

The `TrustZone` debug lock configuration consists of the four modes `SPNIDLOCK`, `SPIDLOCK`, `NIDLOCK` and `DBGLOCK`. The top configurations specifies which mode has been locked by the `security lock --trustzone` command. The bottom configuration specifies the actual state of the mode, whether or not it has been unlocked.

Command Line Input Example

```
$ commander security status --verbose
```

Show verbose output of the security status.

Command Line Output Example

```
SE Firmware version : 1.1.3
Serial number       : 0000000000000000d0cf5efffe68a68b
Debug lock          : Disabled
Device erase        : Enabled
Secure debug unlock : Disabled
Tamper status       : OK
Secure boot         : Disabled
Boot status         : 0x20 - OK
Verbose output: 00000000 00000000 00000000 FFFFFFFF 00000020 03020200 00000000 00000002 FFFFFFFF
DONE
```

`Verbose output` is the entire output from the Secure Element of the device.

6.17.2 Generate Key Pair

This command has been deprecated. For more information on how to generate keys, see 6.18.2 Generating a Signing Key and 6.18.1 Key Generation.

6.17.3 Write Public Key to Device

IMPORTANT: This is a one-time command. It cannot be run more than once per device.

This one-time command permanently locks the device to this key pair. There are two different public keys that can be written to the device.

- **Command key** - the corresponding private key is used to create certificates to perform secure debug unlock.
- **Sign key** - the corresponding private key must sign all code that is to run on the device when Secure Boot is enabled.

When Secure Debug Unlock is enabled, a locked device may temporarily unlock debug access by creating a certificate signed by the private command key.

When Secure Boot is enabled, all code that runs on the device must be signed by the private sign key.

Command Line Syntax

```
$ commander security writekey [--command <public key PEM file>] [--sign <public key PEM file>]
```

Command Line Input Example

```
$ commander security writekey --command command_public_key.pem
```

Command Line Output Example

```
Device has serial number 000000000000000014b457fffed50c35

================================================================================
Please look through any warnings before proceeding.
THIS IS A ONE-TIME command, all code to be run on the device must be signed by this key.
Type 'continue' and hit enter to proceed or Ctrl-C to abort:
================================================================================
continue
DONE
```

6.17.4 Read Public Key from Device

This command reads out a public key from the device. There are two different public keys that can be stored on the device using the `commander security writekey` command.

- **Command key** – the corresponding private key is used to create certificates to perform secure debug unlock or disable tamper.
- **Sign key** – the corresponding private key must sign all code that is to run on the device when Secure Boot is enabled.

By providing an output file, the key will be written to the file. Otherwise, the key will be printed to the Command Line Interface (CLI) as a byte array.

If the optional `--nostore` option is not used, the key will also be stored in the Security Store.

Command Line Syntax

```
$ commander security readkey [--command] [--sign] [--outfile <filename>] [--nostore]
```

Command Line Input Example

```
$ commander readkey --command --outfile command_public_key.pem
```

Command Line Output Example

```
Writing public key file in PEM format to key.pem...
DONE
```

6.17.5 Configure Lock Options

The `security lockconfig` command enables or disables secure debug unlock. When secure debug unlock is enabled, a locked device may be temporarily unlocked by running a `commander security unlock` command. If secure debug unlock is disabled, the only way to unlock a locked device is to run a `commander security erasedevice` command, given that device erase has not been disabled. If both device erase and secure debug unlock are disabled, there is no way to unlock debug access to a locked device.

Note: Secure debug unlock must be enabled before the device is locked.

Command Line Syntax

```
$ commander security lockconfig --secure-debug-unlock <enable/disable>
```

Command Line Input Example

```
$ commander security lockconfig --secure-debug-unlock enable
```

Command Line Output Example

```
Secure debug unlock was enabled.
DONE
```

6.17.6 Lock Debug Access

The `lock` command locks the debug interface on the device. If secure debug unlock has been enabled, the device may be unlocked using the `unlock` command. If device erase has not been disabled, the debug access may also be unlocked using the `commander security erasedevice` command. However, this also triggers a mass erase on the device.

The `--trustzone` option may be used to lock debug access to specific TrustZone modes. The bitmask to set TrustZone debug lock is defined as `<SPNIDLOCK, SPIDLOCK, NIDLOCK, DBGLOCK>`. If the bit is set to 1, debug access to the corresponding TrustZone mode will be locked. Set the bit to 0 to keep it open. By default all modes are open.

Command Line Syntax

```
$ commander security lock [--trustzone <xxxx>]
```

Command Line Input Example

```
$ commander security lock
```

Command Line Output Example

```
Device is now locked.
DONE
```

Command Line Input Example

```
$ commander security lock --trustzone 0011
```

Debug access to TrustZone modes `DBGLOCK` and `NIDLOCK` are locked.

Command Line Output Example

```
Writing debug restriction bits:
DBGLOCK:    1
NIDLOCK:    1
SPIDLOCK:   0
SPNIDLOCK:  0
Device is now locked.
DONE
```

6.17.7 Secure Debug Unlock

The security unlock command opens debug access on a locked device temporarily without erasing the flash content. When running the `commander security unlock` command, Simplicity Commander will use all available files in the Security Store and from command line options in an attempt to unlock debug access. If anything is missing, you will be asked to provide the file as an option to the command. All files generated or given as command line options are stored in the Security Store, unless the `--nostore` option is used.

For more information about Secure Debug, see *AN1190: EFR32xG21 Secure Debug*.

There are several different ways to unlock the debug access, as illustrated in the following figure. The blue fields are actions and the red fields are artifacts.

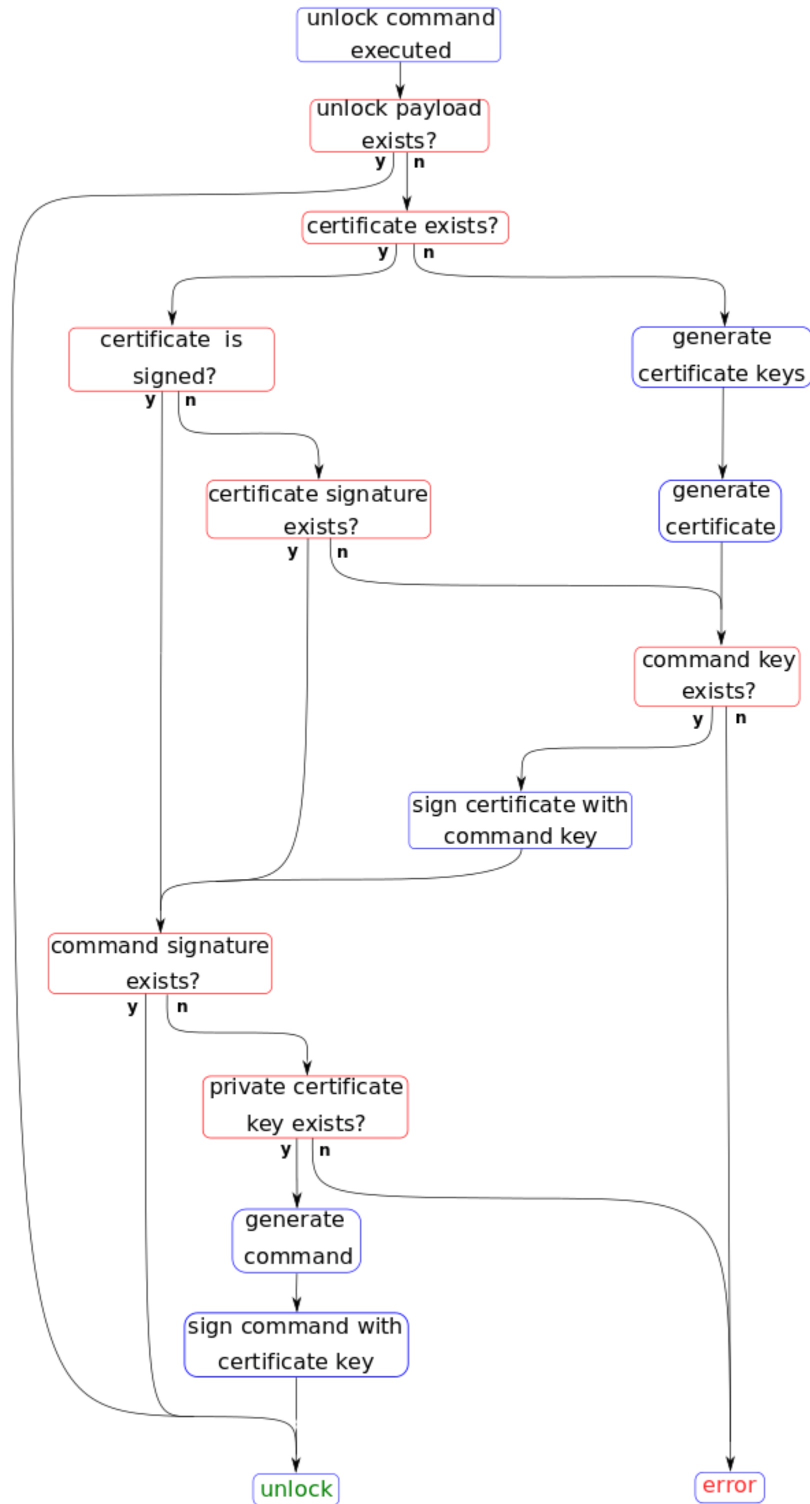


Figure 6.1. Unlock Flow

Command Line Syntax

```
$ commander security unlock [--cert <signed access certificate> --cert-signature <signature> --command-signature <signature> --cert-privkey <keyfile> --cert-pubkey <keyfile> --command-key <keyfile> --nostore]
```

Command Line Input Example

```
$ commander security unlock --command-key command_key.pem
```

This example uses and generates a certificate and command signature on-the-fly using the provided command key to sign the certificate. All the generated files and the command key are stored in the Security Store.

Command Line Output Example

```
Command public key stored in:
/Users/example/Library/Preferences/SiliconLabs/commander/SecurityStore/device_0000000000000000d0cf5efffe68a68b/
command_pubkey.pem
Command private key stored in:
/Users/example/Library/Preferences/SiliconLabs/commander/SecurityStore/device_0000000000000000d0cf5efffe68a68b/
command_key.pem
Authorization file written to Security Store:
/Users/example/Library/Preferences/SiliconLabs/commander/SecurityStore/device_0000000000000000d0cf5efffe68a68b/
certificate_authorizations.json
Generating ECC P256 key pair...
Cert public key stored at:
/Users/example/Library/Preferences/SiliconLabs/commander/SecurityStore/device_0000000000000000d0cf5efffe68a68b/
cert_pubkey.pem
Cert private key stored at:
/Users/example/Library/Preferences/SiliconLabs/commander/SecurityStore/device_0000000000000000d0cf5efffe68a68b/
cert_key.pem
Command key matches public command key found on device. Signing certificate...
Certificate was signed with key:
test-cases/common/security_testfiles/command_key.pem
Successfully stored certificate
Certificate written to Security Store:
/Users/example/Library/Preferences/SiliconLabs/commander/SecurityStore/device_0000000000000000d0cf5efffe68a68b/
access_certificate.bin
Created unsigned unlock command
Signed unlock command using
/Users/example/Library/Preferences/SiliconLabs/commander/SecurityStore/device_0000000000000000d0cf5efffe68a68b/
cert_key.pem
Secure debug successfully unlocked
Command unlock payload was stored in Security Store
DONE
```

Command Line Input Example

```
$ commander security unlock --cert access_certificate.bin --cert-privkey cert_key.pem
```

This example unlocks the device with a signed access certificate and the private certificate key corresponding to the public key in the access certificate. The certificate and key are stored in the Security Store.

Command Line Output Example

```
/Users/example/Library/Preferences/SiliconLabs/commander/SecurityStore/device_0000000000000000d0cf5efffe68a68b/
access_certificate.bin
Cert key written to Security Store:
/Users/example/Library/Preferences/SiliconLabs/commander/SecurityStore/device_0000000000000000d0cf5efffe68a68b/
cert_pubkey.pem
Created unsigned unlock command
Signed unlock command using
/Users/example/Library/Preferences/SiliconLabs/commander/SecurityStore/device_0000000000000000d0cf5efffe68a68b/
cert_key.pem
Secure debug successfully unlocked
Command unlock payload was stored in Security Store
DONE
```

Command Line Input Example

```
$ commander security unlock --cert-signature cert_signature.bin --command-signature command_signature.bin
```

This example uses externally generated signatures for both the access certificate and command file. The access certificate signature is appended to the certificate and stored in the Security Store. The command signature is validated against the public key in the certificate.

Command Line Output Example

```
Using certificate from Security Store:
/Users/example/Library/Preferences/SiliconLabs/commander/SecurityStore/device_0000000000000000d0cf5efffe68a68b/
```

```
access_certificate.bin
Certificate in Security Store is not signed.
Moved existing file to:
/Users/example/Library/Preferences/SiliconLabs/commander/SecurityStore/device_0000000000000000d0cf5efffe68a68b/
archive/access_certificate.bin
Signed certificate written to Security Store:
/Users/example/Library/Preferences/SiliconLabs/commander/SecurityStore/device_0000000000000000d0cf5efffe68a68b/
access_certificate.bin
Command signature is valid
Secure debug successfully unlocked
Command unlock payload was stored in Security Store
```

Command Line Input Example

```
$ commander security unlock
```

When the device has been unlocked with the current challenge, the unlock payload is stored in the Security Store. The next time the unlock command is run, the device is unlocked directly with the unlock payload.

