# UG520: Software Project Generation and Configuration with SLC-CLI

The Silicon Labs Configurator (SLC) example projects describe a single software application (usually made up of multiple components plus application code) that can be used to generate an IDE project. The SLC Command Line Interface (SLC-CLI) tool, among other things, resolves project and component dependencies and generates a project for a specified embedded target and build system (for example, IAR Embedded Workbench or GNU tools via a Makefile). This user's guide provides references to the most common operations done with SLC-CLI.

**KEY POINTS**

- Installation
- Tool configuration
- Project Operations
- General Options
- Working with SDKs and Extensions

# 1 Introduction

The Silicon Labs Configurator (SLC) is a metadata specification for the Gecko SDK (GSDK). It also describes methods of creating and configuring embedded software projects for Silicon Labs IoT devices using this metadata. Software is grouped into components (defined by .slcc files) that may provide features and/or require features provided by other components. Example projects (.slcp) describe a single software application (usually made up of multiple components plus application code) that can be used to generate an IDE project. See the SLC Specification for details about SLC.

The SLC Command Line Interface (SLC-CLI) tool resolves project and component dependencies and generates a project for a specified embedded target and build system (for example, IAR Embedded Workbench or GNU tools via a Makefile), among other things.

SLC-CLI is provided as a downloadable .zip file for three operating systems:

- Windows® OS
- MacOS® X
- Linux® OS

SLC-CLI may be used with the following SDKs and platforms:

- Gecko Bootloader
- Gecko Platform
- The Gecko SDK (the suite of Silicon Labs SDKs)

Example projects (defined in .slcp files) are installed with the SDK in a directory under the Gecko SDK (GSDK) installed directory. The location varies depending on the SDK.

- Amazon AWS: <GSDKpath>\app\amazon\example
- Bluetooth and Bluetooth Mesh SDKs: <GSDKpath>\app\bluetooth\example
- OpenThread SDK: <GSDKpath>\protocol\openthread\sample-apps
- 32-Bit MCU SDK: <GSDKpath>\app\mcu_example
- Proprietary (Flex) SDK: <GSDKpath>\app\flex\example\<Connect or RAIL>
- Gecko Bootloader: <GSDKpath>\platform\bootloader\sample-apps
- GSDK Platform: <GSDKpath>\app\common\example
- Wi-SUN SDK: <GSDKpath>\app\wisun\example
- Z-Wave SDK: <GSDKpath>\protocol\z-wave\apps
- Zigbee SDK: <GSDKpath>\protocol\zigbee\app

SLC-compatible SDKs may also support extensions that may include example projects as well as components. By default, extensions are installed into the “extension” folder at the root of an SDK.

Extension: <*GSDKpath*>*extension*\<*extension_name*>

# 2 Installation

## 2.1 Requirements

The SLC-CLI .zip files are available here:

- https://www.silabs.com/documents/login/software/slc_cli_windows.zip
- https://www.silabs.com/documents/login/software/slc_cli_mac.zip
- https://www.silabs.com/documents/login/software/slc_cli_linux.zip

In addition to the SLC-CLI .zip file and the Gecko SDK, you will need:

- Python 3.3 or higher.
- pip (download option with Python)
- Java 64-bit JVM version 11 or higher, available through Amazon Correto. Note that some files, such as the Windows .msi files, can be found on the releases page.

## 2.2 Installing the CLI

1. Unpack the SLC-CLI zip file.
2. Run `pip install -r requirements.txt`.
   - If you get an environment error, run `pip install --user -r requirements.txt`.
   - If you are running slc with a Python version other than your system default, add the Python path to the command, for example: `your/python/path -m pip install -r requirements.txt`.
3. (Optional) To call SLC-CLI from anywhere in your system, add the path to the downloaded SLC-CLI command line to your PATH.
4. (Optional) All project operations require a path to the GSDK installation directory. To make these operations easier, you may want to create an environment variable for the GSDK location.

## 2.3 Other Tools

SLC-CLI provides a number of options for generating project files. In order to build an application image you also need a compiler tool-chain, such as GCC or IAR.