Command Line Output Example

```
Unlocking with unlock payload:
/Users/example/Library/Preferences/SiliconLabs/commander/SecurityStore/device_0000000000000000d0cf5efffe68a68b/
challenge_4329288395adfc4eea436e5d64dd296b/unlock_payload_0000000000111110.bin
DONE
```

6.17.8 Disable Tamper

Secure Vault products are capable of detecting certain types of tamper events and responding to mitigate the attack. This provides an extra layer of protection against attacks that rely on physically tampering with the product.

Before this command can be executed, the tamper sources must be configured in the One-Time-Programmable (OTP) settings of the devices. See 6.17.16 Write User Configuration for more information about how this is done.

The process of disabling tamper follows the same flow as the `security unlock` command. For more information about the flow, see 6.17.7 Secure Debug Unlock.

A certificate and a signed challenge are required to disable tamper. The certificate—including tamper authorizations—is generated and signed with a command key. The certificate contains a public key and the corresponding private key must be used to sign a challenge from the device to disable tamper sources. The `--disable-param` option determines which tamper sources to disable. If this option is not provided, Simplicity Commander will extract the tamper authorizations from the certificate and disable everything allowed by the certificate. If the certificate is not available, all sources will be disabled.

The tamper sources are disabled until the next Power On Reset.

Command Line Syntax

```
$ commander security disabletamper [--disable-param <disable-mask> --cert <signed access certificate> --cert-signature <signature> --commandsignature <signature> --cert-privkey <keyfile> --cert-pubkey <keyfile> --command-key <keyfile> --nostore]
```

Command Line Input Example

```
$ commander security disabletamper --cert access_certificate.bin --cert-privkey cert_key.pem
```

Command Line Output Example

```
Using tamper parameters from certificate in Security Store: 0xffffffb6
Certificate written to Security Store:
/Users/matundal/Library/Preferences/SiliconLabs/commander/SecurityStore/
device_0000000000000000d6ffffead3617/access_certificate.bin
Cert key written to Security Store:
/Users/matundal/Library/Preferences/SiliconLabs/commander/SecurityStore/
device_0000000000000000d6ffffead3617/cert_pubkey.pem
Using tamper parameters from certificate in Security Store: 0xffffffb6
Created unsigned disable tamper command
Signed disable tamper command using
/Users/matundal/Library/Preferences/SiliconLabs/commander/SecurityStore/
device_0000000000000000d6ffffead3617/cert_key.pem
Tamper successfully disabled.
Command disable tamper payload was stored in Security Store

DONE
```

6.17.9 Device Erase using Secure Element

This command performs a device mass erase and resets the debug configuration to its initial unlocked state.

The complete flash and RAM of the system is cleared, excluding the user data page and one-time programmable commissioning information in the Secure Element.

If device erase has been disabled, this command is not available.

Note: After a device erase, the DCI interface is unavailable until the device has been reset

Command Line Syntax

```
$ commander security erasedevice
```

Command Line Input Example

```
$ commander security erasedevice
```

Command Line Output Example

```
Successfully erased device
DONE
```

6.17.10 Disable Device Erase

IMPORTANT: This is a one-time command. It cannot be run more than once.

This command permanently disables device erase. When device erase is disabled, the `commander security erasedevice` command is no longer available. This means that if debug access is locked, debug access can only be opened if secure debug unlock has been enabled before the device was locked. If not, there is no way to regain debug access. This command can be run after the device has been locked.

Confirmation is required from the user to execute this command, except if the `--noprompt` option is used.

Command Line Syntax

```
$ commander security disabledeviceerase [--noprompt]
```

Command Line Input Example

```
$ commander security disabledeviceerase
```

Command Line Output Example

```
================================================================================
THIS IS A ONE-TIME command which Permanently disables device erase.
If secure debug lock has not been set, there is no way to regain debug access to this device.
Type 'continue' and hit enter to proceed or Ctrl-C to abort:
================================================================================
continue
Disabled device erase successfully
DONE
```

6.17.11 Roll Challenge

This command makes the Secure Element *roll* or update its challenge data. The challenge is random data that must be read from the device before an unlock command can be executed. Rolling the challenge renders existing command signatures invalid. For more information, see 5.3 Challenge and Command Signing.

The challenge cannot be rolled before it has been used at least once—that is, by running the security unlock command or the disable tamper command.

```
$ commander security rollchallenge
```

Command Line Input Example

```
$ commander security rollchallenge
```

Command Line Output Example

```
Challenge was rolled successfully.
DONE
```

6.17.12 Generate Example Authorization File

This command generates a default authorization file to be used in the certificate. The authorization file will be stored in Security Store unless the `--nostore` option is used.

Default Authorization File for Devices without Secure Vault

```
{
    "debug_authorizations":{
        "ENABLE_DEBUG_PORT": true
    }
}
```

Default Authorization File for Devices with Secure Vault

```
{
    "debug_authorizations":{
        "ENABLE_DEBUG_PORT": true
    },
    "tamper_authorizations":{
        "FILTER_COUNTER": 1,
        "WATCHDOG": 1,
        "SE_RAM_CRC": 1,
        "SE_HARDFAULT": 1,
        "SOFTWARE_ASSERTION": 1,
        "SE_CODE_AUTH": 1,
        "USER_CODE_AUTH": 1,
        "MAILBOX_AUTH": 1,
        "DCI_AUTH": 1,
        "OTP_READ": 1,
        "AUTO_CODE_AUTH": 1,
        "SELF_TEST": 1,
        "TRNG_MONITOR": 1,
        "PRS0": 1,
        "PRS1": 1,
        "PRS2": 1,
        "PRS3": 1,
        "PRS4": 1,
        "PRS5": 1,
        "PRS6": 1,
        "PRS7": 1,
        "DECOUPLE_BOD": 1,
        "TEMP_SENSOR": 1,
        "VGLITCH_FALLING": 1,
        "VGLITCH_RISING": 1,
        "SECURE_LOCK": 1,
        "SE_DEBUG": 1,
        "DGLITCH": 1,
        "SE_ICACHE": 1
    }
}
```

Debug Authorization

Enable Debug Port must be set to *true* in order to perform a secure debug unlock. For more information about secure debug unlock, see 6.17.7 Secure Debug Unlock.

Tamper Authorizations

The Tamper Authorizations indicate which sources may be disabled. By default all sources may be disabled. For more information about disabling tamper sources, see 6.17.8 Disable Tamper.

Command Line Syntax

```
$ commander security genauth [-o <filename>] [--nostore]
```

Command Line Input Example

```
$ commander security genauth -o certificate_authorization.json --nostore
```

Command Line Output Example

```
Authorization file stored in:
certificate_authorization.json
DONE
```

6.17.13 Generate Access Certificate

Access certificates are used to unlock debug access or disable tamper on the device. For more information, see 6.17.7 Secure Debug Unlock or Disable Tamper. The certificate and the keys provided to or generated by Simplicity Commander are stored in Security Store unless the `--nostore` option is used. If `--cert-pubkey` or `--authorization` are not used as options on the command line, Simplicity Commander checks if the files are stored in Security Store. If the files are not in Security Store, Simplicity Commander generates a default authorization file that may be edited. If the file is edited, a new certificate must be generated. Simplicity Commander will also generate a pair of certificate keys if the `--cert-pubkey` option is not used. If the certificate keys are generated, the `--nostore` option cannot be used. If the `--command-key` option is not used on the command line and not located in Security Store, the `--extsign` option should be used for Simplicity Commander to generate an unsigned certificate. To use the certificate to unlock debug access, a certificate signature must be generated and provided. If the device for which the certificate is made is connected, Simplicity Commander retrieves the device serial number directly.

Note: Before Simplicity Commander version 1.11.2 unsigned certificates were created with all zeros in replace of the signature. This was fixed in version 1.11.2 making it compatible with external signing using tools such as OpenSSL.

Figure 6.2. Access Certificate

Command Line Syntax

```
$ commander security gencert [--cert-pubkey <public key file>] [--authorization <auth-file>] [--command-key
<private key file>][--extsign][--devserialno <serial number>] [-o <filename>] [--nostore]
```

Command Line Input Example

```
$ commander security gencert --extsign
```

This example generates an unsigned certificate, as the command private key is not provided as a command option, nor is it located in Security Store. The public certificate key is not provided either, so Simplicity commander generates a pair of certificate keys and stores them in Security Store. A default authorization file is also generated and stored in Security Store.

Command Line Output Example

```
Authorization file written to Security Store:
/Users/example/Library/Preferences/SiliconLabs/commander/SecurityStore/device_0000000000000000d0cf5efffe68a68b/
certificate_authorizations.json
Generating ECC P256 key pair...
Cert public key stored at:
/Users/example/Library/Preferences/SiliconLabs/commander/SecurityStore/device_0000000000000000d0cf5efffe68a68b/
cert_pubkey.pem
Cert private key stored at:
/Users/example/Library/Preferences/SiliconLabs/commander/SecurityStore/device_0000000000000000d0cf5efffe68a68b/
cert_key.pem
Successfully stored certificate
Created an unsigned certificate in Security Store:
/Users/example/Library/Preferences/SiliconLabs/commander/SecurityStore/device_0000000000000000d0cf5efffe68a68b/
access_certificate.extsign
DONE
```

Command Line Input Example

```
$ commander security gencert --cert-pubkey cert_pubkey.pem --authorization certificate_authorizations.json --
command-key command_key.pem -o access_certificate.bin --nostore
```

In this example, all files needed to generate the certificate are provided as command line options. The device serial number is taken directly from the connected device. The certificate is signed with the private command key, and is ready to be used to unlock debug access.

Command Line Output Example

```
Command key matches public command key found on device. Signing certificate...
Certificate was signed with key:
command_key.pem
DONE
```

Command Line Input Example

```
$ commander security gencert
```

This example uses files already located in Security Store to generate a signed certificate. The certificate is stored in Security Store.

Command Line Output Example

```
Using authorizations from Security Store:
/Users/example/Library/Preferences/SiliconLabs/commander/SecurityStore/device_0000000000000000d0cf5efffe68a68b/
certificate_authorizations.json
Using public key from Security Store:
/Users/example/Library/Preferences/SiliconLabs/commander/SecurityStore/device_0000000000000000d0cf5efffe68a68b/
cert_pubkey.pem
Found command key in Security Store:
/Users/example/Library/Preferences/SiliconLabs/commander/SecurityStore/device_0000000000000000d0cf5efffe68a68b/
command_key.pem
Certificate was signed with key:
/Users/example/Library/Preferences/SiliconLabs/commander/SecurityStore/device_0000000000000000d0cf5efffe68a68b/
command_key.pem
DONE
```

6.17.14 Generate Unsigned Command File

The `commander security gencommand` command retrieves the security challenge from the device and stores it in a file with other data as described in Figure 5.2 Unlock Command Signature on page 22. The signature of this file using the private certificate key can be used as part of the payload to perform a secure debug unlock.

Unless the `--nostore` option is used, the unsigned command file will be stored in the Security Store.

If the user has the private certificate key, Simplicity Commander automatically generates the command file and signature using the `commander security unlock` command. If the command file is signed by an external process—for example, an HSM—the command signature needs to be passed as a command line option when executing the `commander security unlock` command.

Command Line Syntax

```
$ commander security gencommand --action debug_unlock [-o <output file>] [--nostore]
```

Command Line Input Example

```
$ commander security gencommand --action debug-unlock -o unlock_command_to_be_signed.bin --nostore
```

Command Line Output Example

```
Unsigned command file written to:
unlock_command_to_be_signed.bin
DONE
```

6.17.15 Generate Example Configuration File

This command generates a default configuration file to be used with the `security_writeconfig` command. The file is stored in Security Store unless the `--nostore` option is used.

Default Configuration File for Devices without Secure Vault

```
{
   "mcu_flags": {
       "SECURE_BOOT_ENABLE": true,
       "SECURE_BOOT_VERIFY_CERTIFICATE": false,
       "SECURE_BOOT_ANTI_ROLLBACK": true,
       "SECURE_BOOT_PAGE_LOCK_NARROW": false,
       "SECURE_BOOT_PAGE_LOCK_FULL": true
   }
}
```

Default Configuration File for Devices with Secure Vault

```
{
   "mcu_flags": {
       "SECURE_BOOT_ENABLE": true,
       "SECURE_BOOT_VERIFY_CERTIFICATE": false,
       "SECURE_BOOT_ANTI_ROLLBACK": true,
       "SECURE_BOOT_PAGE_LOCK_NARROW": false,
       "SECURE_BOOT_PAGE_LOCK_FULL": true
   },
   "tamper_levels": {
        "FILTER_COUNTER": 0,
        "WATCHDOG": 4,
        "SE_RAM_CRC": 4,
        "SE_HARDFAULT": 4,
        "SOFTWARE_ASSERTION": 4,
        "SE_CODE_AUTH": 4,
        "USER_CODE_AUTH": 4,
        "MAILBOX_AUTH": 0,
        "DCI_AUTH": 0,
        "OTP_READ": 0,
        "AUTO_CODE_AUTH": 0,
        "SELF_TEST": 4,
        "TRNG_MONITOR": 0,
        "PRS0": 0,
        "PRS1": 0,
        "PRS2": 0,
        "PRS3": 0,
        "PRS4": 0,
        "PRS5": 0,
        "PRS6": 0,
        "PRS7": 0,
        "DECOUPLE_BOD": 4,
        "TEMP_SENSOR": 1,
        "VGLITCH_FALLING": 0,
        "VGLITCH_RISING": 0,
        "SECURE_LOCK": 4,
        "SE_DEBUG": 0,
        "DGLITCH": 0,
        "SE_ICACHE": 4
   },
   "tamper_filter" : {
        "FILTER_PERIOD": 0,
        "FILTER_THRESHOLD": 0,
        "RESET_THRESHOLD": 0
   },
   "tamper_flags": {
        "DGLITCH_ALWAYS_ON": false
   }
}
```

MCU settings

- **Secure Boot Enable** – Enables Secure Boot on the device if set. Requires all applications running on the device to be signed.

- **Secure Boot Verify Certificate** – Applications running on the device must be signed using an intermediary certificate if this option is set. It is still possible to use certificates for signing even if this option is not set. For more information, see 6.5.9 Signing an Application for Secure Boot using an Intermediary Certificate.
- **Secure Boot Anti Rollback** – If set, application images with a lower version than the image currently stored in flash will not run on the device.
- **Secure Boot Page Lock Narrow** – Flash pages validated by the Secure Boot process are locked down to prevent re-flashing by means other than through Root Code. Pages from 0 through the page where the Secure Boot signature of the application is located are locked down, **not including** the last page if the signature is not on a page boundary.
- **Secure Boot Page Lock Full** – Flash pages validated by the Secure Boot process are locked down to prevent re-flashing by means other than through Root Code. Pages from 0 through the page where the Secure Boot signature of the application is located are locked down, **including** the last page if the signature is not on a page boundary.

Tamper Levels

The different tamper sources are listed under tamper levels. The default configuration is an absolute minimum. The Root Code will never set tamper levels to a lower setting than the default configuration. The tamper levels are listed in the following table.

Table 6.1. Tamper Levels

Tamper Level	Description
1	No action taken
2	Generate SE interrupt
3	Increment filter counter
4	System Reset
5	Reserved
6	Reserved
7	Erase OTP (Makes the device unrecoverable; it will neve boot again.)

Command Line Syntax

```
$ commander security genconfig [-o <filename>] [--nostore]
```

Command Line Input Example

```
$ commander security genconfig -o user_configuration.json --nostore
```

Command Line Output Example

```
Configuration file stored in:
user_configuration.json
DONE
```

6.17.16 Write User Configuration

IMPORTANT: This is a one-time command. It cannot be run more than once.

The `commander security writeconfig` command sets the configurations determined in the configuration file in the Root Code.

Secure Boot is enabled through this command. Before Secure Boot is enabled, you must write the public sign key to the device. For more information on writing keys to the device, see 6.17.3 Write Public Key to Device. In addition, a configuration file must be generated and the Secure Boot Enabled flag must be set to true. If no configuration file is provided, a default configuration will be generated.

In Simplicity Commander version 1.9, tamper configuration is supported on devices with Secure Vault. The tamper configuration determines the response from the Secure Element in the occurrence of a tamper event. For more information about the configuration file and tamper configuration, see 6.17.15 Generate Example Configuration File.

For more information about Secure Boot, see *AN1218: Series 2 Secure Boot with RTSL.*

For more information about tamper events, see 6.17.8 Disable Tamper.

Command Line Syntax

```
$ commander security writeconfig [--configfile <config file>] [--nostore] [--nopromt]
```

Command Line Input Example

```
$ commander security writeconfig --configfile user_configuration.json
```

Command Line Output Example

```
================================================================================

THIS IS A ONE-TIME configuration: Please inspect file before confirming:
/Users/example/Library/Preferences/SiliconLabs/commander/SecurityStore/device_0000000000000000d0cf5efffe68a68b/
user_configuration.json
Type 'continue' and hit enter to proceed or Ctrl-C to abort:
================================================================================
continue
DONE
```

6.17.17 Read User Configuration

This command returns the One-Time Programmable (OTP) setting from the device. If the device has not been configured with the 6.17.16 Write User Configuration command, no OTP settings are available to read.

Command Line Syntax

```
$ commander security readconfig
```

Command Line Input Example

```
$ commander security readconfig
```

Command Line Output Example

```
MCU Flags
Secure Boot                     : Enabled
Secure Boot Verify Certificate : Disabled
Secure Boot Anti Rollback       : Enabled
Secure Boot Page Lock Narrow    : Disabled
Secure Boot Page Lock Full      : Enabled

Tamper Levels
FILTER_COUNTER      : 0
WATCHDOG            : 4
SE_RAM_CRC          : 4
SE_HARDFAULT        : 4
SOFTWARE_ASSERTION : 4
SE_CODE_AUTH        : 4
USER_CODE_AUTH      : 4
MAILBOX_AUTH        : 0
DCI_AUTH            : 0
OTP_READ            : 0
AUTO_CODE_AUTH      : 0
SELF_TEST           : 4
TRNG_MONITOR        : 0
PRS0                : 0
PRS1                : 0
PRS2                : 0
PRS3                : 0
PRS4                : 0
PRS5                : 0
PRS6                : 0
PRS7                : 0
DECOUPLE_BOD        : 4
TEMP_SENSOR         : 1
VGLITCH_FALLING     : 0
VGLITCH_RISING      : 0
SECURE_LOCK         : 4
SE_DEBUG            : 0
DGLITCH             : 0
SE_ICACHE           : 4

Tamper Filter
Filter Period   : 0
Filter Treshold : 0
Reset Treshold  : 0

Tamper Flags
Digital Glitch Detector Always On: Disabled

DONE
```

6.17.18 Get Security Store Path

Get the path to the security store. If a device is connected or the --deviceserialno option is provided, the device specific path is returned. Otherwise, the path to Security Store is returned.

Command Line Syntax

```
$ commander security getpath [--deviceserialno <deviceserialno>]
```

Command Line Input Example

```
$ commander security getpath
```

Command Line Output Example

```
/Users/example/Library/Preferences/SiliconLabs/commander/SecurityStore/device_0000000000000000d0cf5efffe68a68b
DONE
```

6.17.19 Write AES Decryption Key

Important: This is a one-time command. It cannot be run more than once per device.

The symmetric 128-bit AES key is used to decrypt GBL files. This key is also known as the MFG_BOOTLOAD_AES_KEY. All encrypted images on this device must be encrypted with the same AES key.

Command Line Syntax

```
$ commander security writekey --decrypt <filename>
```

Command Line Input Example

```
$ commander security writekey --decrypt key.txt
```

Command Line Output Example

```
Device has serial number 000000000000000014b457fffed50c35
================================================================================
Please look through any warnings before proceeding.
THIS IS A ONE-TIME command, all code to be run on the device must be signed by this key.
Type 'continue' and hit enter to proceed or Ctrl-C to abort:
================================================================================
continue
DONE
```

6.17.20 Read Device Certificates

This command reads out a X509 certificate from the device. The available certificates are:

- batch - same for each manufacturing batch
- SE - unique per device
- MCU - unique per device

The certificates form a root-of-trust certificate chain up to the `Silicon Labs Root Certificate` issued by Silicon Labs. The `SE` and `MCU Certificates` are issued by a `Batch Certificate`. The `Batch Certificate` is issued by a `Factory Certificate`, and the `Factory Certificate` is issued by the `Silicon Labs Root Certificate`.

Key information about the certificate is printed to the command line if no outfile is given. The certificate may be read out in entirety by providing the `outfile` argument. The available encodings are `pem` and `der`.

Command Line Syntax

```
$ commander security readcert <cert type> [--outfile <filename>]
```

Command Line Input Example

```
$ commander security readcert batch
```

Command Line Output Example

```
Version             : 3
Subject             : CN=Batch 1001317  O=Silicon Labs Inc.  C=US
Issuer              : CN=Factory  O=Silicon Labs Inc.  C=US
Valid From          : October 17 2019
Valid To            : September 16 2118
Signature algorithm: SHA256
Public Key Type     : ECDSA
Public         key                                                                                 :
b0c113190bba3d1ee507d954e878957ad5cc8903ec7785525b8c0b2c2185514cd1421498487c5ea554801924468f8534e027e6496fcbdecef3659cc
DONE
```

Command Line Input Example

```
$ commander security readcert se --outfile se_cert.pem
```

Command Line Output Example

```
Writing certificate to se_cert.pem...
DONE
```

6.17.21 Vault Device Attestation

Attestation of a device is used to cryptographically prove to a remote party that they are the system they say they are, and ensure that the device they are talking to is the same device as the one that got produced in the factory.

The attestation process starts with authenticating the certificate chain up to the Silicon Labs Root certificate. For more information on certificates, see 6.17.20 Read Device Certificates.

The attestation token is printed to the command line. The token consists of multiple claims as listed in the following table.

Claim ID	Claim friendly name	Present in token	Content
-75000	ARM PSA Profile ID	Always	ASCII 'SILABS_1'
-75008	ARM PSA nonce	Always	Copy of the nonce supplied as input to the token generation command.
-75009	ARM PSA/IETF EAT UEID	Always	The device's EUI-64 pre-pended with 0x06 and zeroes.
-76000	SE status	Always	Current SE status
-76001	OTP configuration	Always when provisioned	User configuration
-76002	MCU Sign key	Always when provisioned	Public sign key
-76003	MCU Command key	Always when provisioned	Public command key
-76004	Current applied tamper settings	Always	Currently applied tamper level per tamper signal (one nibble per tamper signal).

Finally, the signature of the attestation token is verified as shown in the following examples.

```
$ commander security attestation
```

Command Line Input Example

```
$ commander security attestation
```

Command Line Output Example

```
Certificate chain successfully validated up to Silicon Labs device root certificate.

-75008 ARM PSA nonce                   : 1799c9296ac44a854b74fe50dc6f1546a5c1e17de73584afcc478739161db7d0
-75000 ARM PSA Profile ID              : SILABS_1
-75009 ARM PSA/IETF EAT UEID           : 0614b457fffe0f7789
-76000    SE    status                                                                            :
000000010000000000000000000000000000002000010202ffffffff00000002ffffffff
-76002    MCU    sign    key                                                                      :
fb2470314c0710f5a72e89a30d2af607770187568f80cffa7fc6516f61e0dc258a8606fe664a097eb94d3ea29e1b87262babdb969842da31512bdc7
-76003     MCU     command     key                                                                :
a218c9615321567527e94ac1f01230604e231f1eabe699fb1d751af3e28d00feaa3dd823540a2452baa40dfb3475d3bb786b41e7880881b5a5427e7
-76004 Current applied tamper settings : 05044440040004040000000014000440

Successfully validated signature of attestation token.
```

6.18 Util Commands

6.18.1 Key Generation

Generates a keyfile to be used for encryption and decryption and outputs the keyfile to the specified filename.

Command Line Syntax

```
$ commander util genkey --type aes-ccm --outfile <filename>
```

Command Line Input Example

```
$ commander util genkey --type aes-ccm --outfile key.txt
```

Command Line Output Example

```
Using /dev/random for random number generation
Gathering sufficient entropy... (may take up to a minute)...
DONE
```

6.18.2 Generating a Signing Key

Creates an EDCSA-P256 key pair and outputs the result to the specified private and public key files. For more information, see *UG266: Silicon Labs Gecko Bootloader User's Guide for GSDK 3.x and Lower* or *UG489: Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher*.

Command Line Syntax

```
$ commander util genkey --type ecc-p256 --privkey <filename> --pubkey <filename> [--tokenfile <filename>]
```

Command Line Input Example

```
$ commander util genkey --type ecc-p256 --privkey signing_key.pem --pubkey signing_pubkey.pem
```

Command Line Output Example

```
Generating ECC P256 key pair...
Writing private key file in PEM format to signing_key.pem
Writing public key file in PEM format to signing_pubkey.pem
DONE
```

6.18.3 Key to Token

Creates a token text file containing an Elliptic Curve Cryptography (ECC) public key suitable for flashing to a device. For more information, see *UG266: Silicon Labs Gecko Bootloader User's Guide for GSDK 3.x and Lower or UG489: Silicon Labs Gecko Bootloader User's Guide for GSDK 4.0 and Higher.*

Command Line Syntax

```
$ commander util keytotoken <input file> --outfile <filename>
```

Command Line Input Example

```
$ commander util keytotoken my_pubkey.pem --outfile keytokens.txt
```

Command Line Output Example

```
Writing EC tokens to keytokens.txt...
DONE
```

6.18.4 Key Config Generation

Generates a key configuration file to the specified file name. This command is only available for Si917 devices, so the device options is required. The output file is used as input to the 6.25.8 Provision Security Keys to the Device command. The file contains the following keys:

- ATTESTATION_PRIVATE_KEY
- ATTESTATION_PUBLIC_KEY
- M4_OTA_KEY
- M4_PRIVATE_KEY
- M4_PUBLIC_KEY
- OTA_KEY
- TA_PRIVATE_KEY
- TA_PUBLIC_KEY

Command Line Syntax

```
$ commander util genkeyconfig --outfile <filename> --device <device>
```

Command Line Input Example

```
$ commander util genkeyconfig --outfile keys.json --device Si917
```

This example generates a file, `keys.json`, containing the key configuration for a Si917 device.

Command Line Output Example

```
Generating symmetric key...
Generating symmetric key...
Generating ECC P256 key pair...
Generating ECC P256 key pair...
Generating ECC P256 key pair...
Key configuration written to keys.json
DONE
```

6.18.5 Generate Certificate

The process of signing files can be done using an intermediate certificate. These certificates can be generated with the `util gencert` command. There are currently two available certificate types: GBL certificates and Secure Boot certificates. If rollback prevention is enabled, the device will not boot if it has seen a certificate with a higher version number. This is set by the `--cert-version` option. The private key corresponding to the `--cert-pubkey` is used to sign the image. The certificate may either be signed directly by providing a signing key with the `--sign` option or unsigned by providing the `--extsign` option.