In order to flash the image to a target device you need Simplicity Commander. Simplicity Commander enables you to complete these essential tasks:

- Flash an application.
- Configure the application image.
- Manage the target device.

Download an Operating System-specific Simplicity Commander zip file here: https://www.silabs.com/developers/mcu-programming-options.

For instructions on using Simplicity Commander, see UG162: Simplicity Commander Reference Guide.

# 3 Usage

`slc --help` provides details on usage and a list of available commands. `slc <command> -h` shows all options for the command.

Run `slc <command> <command options>` to use the default Python installation, or `your\python\path slc ...` to use a different version of Python.

## 3.1 SLC-CLI Configuration

SLC-CLI operations are based on the context of a specific SLC-compatible SDK. It is recommended to first configure SLC-CLI to use a specific SDK by default.

1. Configure SLC-CLI to a specific GSDK location, for example:

   ```
   slc configuration --sdk users\<NAME>\SimplicityStudio\SDKs\gecko_sdk
   ```

   Then all commands that use an SDK will use this configured location. If you do not do this, you must specify the SDK path with the `--sdk` option each time you issue a command, such as `generate` discussed below.
2. If using GNU toolchain from the command line (for example, with a GNU Make build system), first configure your GCC location.

   ```
   slc configuration -gcc=\path\to\your\GNU\ARM\embedded\toolchain
   ```

   Note, if you do not already have a GNU toolchain installed, you can download the proper version (aligned with what your SDK supports) from here: https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm/downloads.

Example SLC-CLI configuration on MacOS for GNU toolchain with default GSDK installation directory:

```
slc configuration --sdk=~/SimplicityStudio/SDKs/gecko_sdk --gcc-toolchain=/Applications/ARM
```

| Operation | Example | Description |
|---|---|---|
| **configuration** | `slc configuration --sdk="C:\sdk\sdk.slcs"` | Sets the given SDK to be default available (unless explicitly overridden in other commands). Any command that requires an SDK parameter will no longer require it, and will instead use this configured default. This either points directly to an .slcs file, or it points to a folder containing an .slcs file. |
| | `--editor` | Sets the external editor used by the editor command. This is mainly intended as a shortcut for loading up .slcc files for components. See the editor documentation in section 3.4.2 SDK Operations for more details.<br>Known supported editors are Atom and Notepad++. |
| | `-gcc, --gcc_toolchain "C:\path\to\gcc"` | Sets the default GCC toolchain path for use with the Makefile generator. This should point to the directory that contains the bin folder. |

### 3.2 General Options

These options can be specified for any action and must appear after the command. For instance, `slc configuration --cli-config file.cfg -sdk=/sdk/path` is correct.

| Option | Example | Description |
|---|---|---|
| `-v --verbose` | `-v 1` | Accepts 0 or 1<br>Sets verbose levels. 0 is no verbosity. Higher levels provide more logging information. |
| `--cli-config` | `--cli-config /home/myhome/configfile.cfg` | Allows customization of where the configuration information, set via `slc configuration` and subcommands, is stored. Overrides the defaults and can be used for every command if the defaults are not properly working in your environment. This must be a file. |
| `-wrk, --working-directory` | `--working-directory /work/here` | The working directory that the command line should assume it was called from, such as for relative path resolution. In most cases, this should not need to be overridden. |

### 3.3 Working with Projects

This section assumes you have configured the GSDK location as described above.

`slc generate <path\to\example.slcp>` generates a project from an existing .slcp file, such as an SDK example. The path to the example is either the fully defined path, or the path relative to the calling location.

Key options are:

`-d <destination>` (optional) specifies the destination for the generated project. If not specified, the project is generated to the source.slcp location.

`-np` generates a new project by copying the .slcp file and all files defined in it into the destination location. All file references in the .slcp are updated to point to the destination location. Any sources that should be highlighted are shown in the SLC-CLI output.

`-name=<generated-name>` specifies a different generated project name. Otherwise the name of the source .slcp file is used.

`--with <device|board>` customizes the generated project for the target specified by the full part number or board ID, for example “EFR32BG22C224F512IM40” or “brd4184b”.