Command Line Syntax

```
$ commander util gencert --cert-type <cert type> --cert-version <version> --cert-pubkey <key file> [--sign
<key file>|--extsign] --outfile <filename>
```

Command Line Input Example

```
$ commander util gencert --cert-type secureboot --cert-version 1 --cert-pubkey cert_pubkey.pem --sign
signing_key.pem --outfile secureboot_cert.bin
```

In this example the signing key is provided and the certificate is signed directly.

Command Line Output Example

```
Successfully signed certificate
DONE
```

Command Line Input Example

```
$ commander util gencert --cert-type gbl --cert-version 1 --cert-pubkey cert_pubkey.pem --extsign --outfile
gbl_cert.bin
```

In this example an unsigned certificate is created. The signature for the certificate can be created, for example, by a Hardware Security Module (HSM). The certificate can be signed by passing the unsigned certificate and the HSM generated signature to the `util signcert` command.

Command Line Output Example

```
DONE
```

6.18.6 Sign Certificate

Sign a certificate with an externally created signature. You can use the optional `--verify` option to verify the signature by providing the public key corresponding to the private key used to create the signature.

Command Line Syntax

```
$ commander util signcert <cert filename> --cert-type <type> --signature <signature> [--verify <public key
file>] --outfile <filename>
```

Command Line Input Example

```
$ commander util signcert gbl_cert.bin.extsign --cert-type gbl --signature gbl_signature.bin --verify
signing_pubkey.pem --outfile signed_cert.bin
```

Command Line Output Example

```
Successfully verified signature
Successfully signed certificate
DONE
```

6.18.7 Verify Signature

When secure boot is enabled, all code running on the device must be signed. This command can be used as a check to verify that the file was correctly signed, which may help in debugging in case secure boot fails, or as a verification before flashing the image. If the file is signed using an intermediate certificate, the certificate key is used to check the signature of the file. The key given by the `--verify` option is used to verify the signature of the certificate.

Command Line Syntax

```
$ commander util verifysign <input file> --verify <public key file>
```

Command Line Input Example

```
$ commander util verifysign my_application.bin --verify signing_pubkey.pem
```

Command Line Output Example

```
Parsing file my_application.bin...
Found application properties at 0x00000e78
Found certificate in image at location 0x0000b3a4
Successfully verified certificate signature with verification key.
Using certificate key to verify application signature.
Found signature at 0x0000b42c
Successfully verified application signature.
DONE
```

6.18.8 Application Information

Get all available information about an application by parsing the `ApplicationProperties_t` struct in the image. If the file does not have application properties, no information can be extracted from the file.

Command Line Syntax

```
$ commander util appinfo <filename>
```

Command Line Input Example

```
$ commander util appinfo my_application.bin
```

Command Line Output Example

```
Parsing file my_application.bin...
Found application properties in image.
Application protperties info:
Signature location          : 0x0000b42c
Signature type              : ECDSA-P256
Long token section address  : Not set (0x00000000)

Application data info:
If rollback prevention is enabled, the device will not boot if the device has seen an application with a higher version number.
App type                    : The application is an MCU application
App version                 : 0x00000000
Product ID                  : 0x53455f555047524144455f4150500000

Application certificate info:
If rollback prevention is enabled, the device will not boot if the device has seen a certificate with a higher version number.
Certificate located at      : 0x0000b3a4
Certificate version         : 0x00000001
Certificate key                                                                                       :
0x249919c28b28156f19d2e03379b968c8a931aa9b195258e2741da28b686983dd71d0140e9a7b0d7e39de43f592163b8aa38d4e0871f5d2d88b575
Certificate signature                                                                                 :
0x013f2adc310f10f1426db74b503f3612a46ab85c7ce86c967eb965b10f7d24267101192513d9481c49c0eb0b61c1f73392cc6f6d1cd1209a9d58e
DONE
```

6.18.9 Print Section Header Information from an ELF File

Parse and print the section header information from an Executable and Linkable Format (ELF) file.

Command Line Syntax

```
$ commander util elfinfo <filename>
```

Command Line Input Example

```
$ commander util elfinfo my_bootloader.out
```

Displays section header information of ELF file *my_bootloader.out*.

Command Line Output Example

```
Index    Name                     Size      Address     Type
    1    .shstrtab                0x00000111    0x00000000    STRTAB
    2    .strtab                  0x0001e169    0x00000000    STRTAB
    3    .symtab                  0x000243a0    0x00000000    SYMTAB
    4    HEADERS                  0x000000ac    0x00000000    PROGBITS
    5    APP ro                   0x0002ddf4    0x00000200    PROGBITS
    6    SIMEE&LOCKBITS           0x00009000    0x000f7000    NOBITS
    7    ResetHeap                0x00001490    0x20000000    NOBITS
    8    Guard                    0x00000030    0x20001490    NOBITS
    9    APP rw                   0x00002148    0x2003dce0    NOBITS
   10    .debug_abbrev            0x00006325    0x00000000    PROGBITS
   11    .debug_aranges           0x000037ac    0x00000000    PROGBITS
   12    .debug_frame             0x0003a2f5    0x00000000    PROGBITS
   13    .debug_info              0x00063435    0x00000000    PROGBITS
   14    .debug_line              0x00064f5c    0x00000000    PROGBITS
   15    .debug_loc               0x00010fe3    0x00000000    PROGBITS
   16    .debug_macinfo           0x00009941    0x00000000    PROGBITS
   17    .debug_pubnames          0x00007132    0x00000000    PROGBITS
   18    .debug_ranges            0x00003778    0x00000000    PROGBITS
   19    .iar.debug_frame         0x00015349    0x00000000    PROGBITS
   20    .iar.debug_line          0x00020199    0x00000000    PROGBITS
   21    .comment                 0x001d394a    0x00000000    PROGBITS
   22    .iar.rtmodel             0x00000032    0x00000000    PROGBITS
   23    .ARM.attributes          0x0000002e    0x00000000
DONE
```

6.18.10 Get RAM and Flash Usage of an ELF Application

Calculate the static RAM usage and the flash storage usage of an application from an Executable and Linkable Format (ELF) file, and print usage details of the RAM sections.

If the `--map` option is provided with the path to the .map file created when building the application (only GCC map files are supported), the available RAM and flash storage will also be displayed.

If no map file is available, the `--device` option can be provided to let Commander infer the RAM and flash sizes of the device from its part number.

Note: Any changes you might have introduced to the memory regions on your specific device will not be reflected if you are using the `--device` option.

Command Line Syntax

```
$ commander util usage <filename> [--map <filename>|--device <device part no.>] [--include-section <ELF section> --exclude-section <ELF section>]
```

Command Line Input Example

```
$ commander util usage my_elf.out --map my_mapfile.map
```

Command Line Output Example

```
Ram usage       :   262144 /   262144 B (100.00 %)
  .bss          :     3344 B            (  1.28 %)
  .data         :      152 B            (  0.06 %)
  .heap         :   254552 B            ( 97.10 %)
  .stack        :     4096 B            (  1.56 %)

Flash usage     :    23884 /  1564672 B (  1.53 %)
DONE
```

6.18.11 Print Header Information of an RPS File

Parse and print the information contained in the header of an RPS file. The printed information includes (but is not limited to) security settings, signature data, bootloader instructions, flash address, image type, and image size. If the provided RPS file is a combined RPS image, the data for all constituent images is printed sequentially. If the image is encrypted, the bootloader instructions will be unavailable.

RPS files for on-device key upgrades are also supported by this command.

Command Line Syntax

```
$ commander util rpsinfo <filename>
```

Command Line Input Example

```
$ commander util rpsinfo image.rps
```

This command line prints the information contained in the header of 'image.rps'.

Command Line Output Example

```
RPS application image

Application info:
Combined image bit set  : No
Image type              : TA application
Image size              : 0x001986A0 (1672864 B)
Flash address           : 0x00011000
Firmware version        : 0x020101BF
Firmware version ext.   : 0x1610ABFF
Counter                 : 0x00000000 (0)
PSRAM                   : No

Security settings:
Integrity check         : CRC
CRC                     : 0x844D33FA (2219652090)
Encrypted               : No
Signed                  : No

Boot descriptor info:
Boot desc. offset       :     0x0080
IVT offset              : 0x00000000

3 boot descriptor entries found:
Length      :   0x000140 (320)
Destination : 0x00000000

Length      :   0x000CFC (3324)
Destination : 0x00000B04

Length      :   0x01A984 (108932)
Destination : 0x0000E948

DONE
```

6.19 OTA Commands

6.19.1 Create an OTA Bootloader File

Creates a Zigbee Over-the-air (OTA) bootloader file from one or more Gecko Bootloader (GBL) files and writes the output to the specified OTA file.

Command Line Syntax

```
$ commander ota create --upgrade-image <filename> --manufacturer-id <ID> --image-type <image type> --firmware-version <version> --string <text> -o <outfile> [--manufacture-tag <tag ID:filename> -stack-version <version> --credentials <credentials> --destinations <EUI64> --min-hw <version> --max-hw <version>]
```

Command Line Input Example

```
$ commander ota create --upgrade-image example.gbl --manufacturer-id 0x1002 --image-type 0x5678 --firmware-version 0x00000005 --string "Example" -o example.ota
```

Creates an OTA file *example.ota* from the GBL upgrade image *example.gbl*.

Command Line Output Example

```
Initializing OTA file...
Writing header data...
Manufacturer ID : 0x1002
Image Type      : 0x5678
Firmware version: 0x00000005
Stack Version   : 0x0002
Header String   : Example
Writing OTA file ...
DONE
```

6.19.2 Create a Null OTA File

The certification process for the Zigbee Over-the-Air (OTA) Bootload cluster client requires that the manufacturer provides a NULL upgrade file to the test house for testing. A NULL OTA upgrade file does not contain an actual upgrade image inside it (such as a Gecko Bootloader (GBL) file). It is much smaller than a full upgrade image, but otherwise the same as a normal Zigbee OTA file.

You create NULL files by using the `--null` option instead of the `--upgrade-image` option. The `--null` option consists of a tag ID and a tag length. The tag ID should be something other than 0x0000, which Zigbee has defined as "Upgrade Image". The tag length is a number of bytes, usually something small, such as 10. This option generates a sequence of bytes, starting at 0 and incrementing based on the tag length passed in.

Command Line Syntax

```
$ commander ota create --null <tag ID:tag length> --manufacturer-id <ID> --image-type <image type> --firmware-version <version> --string <text> -o <outfile> [--credentials <credentials> --destinations <EUI64> --min-hw <version> --max-hw <version>]
```

Command Line Input Example

```
$ commander ota create --null 0xffff:10 --manufacturer-id 0x110c --image-type 0x5678 --firmware-version 0x0102 --string "NULL OTA file"  -o ~/projects/Binaries/null.ota
```

Creates a NULL OTA file with a tag ID `0xffff` and a tag length of 10 bytes.

Command Line Output Example

```
Initializing OTA file...
Writing OTA file ...
DONE
```

6.19.3 Print OTA File Information

Parses and prints the contents of an Over-the-air (OTA) file.

Command Line Syntax

```
$ commander ota parse <ota file>
```

Command Line Input Example

```
$ commander ota parse example.ota
```

Displays content of the OTA file *example.ota*.

Command Line Output Example

```
Header Magic:              0x0beef11e
Header Version:            0x0100
Header Length:             56 bytes
Header Field Control:      0x0000
Manufacturer ID:           0x110c
ImageType:                 0x0027 (Manufacture Specific)
Firmware Version:          0x01020509
Zigbee stack version:      0x0002 (ZigBee Pro)
Header String:             NULL
Image Size:                121680 bytes
Found 4 tags
   Tag ID:                 0x0000 (Upgrade Image)
   Tag Length:             120572 bytes

   Tag ID:                 0xff01 (Manufacturer Specific)
   Tag Length:             516 bytes

   Tag ID:                 0xff3e (Manufacturer Specific)
   Tag Length:             504 bytes

   Tag ID:                 0xff46 (Manufacturer Specific)
   Tag Length:             8 bytes
DONE
```

6.19.4 Sign an OTA File

The Zigbee Smart Energy Profile requires that the manufacturer signs Over-the-Air (OTA) files. The OTA client must validate the downloaded files prior to installation. Images are signed using certificates issued by Certicom. After the images are signed, the signer's certificate is included automatically as a tag in the OTA file, and a signature tag is added as the last tag in the OTA file. For more information, see *AN714: Smart Energy ECC-Enabled Device Setup Process*.

Note: MacOS does not support OTA signing.

Command Line Syntax

```
$ commander ota create --sign --certificate <certificate>  --upgrade-image <filename> --manufacturer-id <ID> --image-type <image type> --firmware-version <version> --string <text> -o <outfile> [--credentials <credentials> --destinations <EUI64> --min-hw <version> --max-hw <version>]
```

Command Line Input Example

```
$ commander ota create --sign --certificate certificate.txt --upgrade-image example.gbl --manufacturer-id 0x0345 --image-type 0x4567 --firmware-version 0x00000002  --string "Signed OTA file"  -o signed_file.ota
```

Creates a signed OTA file using certificate `certificate.txt`.

Command Line Output Example

```
Creating OTA file...
Writing header data...
Manufacturer ID : 0x0345
Image Type      : 0x4567
Firmware version: 0x00000002
Stack Version   : 0x0002
Header String   : Signed OTA file
Digest: 8DFD32A4C6F3C39E6C152F33A16AEAD2
Signed file using certificate.
Successfully verified signature.
Writing OTA file signed_file.ota...
DONE
```

6.19.5 Create an OTA File for External Signing

Create an Over-the-air (OTA) image to be signed externally. The external certificate is added to the image. The signature can be added to the image using the signcommand.