To generate a new project with a new name for all supported toolchains:

```
slc generate \path\to\example.slcp -np -d <project destination> -name=<new name> --with <board or device_that_supports_project>
```

A number of files are generated that can be used with different tools. For example, to build the project with Make (if Make is in your path):

```
make -f <project>.Makefile
```

Examples:

Windows: Generate for all toolchains, for EFR32MG12P232F512GM68 device:

```
slc generate C:\Users\<user>\SimplicityStudio\SDKs\gecko_sdk\app\bluetooth\exam-
ple\soc_empty\soc_empty.slcp -np -d c:\test-soc-empty\ -name=test-soc-empty --with
EFR32MG12P232F512GM68
```

MacOS: Generate, build (GNU Make/GCC), and flash (Simplicity Commander) project to Thunderboard Sense 2 (BRD4166A):

```
$ GSDK=~/SimplicityStudio/SDKs/gecko_sdk
$ slc configuration --sdk=$GSDK --gcc-toolchain=/Applications/ARM
$ slc generate $GSDK/app/common/example/blink_baremetal -np -d blinky -name=blinky -o Makefile
  --with brd4166a
$ cd blinky
$ make -f blinky.Makefile
```

```
$ commander flash build/debug/blinky.hex
```

### 3.3.1 Project Operation Options

All project-level operations listed in section 3.2.2 Project Operations can accept the same basic arguments enumerated here.

| Option | Required | Example | Description |
|---|---|---|---|
| `-p, --project-file` | yes | `-p blink.slcp` | The actual project file that any project operations will be working against. |
| `--with` | no | `--with brd3200c,micriumos`<br>`--with pwm:led0:led1,brd2200a` | Comma-separated list of components to include in the project in addition to components enumerated by the .slcp file itself and in addition to any auto-computed dependencies.<br>If a component is instantiable, then the instance names must be supplied separated by ':', with the first of the ':' separated list being the actual id, and subsequent ones being instance names. |

### 3.3.2 Project Operations

Project operations always specify an SDK to load from as well as a project file *<project_name>.slcp* to draw from. You must specify an SDK (`-s` or `--sdk`) for every project operation unless you have configured a default SDK.

| Operation | Example/Arguments | Description |
|---|---|---|
| **generate** | `generate -p="blink/blink.slcp" -d="blink/output/blink_project"` | Generates the project to the given destination. By default this links all sources.<br>Destination is not required. If not specified, then the slcp directory is used. |
| | `--require-clean-project` | If the slcp parser finds a potential problem and issues a warning, generation is halted.<br>Examples: a project refers to a component not in the SDK , or a project uses deprecated names (like name instead of id for component listing, or name instead of project_name for the project's default name) |
| | `-cp, --copy-sources` | Copies all files referenced by this project, selected components, and any other running tools (Pin Tool, etc.). By default, no files are copied. |
| | `-cpproj, --copy-proj-sources` | Copies all files referenced by the project and links any SDK sources. This can be combined with `-cpsdk`. |
| | `-cpsdk, --copy-sdk-sources` | Copies all files referenced by the selected components and links any project sources. This can be combined with `-cpproj`. |
| | `-np, --new-project` | Runs the new project creation layout. This runs a standard generation step and then copies over any project sources (slcp, config folder, etc.), fixing any paths in the slcp file to point to the new location of the files. This defaults to `-cpproj`. |
| | `-name, --project-name` | Overrides the project name in the slcp. This determines some output file names and the generated binary names. |
| | `-tlcn, --toolchain` | Generates for the specified toolchain. The toolchains are not treated in the same way as components and so do not appear selected in the project. The current valid selections are `gcc` and `iar`. |