Command Line Syntax

```
$ commander ota create --extsign --certificate <certificate>  --upgrade-image <filename> --manufacturer-id <ID> --image-type <image type> --firmware-version <version> --string <text> -o <outfile> [--credentials <credentials> --destinations <EUI64> --min-hw <version> --max-hw <version>]
```

Command Line Input Example

```
$ commander ota create --extsign --certificate certificate.txt --upgrade-image example.gbl --manufacturer-id 0x0345 --image-type 0x4567 --firmware-version 0x00000002  --string "Silicon Labs Ota Support"  -o example_file.ota
```

Command Line Output Example

```
Creating OTA file...
Writing header data...
Manufacturer ID : 0x0345
Image Type      : 0x4567
Firmware version: 0x00000002
Stack Version   : 0x0002
Header String   : Silicon Labs Ota Support
Writing OTA file example_file.ota.extsign...
DONE
```

6.19.6 Externally Sign an OTA File

Use the Simplicity Commander `sign` command to append an externally created signature to an Over-the-Air (OTA) file. You must specify the curve used to create the signature using the `--curve` option. Available curves are 163k1 or 283k1.

Command Line Syntax

```
$ commander ota sign <filename> --curve <curve (163k1|283k1)>  --signature <filename> -o <outfile>
```

Command Line Input Example

```
$ commander ota sign example.ota --curve <curve (163k1|283k1)>  --signature signature.txt -o signed_file.ota
```

Appends the externally created signature to the OTA file.

Command Line Output Example

```
DONE
```

6.19.7 Verify Signature of an OTA File

Use the Simplicity Commander `verify` command to verify the signature of an Over-the-Air (OTA) file. You must provide the certificate used to sign the file.

Note: MacOS does not support OTA signature verification.

Command Line Syntax

```
$ commander ota verify <filename> --certificate <certificate>
```

Command Line Input Example

```
$ commander ota verify signed_file.ota --certificate certificate.txt
```

Verifies the signature of the OTA file.

Command Line Output Example

```
Digest: 8ABB04618622595401AD45FA33C7D670
Successfully verified signature
DONE
```

6.19.8 Create an OTA Matter File

Create a Matter Over-the-air (OTA) software update file from an application and write the output to the specified OTA file.

Command Line Syntax

```
$ commander ota create --type matter --input <filename> --vendorid <vendor ID> --productid <product ID> --swversion <version> --swstring <version string> --digest <digest algorithm> [--releasenote <url> --min-sw <version> --max-sw <version>] --output <filename>
```

Supported digest algorithms are

- sha256
- sha384
- sha512
- sha3_224
- sha3_256
- sha3_384
- sha3_512

Command Line Input Example

```
$ commander ota create --type matter --vendorid 0x1234 --productid 0x4321 --swversion 0x300001 --swstring "3.0.1" --input application.bin --digest sha256 --releasenote "https://releasenotes.com" --min-sw 0x300000 --max-sw 0x400000 --output upgrade_file.ota
```

Creates an OTA file *upgrade_file.ota* from the application image *application.bin*.

Command Line Output Example

```
Creating OTA file...
Writing header data...
Vendor ID              : 0x1234
Product ID             : 0x4321
Software Version       : 0x00300001
Software Version String: 3.0.1
Min Software Version   : 0x00300000
Max Software Version   : 0x00400000
Release Note           : https://releasenotes.com
Digest Type            : sha256
Writing OTA file upgrade_file.ota...
DONE
```

6.19.9 Parse a Matter OTA File

Parse and print the contents of a Matter Over-the-air (OTA) software update file. The optional `--output` option extracts the application from the OTA file and writes it to the file specified by the `--output` option.

Command Line Syntax

```
$ commander ota parse <ota file> --type matter [--output <application>]
```

Command Line Input Example

```
$ commander ota parse example.ota --type matter --output my_application.bin
```

Displays content of the OTA file *example.ota* and extracts the application from the OTA file and writes it to *my_application.bin.*

Command Line Output Example

```
Magic: 1beef11e
Total Size  : 977 bytes
Header Size : 105 bytes
Header TLV:
[0] Vendor ID     : 4660     (0x1234)
[1] Product ID    : 17185    (0x4321)
[2] Version       : 3145729  (0x300001)
[3] Version String: 3.0.1
[4] Payload Size  : 856      (0x358)
[5] Min Version   : 3145728  (0x300000)
[6] Max Version   : 4194304  (0x400000)
[7] Release Notes : https://releasenotes.com
[8] Digest Type   : 1        (0x1)
[9] Digest        : 8f259c4727adbe755ec1a49e8bfdbedb3486f53721cae5434efe5d9971eb5d55
Writing application to my_application.bin...
```

6.20 Post-Build Command

6.20.1 Execute a Project Post-Build File

Simplicity Commander takes a project post-build description file in Yaml Ain't Markup Language (YAML) format, produced by Simplicity Studio, and executes sequentially the specified tasks in the file.

Command Line Syntax

```
$ commander postbuild <filename> [--parameter <name:value>]
```

Command Line Input Example

```
$ commander postbuild project_name.slpb --parameter "build_dir:path_to_build_dir"
```

Executes the steps in the post-build pipeline defined in *project_name.slpb*.

Command Line Output Example

```
Parsing file project_name.slbp...
Running task copy...
Running task convert...
Running task GBL create...
Running task OTA create...
DONE
```

The post-build pipeline consists of three sections:

- Parameters: named variables whose value is taken from the command line upon pipeline invocation.
- Constants: named variables whose value is taken from the post-build file itself or a path to another post-build file from which constants are inherited.
- Steps: list of tasks to be invoked, making use of the above declared variables.

See below for an example post-build file:

```
parameters:
  - name: artifact
  - name: build_dir
constants:
  - name: project_name
    value: my_project

steps:
  - task: copy
    input: "{{build_dir}}/{{project_name}}.s37"
    output: "{{artifact}}/{{project_name}}.s37"

  - task: convert
    input: "{{build_dir}}/{{project_name}}.out"
    include-section: P2 ro
    output: "{{artifact}}/{{project_name}}.bin"
```

Tasks

Seven different types of tasks are supported. The tasks are identified with the following names:

- *copy*
- *convert*
- *convert_rps*
- *create_gbl*
- *create_ota*
- *create_rps*
- *usage*

The tables below summarize the required options and optional options for each task.

Table 6.2. *copy*

Required Options
input: <filename>
output: <filename>
Optional Options
export: <constant value>

Table 6.3. *convert*

Required Options
input: <filename>
output: <filename>
Optional Options
export: <constant value>
keyfile: <key file>
crc: <true>
certificate: <certificate file>
include-section: <ELF section>
exclude-section: <ELF section>
signature: <signature file>
verify: <key file>

Table 6.4. *convert_rps*

Required Options
output: <RPS filename>
Optional Options
app: <M4 RPS filename>
taapp: <TA RPS filename>
app-version: <version number>

fw-info: <firmware info>
sign: <key filename>
sha-type: <SHA-XXX>
encrypt: <key filename>
mic: <key filename>
combinedimage: <true>

Table 6.5. *create_gbl*

Required Options
output: <filename>
Optional Options
export: <constant value>
app: <app image>
bootloader: <bootloader image>
seupgrade: <SE upgrade image>
metadata: <metadata bin file>
compress: <app compression algorithm>
certificate: <certificate file>
sign: <key file>
encrypt: <AES key file>
extsign: <true>
include-section: <section>
exclude-section: <section>

Table 6.6. *create_ota*

Required Options
input: <filename> (same as upgrade-image)
output: <filename>
manufacturer-id: <ID>
firmware-version: <version>
image-type: <image type>
string <text>
Optional Options
export: <constant value>
upgrade-image: <filename>
manufacturer-tag: <tag ID>
stack-version: <version>

credentials: <credentials>
destination: <EUI64>
min-hw: <version>
max-hw: <version>
certificate: <filename>
sign: <true>

Table 6.7. *create_rps*

Required Options
input: <application filename>
output: <RPS filename>
Optional Options
address: <address>
app-version: <version number>
include-section: <section>
exclude-section: <section>
fw-info: <firmware info>
sign: <key filename>
sha-type: <SHA-XXX>
encrypt: <key filename>
mic: <key filename>
combinedimage: <true>

Table 6.8. *usage*

Required Options
input: <application ELF filename>
Optional Options
map: <filename>
device: <device part number>
include-section: <ELF section>
exclude-section: <ELF section>

6.21 RPS Commands

SiWx917 devices require that application binaries are converted to RPS images before flashing. Simplicity Commander can be used to convert M4 application binaries to RPS images, apply security features, and to combine multiple RPS images into a single RPS file.

Simplicity Commander's RPS image creation supports bin, hex, SRec and ELF image formats. Commander will prepend an RPS style header to the provided application image, containing information used by the device's bootloader. An application version number may be provided using the `--app-version` option, and additional firmware/device information can be provided using the `--fw-info` option. If you intend on combining one or multiple RPS application images, the `--combinedimage` flag can be provided to prepare the image for combining with other eligible RPS images.

Simplicity Commander also supports creating RPS key images for upgrading on-device M4 keys.

6.21.1 Create an RPS File From a Binary Image

To create an RPS file from a binary image you **must provide** an application start address using the `--address` flag.

Command Line Syntax

```
$ commander rps create <output filename> --app <filename> --address <start address> [--app-version <version no.> --fw-info <firmware info> --combinedimage]
```

Command Line Input Example

```
$ commander rps create output.rps --app app.bin --address 0x08212000
```

This command line creates an RPS file from a binary image with flash address '0x08212000' and saves it to the file named 'output.rps'.

Command Line Output Example

```
Parsing file app.bin...
RPS file successfully created at 'output.rps'
DONE
```

6.21.2 Create an RPS File From an ELF Image

When generating an RPS file from an Execution and Linkable Format (ELF) image, you can use the `--include-section` and `--exclude-section` options to either include or exclude certain ELF sections from the application image of the output RPS file. If neither of these options is provided, Simplicity Commander will include all sections that appear to be part of the application.

You can include or exclude multiple sections by providing the respective options repeatedly.

Command Line Syntax

```
$ commander rps create <output filename> --app <filename> [--include-section <section> --exclude-section <section> --app-version <version no.> --fw-info <firmware info> --combinedimage]
```

Command Line Input Example

```
$ commander rps create output.rps --app app.axf --include-section .text --include-section .data
```

This command line creates an RPS file from the sections '.text' and '.data' of an ELF application file and saves it to the file named 'output.rps'.

Command Line Output Example

```
Including ELF section(s):
  .text
  .data
Parsing file app.axf...
RPS file successfully created at 'output.rps'
DONE
```

6.21.3 Create an RPS File from a Hex/s37 Image

You can create an RPS file from an Intel Hex (hex) image or from a Motorola S-record (s37) image.

Command Line Syntax

```
$ commander rps create <output filename> --app <filename> [--app-version <version no.> --fw-info <firmware info> --combinedimage]
```

Command Line Input Example

```
$ commander rps create output.rps --app app.hex
```

This command line creates an RPS file from a hex image and saves it to the file named 'output.rps'.

Command Line Output Example

```
Parsing file app.hex...
RPS file successfully created at 'output.rps'
DONE
```

6.21.4 Create an RPS File For Upgrading On-Device Key

Creating an RPS key file requires a new key to store on the device, the previous (current) key stored on the device, as well as a private ECDSA key (.pem) for signing the RPS file. Only the device's M4 public key and the M4 OTA key can be upgraded, being denoted by the key types `public` and `OTA`, respectively.

Options `--new-key` and `--prev-key` support keys as plain hex-strings (e.g. '0123456789ABCDEF'), or as .h-files containing comma-separated hexadecimal values (each prefixed with '0x'). If the provided key type is `public`, the new and previous keys can also be provided as .pem-files. Alternatively, an eligible key configuration JSON file can be provided to let Commander collect the required keys automatically.

Command Line Syntax

```
$ commander rps create <output filename> --key-type <'public'|'ota'> --new-key <key> --prev-key <key> --sign <filename>
```

Command Line Input Example

```
$ commander rps create key.rps --key-type 'public' --new-key new-key.h --prev-key old-key.h --sign private-key.pem
```

This command line creates an RPS key file for updating the on-device M4 public key, and saves it to the file named 'key.rps'.

Command Line Output Example

```
Parsing new key 'new-key.h'...
Parsing previous key 'old-key.h'...
Parsing signing key 'private-key.pem'...
Signing image...
Image SHA256: 1c01440a60849ff35f56ed09fb468bbf2f92f3c8d6e50cb5b9c12b4cb38c9df3
R = E03FC4A415E6FEA584F48CC08E1F8EE45090A2CE5E8C176C44720D8314DAEA1C
S = B7CED83970B74B2E75F3E42B229DBA022265BB6E319A777AA9F530380052494B
RPS file successfully created at 'key.rps'.
DONE
```

6.21.5 Create a Secure RPS Application Image

RPS application images support multiple security-related features: AES-ECB-based encryption, AES-CBC MIC integrity check, and ECDSA signatures (SHA-256, SHA-384, and SHA-512). By default, these features are disabled, and a CRC-based integrity check is used on the RPS file contents.

The keys for encryption and MIC are symmetric keys (32 bytes in length), and can be provided as hex strings or as .h-files containing comma-separated hexadecimal values (each prefixed with '0x'). Alternatively, an eligible key configuration JSON file can be provided to let Commander collect the required keys automatically.

Command Line Syntax

```
$ commander rps create <output filename> --app <application filename> --encrypt <key> --mic <key> --sign <key>
[--app-version <version no.> --fw-info <firmware info> --combinedimage]
```

Command Line Input Example

```
$ commander rps create secure-app.rps --app app.hex --encrypt ekey.h --mic mkey.h --sign private-key.pem --sha
SHA-512
```

This command line creates an RPS file from a binary image with flash address '0x08212000' and saves it to the file named 'output.rps'.

Command Line Output Example

```
Parsing file app.hex...
Parsing MIC key 'mkey.h'...
Calculating MIC of image...
Parsing encryption key 'ekey.h'...
Encrypting image...
Parsing signing key 'private-key.pem'...
Signing image...
Image                                                 SHA512:
cd7c5ca70167e91ae22e519e25e8f1f1967879bbfda852e75d77c1c3a54c07cd790a2ddfd54f0a55d065dd964cb1de49afb92f96d86acf52d591e21
R = CE26333E667842859469622C4E35B72B1C1FCA7D148F58FD67F66C70449A4092
S = 91EA3A02A4B7374401A46161869819AA14065FE760C2781466BAD0643AD8FF60
RPS file successfully created at 'secure-app.rps'.
DONE
```

6.21.6 Convert an Existing RPS Application Image

Simplicity Commander can be used to convert already existing non-secure (no encryption, MIC, or signature) RPS images (both ThreadArch (TA) and M4 images) into secure images by applying AES-ECB encryption, AES-CBC MIC integrity check, and ECDSA signatures. Non-secure images can also be modified to support combining with other RPS images by providing the `--combinedimage` flag, which sets the `COMBINED_IMAGE` bit in the RPS header.

M4 RPS images are provided using the `--app` option, whereas TA RPS images are provided using the `--taapp` option.

Command Line Syntax

```
$ commander rps convert <output filename> --app <application filename> | --taapp <application filename> [--encrypt <key> --mic <key> --sign <key> --app-version <version no.> --fw-info <firmware info> --combinedimage]
```

Command Line Input Example

```
$ commander rps convert secure-app.rps --app app.rps --encrypt ekey.h --mic mkey.h --sign private-key.pem --app-version 0x00010209 --combinedimage
```

This command line takes the non-secure M4 RPS 'app.rps' and creates a secure RPS application image with encryption, MIC integrity check, and SHA-512 based signature, and saves it to the file named 'secure-app.rps' The command also sets a new application version number in the RPS header, and it prepares the image for combining.

Command Line Output Example

```
Setting COMBINED_IMAGE flag...
Parsing file app.hex...
Parsing MIC key 'mkey.h'...
Calculating MIC of image...
Parsing encryption key 'ekey.h'...
Encrypting image...
Parsing signing key 'private-key.pem'...
Signing image...
Image SHA256: e53775814dc61c2ecbe14f1b1d9310c8d79ad96681a9f6258cd427cbc9cd6576
R = CE26333E667842859469622C4E35B72B1C1FCA7D148F58FD67F66C70449A4092
S = 91EA3A02A4B7374401A46161869819AA14065FE760C2781466BAD0643AD8FF60
RPS file successfully created at 'secure-app.rps'.
DONE
```

6.21.7 Combine Multiple RPS Images Into a Single RPS File

Using Simplicity Commander, you can combine an M4 RPS application image with a ThreadArch (TA) RPS application image into a single RPS file. For an RPS image to be eligible for combining, the `COMBINED_IMAGE` bit must be set in the header of the image, either during the image's creation, or by converting an already existing non-secure RPS image.

The M4 image is provided via the `--app` option, and is always placed first within the combined image. The TA image is provided using the `--taapp` option.

The combined image can be signed with a private ECDSA key, provided in .pem format.

Command Line Syntax

```
$ commander rps convert <output filename> --app <M4 application filename> --taapp <TA application filename> [--
sign <key filename>]
```

Command Line Input Example

```
$ commander rps convert combined-image.rps --app image1.rps --taapp image2.rps --sign private-key.pem
```

This command line takes the M4 RPS image 'image1.rps' and combines it with the TA RPS image 'image2.rps' into a single RPS image with signature.

Command Line Output Example

```
Combining images...
  Adding image1.rps...
  Adding image2.rps...
Parsing signing key 'private-key.pem'...
Signing combined image...
Image SHA256: e53775814dc61c2ecbe14f1b1d9310c8d79ad96681a9f6258cd427cbc9cd6576
R = CE26333E667842859469622C4E35B72B1C1FCA7D148F58FD67F66C70449A4092
S = 91EA3A02A4B7374401A46161869819AA14065FE760C2781466BAD0643AD8FF60
RPS file successfully created at 'combined-image.rps'.
DONE
```

6.22 VUART Commands

Simplicity Commander supports reading and sending data over Virtual UART (VUART) over IP using the `vuart connect` command. When the command is executed, a TCP socket is opened and connected to the hostname/IP address of the adapter, using port 4900. The command will then allow for sending and receiving data over the VUART line until termination by either pressing CTRL+C, or by meeting one of the conditions described below.

6.22.1 VUART Communications Until Timeout

If the `--timeout` option is used, the command will terminate if no data is received from the target within the specified time (in seconds).

Command Line Syntax

```
$ commander vuart connect <IP or hostname> [--timeout <timeout in s>]
```

Command Line Input Example

```
$ commander vuart connect 10.0.0.1 --timeout 5
```

This command line connects to the target device via VUART and will terminate if no data is received in 5 seconds.

Command Line Output Example

```
Attempting to connect to IP 10.0.0.1 at port 4900...
Connection established!
<data written by the target application>
Timeout: No data received for 5 seconds.
DONE
```

6.22.2 VUART Communications Until a Marker is Found

If the `--endmarker` option is used, the command will terminate after finding the specified string in the incoming VUART data stream.

Command Line Syntax

```
$ commander vuart connect <IP or hostname> [--endmarker <string>]
```

Command Line Input Example

```
$ commander vuart connect 10.0.0.1 --endmarker STOP
```

This command line connects to the target at IP 10.0.0.1 via VUART and terminates if the string 'STOP' is found in the data coming from the target.

Command Line Output Example

```
Attempting to connect to IP 10.0.0.1 at port 4900...
Connection established!
<data written by the target application>
Process complete STOP
End marker 'STOP' found.
DONE
```

6.23 RTT Commands

Simplicity Commander supports reading data from and sending data to the target via SEGGER Real Time Transfer (RTT) using the `rtt connect` command. The communications will be active until terminated by pressing CTRL+C, or if one of the conditions described below is met.

By default, the target will be reset during the initialization of the RTT connection. Providing the `--noreset` option will prevent this.

6.23.1 RTT Communications Until a Marker is Found

If the `--endmarker` option is used, the command will terminate after finding the specified string in the RTT data stream.

Command Line Syntax

```
$ commander rtt connect [--endmarker <string>]
```

Command Line Input Example

```
$ commander rtt connect --endmarker STOP
```

This command line starts RTT communications with the target device and will terminate if the string 'STOP' is received from the target device.

Command Line Output Example

```
RTT successfully initialized.
Searching for RTT block in device memory...
Searching for RTT block in device memory...
RTT buffer 'Terminal' found!
RTT status:    Running
Read buffers:  3
Write buffers: 3
RTT console connected, enter CTRL+C to terminate.
<data written by application>
Process complete STOP
End marker 'STOP' found.
DONE
```

6.23.2 RTT Communications Until Timeout

If the `--timeout` option is used, the command will terminate if no data is received from the target within the specified time (in seconds).

Command Line Syntax

```
$ commander rtt connect [--timeout <timeout in s>]
```

Command Line Input Example

```
$ commander rtt connect --timeout 20
```

This command line starts RTT communications with the target device and will time out after 20 seconds if no more data is received.

Command Line Output Example

```
RTT successfully initialized.
Searching for RTT block in device memory...
Searching for RTT block in device memory...
RTT buffer 'Terminal' found!
RTT status:    Running
Read buffers:  3
Write buffers: 3
RTT console connected, enter CTRL+C to terminate.
<data written by application>
Timeout: No data received for 20 seconds.
DONE
```

6.23.3 RTT Communications Over Virtual Terminals

Commander supports reading data from 16 virtual RTT terminals (indexed 0-15), specified by the `--terminal` option.

The default virtual terminal used is virtual terminal 0.

Command Line Syntax

```
$ commander rtt connect [--terminal <virtual terminal index>]
```

Command Line Input Example

```
$ commander rtt connect --terminal 4
```

This command line starts RTT communications with the target device and listens to RTT virtual terminal 4.

Command Line Output Example

```
RTT successfully initialized.
Searching for RTT block in device memory...
Searching for RTT block in device memory...
RTT buffer 'Terminal' found!
RTT status:    Running
Read buffers:  3
Write buffers: 3
RTT console connected, enter CTRL+C to terminate.
<data written by application>
Connection terminated by user.
DONE
```

6.23.4 RTT Communications With a Custom RTT Buffer Configuration

By default Commander will try to locate the RTT block automatically. However, the RTT block address may be specified explicitly by providing the `--blockaddress` option. The default read buffer (RTT up-buffer) and write buffer (RTT down-buffer) indices default to 0, but may be set by providing the `--readbuffer` and `--writebuffer` options, respectively.

Note: Commander will look for the specified RTT read buffer during the RTT block search. If this buffer was not initialized by the target application before the search was started, the RTT block search may fail.

Command Line Syntax

```
$ commander rtt connect [--blockaddress <address> --readbuffer <buffer index> --writebuffer <buffer index>]
```

Command Line Input Example

```
$ commander rtt connect --blockaddress 0x10002000 --readbuffer 1 --writebuffer 2
```

This command line starts RTT communications with the target device and looks for RTT read buffer 1 in the RTT block located at address 0x10002000 in the device memory. After the connection is established, data will be read from the RTT up-buffer of index 1, and data will be written to the target via the RTT down-buffer of index 2.

Command Line Output Example

```
RTT successfully initialized.
Searching for RTT block at address 0x10002000...
RTT buffer 'CustomBuffer' found!
RTT status:    Running
Read buffers:  3
Write buffers: 3
RTT console connected, enter CTRL+C to terminate.
<data written by application>
Connection terminated by user.
DONE
```

6.24 Serial Commands

Simplicity Commander can be used to transfer files to SiWx917 devices over the adapter's serial (VCOM) port, using the Embedded Kermit protocol. These files include M4 or ThreadArch(TA) application images, as well as tokens for unlocking debug access to either device core.

All `serial` commands require a physical data connection (i.e. USB cable) between the host computer and the adapter. The serial port can be explicitly provided using the `--serialport` option; this will also bypass all J-Link specific handling of the adapter board/kit. If the J-Link serial number is provided via the `--serialno` option, the adapter's serial port is automatically inferred by Commander.

`serial` file transfers can be aborted by pressing CTRL+C.

Note: Prior to running any of the `serial` commands, the target device **must** be booted in ISP mode.

6.24.1 Load an RPS Application Over Serial

RPS images can be loaded to either the M4 or the TA core of the SiWx917 device using the `serial load` command. The core to which the application is loaded is determined by the contents of the image's RPS header.

Command Line Syntax

```
$ commander serial load <RPS filename> [--serialport <port name>]
```

Command Line Input Example

```
$ commander serial load app.rps --serialport COM4
```

This command line loads the application image 'app.rps' to the device, using serial port COM4.

Command Line Output Example

```
Using serial port 'COM4' for file transfers.
Initializing M4 firmware upgrade...
Sending file(s):
  app.rps
M4 firmware was successfully uploaded.
DONE
```

6.24.2 Lock Debug Access to M4/TA Core

Simplicity Commander can lock debug access via the JTAG interface to both the M4 and the TA core of SiWx917 devices.

Providing the `--token` option, a token can be created upon locking, which can be used for unlocking debug access to the device later. Creating this token requires a private ECDSA key provided via the `--key` option, used for signing the token.

For the sake of redundancy, in case the process of saving the token file should fail, the complete token raw data is always printed to the console.

Optionally, 7 bytes (exactly) of user data can be provided using the `--userdata` option, to be stored in the token file. These bytes are provided as a hex string.

Note: After the `serial lock` command has been run, the device needs to be power cycled for the debug access changes to take effect.

Command Line Syntax

```
$ commander serial lock <'M4'|'TA'> [--token <filename> --key <filename> --userdata <hex string> --serialport
<port name>]
```

Command Line Input Example

```
$ commander serial lock TA --token unlock.token --key private-key.pem --userdata AABBCCDDC0FFEE --serialport
COM4
```

This command line locks the JTAG debug access to the TA core of the device, and saves the debug access unlock token to the file 'unlock.token'. The bytes `AABBCCDDC0FFEE` is stored in the user data section of the token, and the token is signed by the 'private-key.pem' ECDSA key.

Command Line Output Example

```
Using serial port 'COM4' for file transfers.
Initializing debug lock...
Nonce generated by the device: 7A3FFEEFBB48EFC7EB7617E90E7FDDEE
Debug access locked.
Parsing signing key 'private-key.pem"...
Debuglock token raw data:
7a3ffeefbb48efc7eb7617e90e7fddee74aabbccddc0ffee304502210094cf2c372ad5f3a9fd2b46b2b0c25a7d6d853e3aab1093bcdd9746b35648b
Debuglock token written to 'unlock.token'.
Debug access will be locked after the device is reset.
DONE
```

6.24.3 Unlock Debug Access to M4/TA Core With Existing Token

If the unlock token from the last time the device was locked is available, debug access to the M4/TA core over the JTAG interface can be unlocked by using the `serial unlock` command.