| Operation | Example/Arguments | Description |
|---|---|---|
| | `-o, --output-type` | The output of the generation, effectively what kinds of files should be generated, and for what tools/IDEs. You may list more than one type, separated by commas. |
| | `-lfewp, --list-files-ewp` | Forces the IAR EWP generator to define all SLC contributed sources in the *.ewp file as well as the *.ipcf file. This mimics the IAR Project Connection support in IAR Embedded Workbench and is useful for Linux builds of IAR, which do not support *.ipcf files. |
| | `--generator-timeout` | Overrides the default 30s timeout for each generation cycle. One call to `generate` may include multiple generation cycles, so this total generation time could be multiples of this timeout. |
| **validate-project** | `validate-project -p="blink/blink.slcp"` | Validates if a project would generate, essentially a dry run of generation. If the project is invalid, more information (such as a dependency tree) is output, indicating what and why it failed to auto-select all required dependencies, and what was missing. Python validation is not run by default. |
| | `--config-location <cfg_location>` | Tells the configuration validation scripts to run using the specified config folder for the project. Configuration files on that path will contribute their configuration values. The rule is that, if a configuration value appears in a config file on this path, it is considered the ultimate user configuration and overrides any default values. |
| **summarise** | `summarise --project blink.slcp` | Shows a summary of all components that make up a project, including both those explicitly set in the .slcp, and those implicitly brought in via dependency resolution. |
| | `--why` | Summary of implicit components will be replaced with a long list of implicit components that include why they were brought in (the component that needed them, and the API rule.) In many cases, the component name and API ID are identical, but essentially the left side of 'for' is the component ID, and the right side is the API(s). |
| | `--api` | An additional list is provided at the end of the summary showing every API that is provided by the project. In other words, every selected component put together will form a total set of available API. |
| **choices** | `choices --project blink.slcp -max 42` | Shows a list of potential candidate components that provide missing APIs required by the project that could not otherwise be solved by auto-selection, and draws attention to any that are recommended (color consoles will be in green, plain consoles will show an '*' asterisk).<br>Candidates are split between those that provide the APIs unconditionally and those that provide conditionally. Always use `slc examine` for more detailed information. Components are sorted according to what provides the most, although this can mean that a completely different kind of choice with fewer APIs may be hidden by having too low a 'max' value. |
| | `-max` | Indicates that the `choices` command shows only 'max' choices (separate maximums for conditionally/unconditionally provided lists). If set to 0 or negative values, shows all choices. By default, if left unspecified = 20. |
| **graph** | `graph --project blink.slcp`<br>`graph --project blink.slcp --validate --focus "emlib_common, cmsis_core"` | Shows a dependency graph instead of the inverted dependency system that `summarise --why` shows. This dependency graph is represented as a tree, where the first occurrence of a component shows that component's dependencies as well, but subsequent occurrences show only a '^', indicating that a deeper dive has been done earlier in the tree. This is essentially a compromise solution, since dependency graphs can otherwise be very web-like. Color and non-color systems both support drawing attention in their own way to different things.<br>Any component that provides an API other than itself will have that API listed after its name if that API is responsible for providing something another component in this project required.<br>Any component that appears as a 'child' to itself will show a green '[Cycle]' phrase. Cycles do not prevent dependency management from working. This merely calls attention to them. |

| Operation | Example/Arguments | Description |
|---|---|---|
| | `--validate` | Option to show validation issues in-line with the dependency graph. Validation issues are prefaced with an '!' and, if color is supported, are red. |
| | `--focus` | Draws attention to a specific component/component(s) via either color or text, wherever they appear in the tree. |
| **clone** | `clone --project blink.slcp --target micriumos_kernel` | Clones the `--target` component into a custom components folder (either uses the first enumerated path in the .slcp, or creates and adds a default custom folder, if one does not already exist) and adds it to the project's selected components. The .slcc is copied and transformed to add clone information such as the timestamp of the clone, and the author field is populated if one is provided.<br>Cloned components have all relevant file references copied over. This includes Configuration Files, Source Files, Includes (only those listed in file_list), Validation Scripts and Libraries, Template Files, and Local Libraries.<br>Cloned component IDs are auto-assigned and will be unique across the current SDK and any other custom components that may exist, but labels and description data are unchanged. |
| | `--author "Fozzy Bear"` | If the optional author option is specified, the cloned component's author field in the .slcc is updated. |
| | `--clone-folder wakawaka` | If the optional `clone-folder` option is specified, it clones into that specific custom component directory (and adds it to the .slcp custom search paths). |
| **upgrade** | `upgrade blink.slcp` | Upgrades the given project in place. If no upgrade rules report verification is needed or an upgrade is impossible, this modifies the project and the config folder in the same folder as the project. Does nothing no config folder exists or the upgrade cannot complete properly. .bak files are created as backups for all files affected by the upgrade. |
| | `--dry-run` | Runs the upgrade up to the point where the temporary configuration files that have been modified would be merged into the project, but stops short. It reports any issues upgrading, but even if there none, it will not actually modify the project. |
| | `--verified` | If any upgrade rules indicate that 'verification is required', passing this allows the upgrade to take place. |