Note: After the `serial unlock` command has been run, the device needs to be power cycled for the debug access changes to take effect.

Command Line Syntax

```
$ commander serial unlock <'M4'|'TA'> --token <filename> [--serialport <port name>]
```

Command Line Input Example

```
$ commander serial unlock TA --token unlock.token --serialport COM4
```

This command line unlocks debug access to the TA core, by sending the token 'unlock.token' to the device.

Command Line Output Example

```
Using serial port 'COM4' for file transfers.
Verifying debuglock token 'unlock.token'...
Initializing debug unlock...
Sending file(s):
  unlock.token
Debug access will be unlocked after device is reset.
DONE
```

6.24.4 Unlock Debug Access to M4/TA Core Without Existing Token

Simplicity Commander can unlock the debug access to the M4/TA core without the token from when the device was last locked. This is effectively done by locking the device (thus generating a new token), immediately followed by unlocking the device using this new, intermediate token. The `--key` option is required in this configuration, as the intermediate token needs to be signed using a private ECDSA key. The `--userdata` option is optional.

Note: After the `serial unlock` command has been run, the device needs to be power cycled for the debug access changes to take effect.

Command Line Syntax

```
$ commander serial unlock <'M4'|'TA'> --key <filename> [--userdata <hex string> --serialport <port name>]
```

Command Line Input Example

```
$ commander serial unlock TA --key private-key.pem --serialport COM4
```

This command line unlocks debug access to the TA core by creating a temporary token file that is signed by the ECDSA key 'private-key.pem'.

Command Line Output Example

```
Using serial port 'COM4' for file transfers.
Initializing debug unlock...
Nonce generated by the device: 7A3FFEEFBB48EFC7EB7617E90E7FDDEE
Parsing signing key 'private-key.pem"...
Debuglock token raw data:
7a3ffeefbb48efc7eb7617e90e7fddee74aabbccddc0ffee304502210094cf2c372ad5f3a9fd2b46b2b0c25a7d6d853e3aab1093bcdd9746b35648b
Debuglock token written to 'unlock.token'.
Sending file(s):
  unlock.token
Debug access will be unlocked after the device is reset.
DONE
```

6.25 Manufacturing Commands

Note: Since Simplicity Commander version 1.16.3, the `manufacturing` keyword has been changed to `mfg917`. The `manufacturing` alias is (as of version 1.16.4) still available, but it is considered deprecated and may therefore be removed without notice in any future release of Simplicity Commander.

Simplicity Commander provides tools for provisioning SiWx917 devices with initial configuration and keys for signing/encryption. The tools are suitable for use in a manufacturing setting and can be used to write and read data to and from regions related to both M4 and ThreadArch(TA) cores.

For writing/erasing manufacturing data, or running the key provisioning processes (including its initialization), Simplicity Commander depends on a custom TA firmware being loaded to the device. Loading this firmware can be skipped by providing the `--skipload` flag. This option must only be provided if you are completely certain that the TA firmware is already loaded and running on your device, and the use of the `--skipload` flag is therefore generally discouraged.

In case an external flash configuration is used, the flash pinset index can be set using the `--pinset` option.

For configuring radio-/network co-processor (RCP/NCP) devices, Simplicity Commander communicates via a proprietary serial protocol to an interface device, which in turn communicates with the RCP/NCP device via serial peripheral interface (SPI) or secure digital input output (SDIO) commands. In this case, Simplicity Commander requires that you also provide the `--serialinterface` option along with the `--device` option. In case you are running multiple `mfg917` commands in sequence and without resetting the target device in between, you may also provide the `--skipinit` option to skip initializing the target device's SPI/SDIO interface.

Note: `mfg917` commands are currently only supported on SiWx917 devices.

6.25.1 List Available Memory Regions

Simplicity Commander can be used to read/write/erase certain memory regions, including but not limited to:

- M4 and TA master boot records (MBR)
- Efuse
- Boot descriptors
- Keys

To see the available memory regions on your device as well as their ability for producing JSON output, provide the `--list` option with either of the commands `read`, `write`, or `erase`.

Command Line Syntax

```
$ commander mfg917 read|write|erase --list
```

Command Line Input Example

```
$ commander mfg917 read --list
```

This command line lists all available memory regions that can be read using the `manufacturing` commands.

Command Line Output Example

```
Region name     JSON supported   Description
bfc             No               Bootloader firmware controller
bootdesc        No               Boot descriptors
certs           No               Certificates
efuse           Yes              Efuse
efusecopy       Yes              Efuse copy
efuseipmu       Yes              Efuse intelligent power management unit
keydesctable    No               Key descriptor table
m4fmccf         No               M4 core flash memory controller (common flash)
m4fmcdf         No               M4 core flash memory controller (dual flash)
m4ipmucf        Yes              M4 core intelligent power management unit (common flash)
m4ipmudf        Yes              M4 core intelligent power management unit (dual flash)
m4mbrcf         Yes              M4 core master boot record (common flash)
m4mbrdf         Yes              M4 core master boot record (dual flash)
m4ptinfocf      No               M4 production information (common flash)
m4ptinfodf      No               M4 production information (dual flash)
pufactkey       No               PUF activation key
rompatch        No               ROM patches
signature       No               Signature
statickeys      No               Static keys
storeconf       No               Store configuration
tafmc           No               ThreadArch flash memory controller
tafwimg         No               ThreadArch firmware image
taipmu          Yes              ThreadArch core intelligent power management unit
tambr           Yes              ThreadArch core master boot record

DONE
```

6.25.2 Read Memory Region Data From Device

The read data can be output to the terminal or saved to a raw binary file. For supported regions, the output data can also be stored in a human-readable JSON file. If JSON output is supported and desired, provide an output filename with '.json' extension.

Command Line Syntax

```
$ commander mfg917 read <region> [--out <filename]
```

Command Line Input Example

```
$ commander mfg917 read tambr --out device-mbr.json
```

This command line reads the `tambr` region of the device and stores the data output to the file 'device-mbr.json' in JSON format.

Command Line Output Example

```
Reading data from region: tambr
Reading 496 bytes from 0x04000000
Writing JSON...
Manufacturing data saved to file 'device-mbr.json'
DONE
```

6.25.3 Read Specific Fields From Memory Region

If you are interested in reading only certain data fields from a JSON-supported region, singular values can be extracted by providing the `--property` option along with which field and/or sub-field to read from. The `--property` option can be provided multiple times, and the fields/sub-fields are given on the format 'field:sub-field'. If you provide only the field name for a data field that also contains named sub-fields, all the sub-fields will be included in the output.

If JSON output is supported and desired, provide an output filename with '.json' extension to store the selected fields in a JSON file.

Command Line Syntax

```
$ commander mfg917 read <region> [--out <filename> --property <field name[:sub-field name]>]
```

Command Line Input Example

```
$ commander mfg917 read tambr --property m4_clk_configs --property flash_size --property psram_misc_configs:spi_mode
```

This command line reads the selected fields from the `tambr` region of the device.

Command Line Output Example

```
Reading data from region: tambr
Reading 496 bytes from 0x04000000

flash_size = 64
m4_clk_configs:clk_source = 0
m4_clk_configs:div_factor = 0
psram_misc_configs:spi_mode = 2
DONE
```

6.25.4 Write Memory Region Data to Device

Manufacturing data that is to be written to a memory region of the device must be provided in either binary format, or, for eligible memory regions, as a JSON file.

Note: Proceed with caution when writing manufacturing data to your device; writing unsupported/erroneous data/configurations may result in your device being rendered unrecoverable!

If the provided data is a binary file, the data is written to the device verbatim.

If a JSON file is provided, the region data is instead updated; the memory region is first read from the device, and the fields present in the JSON file are then used to update the corresponding region data. Lastly, the updated region data is written back to the device. If applicable, Simplicity Commander will by default also update the region's CRCs/integrity checks when changes have been made. This can be omitted by providing the `--nocrc` option.

The `--dryrun` flag can be added to output the new region data to the terminal instead of writing it to the device.

Command Line Syntax

```
$ commander mfg917 write <region> --data <filename> [--pinset <index> --skipload --nocrc --dryrun]
```

Command Line Input Example

```
$ commander mfg917 write tambr --data mbr-updates.json
```

This command line writes an updated `tambr` to the device by first reading the `tambr` region of the device, and then applying the changes from the fields provided in the 'mbr-updates.json' JSON file, finally writing this updated `tambr` data back to the device.

Command Line Output Example

```
Reading 496 bytes from 0x00400000...
Reading JSON...
Writing data to region: tambr
<process output shortened for documentation>
Data loaded successfully
Region 'tambr' was successfully written to device.
DONE
```

6.25.5 Erase Memory Region Data From Device

Simplicity Commander can be used to erase the data in a memory region, using the `mfg917 erase` command.

Command Line Syntax

```
$ commander mfg917 erase <region> [--pinset <index> --skipload]
```

Command Line Input Example

```
$ commander mfg917 erase efusecopy
```

This command line erases the contents in the `efusecopy` region of the device.

Command Line Output Example

```
Writing data to region: efusecopy
<process output shortened for documentation>
Data loaded successfully
Region 'efusecopy' was successfully erased.
DONE
```

6.25.6 Dump Configuration Data of Device

Simplicity Commander supports dumping all data regions containing configuration data to a zip archive, using the `mfg917 dump` command.

Note: For zip file compression functionality, the `mfg917 dump` command requires Microsoft PowerShell version 5.0 or above on Windows, and the `zip` and `unzip` system utilities on Linux/Mac

Command Line Syntax

```
$ commander mfg917 dump <zip archive filename>
```

Command Line Input Example

```
$ commander mfg917 dump device_data.zip
```

This command line reads and dumps all data regions containing configuration data into the 'device_data.zip' archive file.

Command Line Output Example

```
Reading 40960 bytes from 0x0400f000
Reading 184320 bytes from 0x047cf000
Reading 1024 bytes from 0x40012000
Reading 1024 bytes from 0x040003e0
Reading 58 bytes from 0x40012181
Reading 88 bytes from 0x04000300
Reading 496 bytes from 0x04000000
Reading 1024 bytes from 0x041c0000
Reading 496 bytes from 0x04000000
Reading 58 bytes from 0x041b0258
Reading 496 bytes from 0x04000000
Reading 496 bytes from 0x041b0000
Reading 496 bytes from 0x04000000
Reading 3 bytes from 0x041b0292
Reading 1192 bytes from 0x04002000
Reading 200 bytes from 0x04000238
Reading 72 bytes from 0x040001f0
Reading 4096 bytes from 0x04005000
Reading 3072 bytes from 0x04010400
Reading 1024 bytes from 0x04010000
Reading 58 bytes from 0x04000561
Reading 496 bytes from 0x04000000
Zip archive created at "path/to/device_data.zip".
DONE
```

6.25.7 Initialize PUF And Generate Activation Code

To enable security features (encryption/MIC integrity check/signing) on your SiWx917 device, the device's PUF first needs to be initialized and an activation code must be generated on the device. This can be done via the `mfg917 init` command.

Providing a TA MBR (as a binary file) using the `--mbr` option is optional, and its purpose is to provide information about the destination address of the activation code. If required, updates to the TA MBR can be applied by providing a JSON file with the `--data` option. If the `--mbr` option is omitted, the default activation code address is used. Providing `--mbr 'default'` will use the default TA MBR for your device, based on the provided device part number using the `--device` option.

Note: After the `mfg917 init` command has been run, the device needs to be power cycled for any changes to take effect.

Command Line Syntax

```
$ commander mfg917 init [--mbr <filename|'default'> --data <filename> --pinset <index> --skipload]
```

Command Line Input Example

```
$ commander mfg917 init --mbr default --device SiWG917M111LGTBA
```

This command line initializes the device's PUF and generates an activation code, using a default TA MBR for the device part number 'SiWG917M111LGTBA'.

Command Line Output Example

```
Using default MBR for SiWG917M111LGTBA...
<process output shortened for documentation>
Activation code generated successfully
DONE
```

6.25.8 Provision Security Keys to the Device

Provisioning device keys is done using the `mfg917 provision` command, by providing a key configuration JSON file containing the keys you want to store on your device with the `--keys` option. Supported keys for storing on the device are M4/TA public keys and M4/TA OTA keys. In addition, a private attestation key is required for the provisioning sequence.

If you don't want to provision any keys during the provisioning sequence, the `--keys` option may be omitted.

If required, updates to the TA MBR can be applied before writing it to the device by providing a JSON file with the `--data` option.

Note: After the `mfg917 provision` command has been run, the device needs to be power cycled for any changes to take effect.

Command Line Syntax

```
$ commander mfg917 provision --mbr <filename|'default'> [--keys <filename> --data <filename> --pinset <index>
--skipload]
```

Command Line Input Example

```
$ commander mfg917 provision --keys keys.json
```

This command line provisions the keys contained in the JSON file 'keys.json' to the device.

Command Line Output Example

```
Reading MBR from the connected device...
<process output shortened for documentation>
Found valid activation code address in MBR: 0x00002000
<process output shortened for documentation>
Intrinsic keys generated successfully.
Programming TA OTA Key...
<process output shortened for documentation>
Key successfully stored
Programming M4 OTA Key...
<process output shortened for documentation>
Key successfully stored
Programming TA Public Key...
<process output shortened for documentation>
Key successfully stored
Programming M4 Public Key...
<process output shortened for documentation>
Key successfully stored
Programming attestation Key...
<process output shortened for documentation>
Key successfully stored
Programming TA MBR...
<process output shortened for documentation>
Data loaded successfully
Programming key descriptor table...
<process output shortened for documentation>
Data loaded successfully
DONE
```

6.25.9 Get Information About Device Configuration

Key information about the device configuration can be extracted using the `mfg917 info` command. Simplicity Commander will read certain regions of your device (including the TA MBR and the Efuse regions), parse the information, and present key information like the device's MAC address, flash size, and current security parameters.

Command Line Syntax

```
$ commander mfg917 info
```

Command Line Input Example

```
$ commander mfg917 info
```

This command line extracts information about the current configuration of the target device.

Command Line Output Example

```
Reading 1024 bytes from 0x040003e0
OPN: SiWG917M111MGTBA
Reading 1024 bytes from 0x040003e0
WiFi MAC address: 6C5CB1C43F90
Reading 496 bytes from 0x04000000
Flash size: 8MB
Flash variant: 0
Flash type: 0
Application region start address: 0x081b0000
Application code start address: 0x081c2000
Application region end address: 0x083effff
Common flash configuration.
Mode: SOC
NWP roll-back prevention disabled.
NWP digital signature validation disabled.
NWP firmware encryption disabled.
NWP secure boot disabled.
Application roll-back prevention disabled.
Application digital signature validation disabled.
Application code encryption disabled.
Application secure boot disabled.
DONE
```

7. Software Revision History

The following subsections summarize the new features of Simplicity Commander by version number.

7.1 Version 1.16

2023-09-28

Added these Simplicity Commander commands:

Section 6.25 Manufacturing Commands
- 6.25.3 Read Specific Fields From Memory Region
- 6.25.6 Dump Configuration Data of Device
- 6.25.9 Get Information About Device Configuration

Modified these Simplicity Commander commands:

Section 6.13 Advanced Energy Monitor Commands
- 6.13.1 Measure Average Current in a Time Window

Section 6.18 Util Commands
- 6.18.10 Get RAM and Flash Usage of an ELF Application

Section 6.25 Manufacturing Commands
- 6.25.1 List Available Memory Regions
- 6.25.2 Read Memory Region Data From Device
- 6.25.4 Write Memory Region Data to Device
- 6.25.5 Erase Memory Region Data From Device
- 6.25.7 Initialize PUF And Generate Activation Code
- 6.25.8 Provision Security Keys to the Device

7.2 Version 1.15

2023-05-16

Added these Simplicity Commander commands:

Section 6.18 Util Commands

- 6.18.10 Get RAM and Flash Usage of an ELF Application
- 6.18.11 Print Header Information of an RPS File

Section 6.19 OTA Commands

- 6.19.8 Create an OTA Matter File
- 6.19.9 Parse a Matter OTA File

Section 6.21 RPS Commands

- 6.21.4 Create an RPS File For Upgrading On-Device Key
- 6.21.5 Create a Secure RPS Application Image
- 6.21.6 Convert an Existing RPS Application Image
- 6.21.7 Combine Multiple RPS Images Into a Single RPS File

Section 6.22 RTT Commands

- 6.23.1 RTT Communications Until a Marker is Found
- 6.23.2 RTT Communications Until Timeout
- 6.23.3 RTT Communications Over Virtual Terminals
- 6.23.4 RTT Communications With a Custom RTT Buffer Configuration

Section 6.23 VUART Commands

- 6.22.2 VUART Communications Until a Marker is Found
- 6.22.1 VUART Communications Until Timeout

Section 6.24 Serial Commands

- 6.24.1 Load an RPS Application Over Serial
- 6.24.2 Lock Debug Access to M4/TA Core
- 6.24.3 Unlock Debug Access to M4/TA Core With Existing Token
- 6.24.4 Unlock Debug Access to M4/TA Core Without Existing Token

Section 6.25 Manufacturing Commands

- 6.25.1 List Available Memory Regions
- 6.25.2 Read Memory Region Data From Device
- 6.25.4 Write Memory Region Data to Device
- 6.25.5 Erase Memory Region Data From Device
- 6.25.7 Initialize PUF And Generate Activation Code
- 6.25.8 Provision Security Keys to the Device

Modified these Simplicity Commander commands:

Section 6.14 Serial Wire Output Read Commands

- 6.14.1 Configure SWO Speed
- 6.14.3 Read SWO Until a Marker Is Found
- 6.14.4 Dump Hex Encoded SWO Output

Section 6.20 Post-Build Command

- 6.20.1 Execute a Project Post-Build File

Section 6.21 RPS Commands

- 6.21.1 Create an RPS File From a Binary Image
- 6.21.2 Create an RPS File From an ELF Image
- 6.21.3 Create an RPS File from a Hex/s37 Image

7.3 Version 1.14

2022-11-18

Added these Simplicity Commander commands:

Section 6.13 Advanced Energy Monitor Measure Commands
- 6.13.1 Measure Average Current in a Time Window
- 6.13.2 Log Current Measurements as Time Series Data
- 6.13.3 Start Logging on Trigger Event

Section 6.19 OTA Commands
- 6.19.4 Sign an OTA File
- 6.19.5 Create an OTA File for External Signing
- 6.19.6 Externally Sign an OTA File
- 6.19.7 Verify Signature of an OTA File

Section 6.21 RPS Commands
- 6.21.1 Create an RPS File From a Binary Image
- 6.21.2 Create an RPS File From an ELF Image
- 6.21.3 Create an RPS File from a Hex/s37 Image

Modified these Simplicity Commander commands:

Section 6.17 Security Commands
- 6.17.1 Get Device Status

Section 6.20 Post-Build Command
- 6.20.1 Execute a Project Post-Build File

7.4 Version 1.13

2022-05-06

Added these Simplicity Commander commands.

Section 6.5 Convert and Modify Commands
- 6.5.10 Add a Trust Zone Decryption Key
- 6.5.11 Extract Sections from ELF Files

Section 6.7 GBL Commands
- 6.7.13 Create a GBL File from an ELF File
- 6.7.15 Create a GBL File with Version Dependencies
- 6.7.14 Create an Encrypted GBL File with an Unencrypted Secure Element Upgrade File

Section 6.18 Util Commands
- 6.18.9 Print Section Header Information from an ELF File

Section 6.19 OTA Commands
- 6.19.1 Create an OTA Bootloader File
- 6.19.3 Print OTA File Information
- 6.19.2 Create a Null OTA File

Section 6.20 Post-Build Command
- 6.20.1 Execute a Project Post-Build File

7.5 Version 1.12

2021-11-15

Added debug modes:

6.8.4 Adapter Debug Mode Command

7.6 Version 1.11

2021-05-14

Added a general command option:

3.3.7 Timestamp (`--timestamp`)

7.7 Version 1.10

2020-11-19

Revised the implementation details for these commands:

6.1.1 Flash Image File

6.3 Memory Read Commands

2020-05-08

- Resolved issue related to external flash in certain scenarios
- Added these Security commands:

  6.17.19 Write AES Decryption Key

  6.17.20 Read Device Certificates
  6.17.21 Vault Device Attestation
- Added VCOM port information to output from adapter probe command

7.8 Version 1.9

2020-03-09

- Added this Convert and Modify File command:

  Signing an Application for Secure Boot using an Intermediary Certificate
- Added these Security commands:

  Disable Tamper

  Read User Configuration
- Added these Util commands:

  Key Generation

  Generating a Signing Key

  Key to Token

  Generate Certificate

  Sign Certificate

  Verify Signature

  Application Information
- Added clarifying details in 4.5 Memory Regions regarding mass erase and differences between EFR32 Series 1 and Series 2 devices.

7.9 Version 1.8

2019-11-21

- Added the `security` commands that support Secure Element functionality. See 5. Security Overview and 6.17 Security Commands for details.
- Improved GUI
  - Support for EFR32xG2x devices
  - Added flash map feature
  - Added blank check feature

7.10 Version 1.7

2018-11-28

- Added CTUNE manufacturing token commands.
- Added support for EFR32XG21 devices.
- Added support for generating Secure Element upgrade GBL files.

7.11 Version 1.5

2018-10-02

- Added support for analyzing the memory usage of the application using an Application Address Table (AAT).

7.12 Version 1.4

2018-09-19

- Added support for module part numbers (e.g. BGM111) as --device parameter
- Module part numbers will be read from the device when it exists (new modules only)

7.13 Version 1.3

2018-08-14

- Added support for manipulating and writing NVM3 data.
- Added support for custom token definition files in any location.

7.14 Version 1.2

2018-03-23

- Added support for creating GBL images using the LZMA compression algorithm.

7.15 Version 1.1

2018-01-19

- Added support for writing CRC32 to an image as a means of integrity check when not using Secure Boot.
- Added the nvm3 command which supports reading NVM3 data from a device and parsing an image file containing NVM3 data.

7.16 Version 1.0

2017-11-28

- Added support for EM3xx devices.

7.17 Version 0.25

2017-06-09

Added support for lz4 compression of GBL files:

- `gbl create --compress lz4`

7.18 Version 0.24

2017-04-25

Added commands that support the Gecko Bootloader Security features:

- `convert --secureboot`
- `gbl keygen --type ecc-p256`
- `gbl keyconvert`
- `gbl create`

  `--bootloader` option

  `--sign` option

  `--extsign` option
- `gbl sign`

7.19 Version 0.22

2017-03-03

Added commands that support the Gecko Bootloader (GBL) file format:

- `gbl create`
- `gbl parse`
- `gbl keygen`

7.20 Version 0.21

2017-02-02

Added commands:

- `ebl create`
- `ebl parse`

Deprecated and hid these commands that only support version 2 of the EBL format:

- `ebl encrypt`
- `ebl decrypt`

These commands have been replaced by `ebl create` and `ebl parse` which support both version 2 and 3 of the EBL format.

Changed command:

- Creating and parsing EBL files using the `convert` command has been deprecated, but still supports parsing and creating EBL v2 files for backwards compatibility. New applications should use the `ebl create` and `ebl parse` commands instead.

7.21 Version 0.16

2016-06-16

Added commands:

- `aem measure`
- `adapter ip`
- `swo read`

7.22 Version 0.15

2016-04-27

Added commands:

- `extflash`
- `adapter reset`
- `adapter dbgmode`

7.23 Version 0.14

2016-02-05

Added commands:

- `device lock`
- `device protect`
- `device pageerase`
- `device recover`

7.24 Version 0.13

Not released

- Added `tokenheader` command.

7.25 Version 0.12

2016-01-20

- Added support for EFR32 custom tokens.

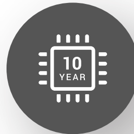
7.26 Version 0.11

2016-01-15

Initial release.

Smart. Connected.
Energy-Friendly.

IoT Portfolio
www.silabs.com/products



Quality
www.silabs.com/quality

Support & Community
www.silabs.com/community

Disclaimer
Silicon Labs intends to provide customers with the latest, accurate, and in-depth documentation of all peripherals and modules available for system and software implementers using or intending to use the Silicon Labs products. Characterization data, available modules and peripherals, memory sizes and memory addresses refer to each specific device, and "Typical" parameters provided can and do vary in different applications. Application examples described herein are for illustrative purposes only. Silicon Labs reserves the right to make changes without further notice to the product information, specifications, and descriptions herein, and does not give warranties as to the accuracy or completeness of the included information. Without prior notification, Silicon Labs may update product firmware during the manufacturing process for security or reliability reasons. Such changes will not alter the specifications or the performance of the product. Silicon Labs shall have no liability for the consequences of use of the information supplied in this document. This document does not imply or expressly grant any license to design or fabricate any integrated circuits. The products are not designed or authorized to be used within any FDA Class III devices, applications for which FDA premarket approval is required or Life Support Systems without the specific written consent of Silicon Labs. A "Life Support System" is any product or system intended to support or sustain life and/or health, which, if it fails, can be reasonably expected to result in significant personal injury or death. Silicon Labs products are not designed or authorized for military applications. Silicon Labs products shall under no circumstances be used in weapons of mass destruction including (but not limited to) nuclear, biological or chemical weapons, or missiles capable of delivering such weapons. Silicon Labs disclaims all express and implied warranties and shall not be responsible or liable for any injuries or damages related to use of a Silicon Labs product in such unauthorized applications. **Note: This content may contain offensive terminology that is now obsolete. Silicon Labs is replacing these terms with inclusive language wherever possible. For more information, visit www.silabs.com/about-us/inclusive-lexicon-project**

Trademark Information
Silicon Laboratories Inc.®, Silicon Laboratories®, Silicon Labs®, SiLabs® and the Silicon Labs logo®, Bluegiga®, Bluegiga Logo®, EFM®, EFM32®, EFR, Ember®, Energy Micro, Energy Micro logo and combinations thereof, "the world's most energy friendly microcontrollers", Redpine Signals®, WiSeConnect , n-Link, ThreadArch®, EZLink®, EZRadio®, EZRadioPRO®, Gecko®, Gecko OS, Gecko OS Studio, Precision32®, Simplicity Studio®, Telegesis, the Telegesis Logo®, USBXpress® , Zentri, the Zentri logo and Zentri DMS, Z-Wave®, and others are trademarks or registered trademarks of Silicon Labs. ARM, CORTEX, Cortex-M3 and THUMB are trademarks or registered trademarks of ARM Holdings. Keil is a registered trademark of ARM Limited. Wi-Fi is a registered trademark of the Wi-Fi Alliance. All other products or brand names mentioned herein are trademarks of their respective holders.

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

www.silabs.com