### 3.4 General Options

These options can be specified for any action and must appear after the command. For instance, `slc configuration --cli-config file.cfg -sdk=/sdk/path` is correct.

| Option | Example | Description |
|---|---|---|
| `-v --verbose` | `-v 1` | Accepts 0 or 1<br>Sets verbose levels. 0 is no verbosity. Higher levels provide more logging information. |
| `--cli-config` | `--cli-config /home/myhome/configfile.cfg` | Allows customization of where the configuration information, set via `slc configuration` and subcommands, is stored. Overrides the defaults and can be used for every command if the defaults are not properly working in your environment. This must be a file. |
| `-wrk, --working-directory` | `--working-directory /work/here` | The working directory that the command line should assume it was called from, such as for relative path resolution. In most cases, this should not need to be overridden. |

### 3.5 Working with SDKs and Extensions

These commands provide information about the components (.slcc files) included in an SDK or extension.

### 3.5.1 SDK Options

These options are shared among all SDK commands.

| Option | Example | Description |
|---|---|---|
| `-s --sdk` | `-s '/sdk/gsdk'` | Indicates the location of the primary SDK to use. This option overrides the global SDK configuration. If no SDK is configured (see above for **Configuration**) then this option becomes required. This either points directly to the .slcs file, or it points to a folder containing that file. |
| `--slccverify-release` | | Turns on release verification, which is a stricter form of .slcc checking. By default, certain warnings are suppressed in dev mode for one reason or another but will appear in this mode. Library files that are missing will be reported. |

### 3.5.2 SDK Operations

| Operation | Example/Arguments | Description |
|---|---|---|
| **where** | `where micriumos` | Displays the location of a component based on the name. This is a subset of the information in **examine**. |
| **examine** | `examine micriumos` | Displays information about a component, such as where it is defined in the SDK, sources, provided APIs, and all additional metadata defined in the yaml itself. |
| **validate** | `validate sdk/platform/component/rtx.slcc` | Validates the .slcc file only, without validating anything else in the SDK. This ensures it is both proper yaml, and that it makes no semantic errors (such as missing required fields, misnamed fields, junk fields, or incorrect types). |
| **show-available** | `show-available toolchains [component option]` | Shows available toolchains and/or project types to use with **generate** or all distinct values for a component option (quality, provides, category, etc.). |
| **editor** | `editor emlib_emu` | Opens the configured external editor (see **configuration**) to the .slcc file that houses the component in question. An editor must be configured first for this to work. |
| **prune** | `prune --with brd2204a,EFM32GG11B820F2048GL192` | Returns a list of component IDs that are not conflicting with the given components. This is most effectively used to filter out components that would not work on certain hardware. Unlike **slc choices**, this is not intended to help fix validation errors, but rather to show everything that one could possibly add to a project given the current constraints.<br>This uses the `project-level --with` commands. A project may still be specified, however, in which case the total components in the project are treated as the input to this command.<br>Note: Using `--with` instead of an .slcp means including the components on the command line only. No dependencies are automatically resolved! If used directly with .slcp files, project dependencies are resolved as normal. |
| | `--trace` | After displaying all available components that could be added, shows a (typically very long) section afterwards of everything that was filtered out due to conflicts of dependencies, and then a list of APIs that are subsequently unavailable. The unavailable API list includes 1 to many OR-listings of requirements that would have had to have existed for the API to be acceptable for this project. |

# Smart. Connected. Energy-Friendly.


**IoT Portfolio**
www.silabs.com/products

**Quality**
www.silabs.com/quality

**Support & Community**
www.silabs.com/community



SILICON LABS

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

**www.silabs.com**