Simplicity Silicon Labs Configurator (SLC)

Overview

Overview of the SLC-CLI

The Silicon Labs Configurator (SLC) is a metadata specification for the Simplicity SDK. 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, 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), among other things. This section provides references to the most common operations done with SLC-CLI.

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

- Windows
- MacOS
- Linux

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

- Gecko Bootloader
- Gecko Platform
- The Simplicity SDK

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

- Amazon AWS: <SiSDKpath>\app\amazon\example
- Bluetooth SDK: <SiSDKpath>\app\bluetooth\example
- Bluetooth Mesh SDK: <SiSDKpath>\app\btmesh\example
- OpenThread SDK: <SiSDKpath>\protocol\openthread\sample-apps
- 32-Bit MCU SDK: <SiSDKpath>\app\mcu_example
- Proprietary (Flex) SDK: <SiSDKpath>\app\flex\example\<Connect or RAIL>
- Gecko Bootloader: <SiSDKpath>\platform\bootloader\sample-apps
- SDK Platform: <SiSDKpath>\app\common\example
- Wi-SUN SDK: <SiSDKpath>\app\wisun\example
- Z-Wave SDK: <SiSDKpath>\protocol\z-wave\apps
- Zigbee SDK: <SiSDKpath>\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: *<SDKpath>/extension/<extension_name>*

Release Notes

Silicon Labs Configurator 6.0.15 (Jan 22, 2026) Release Notes

Release Summary

Key Features

Added in 6.0.15

- Added Installer version information on main page.

Bug Fixes

- None

Chip Enablement

- This release supports Series 2 and Series 3 devices.

Removed/Deprecated Features

- None

Known Issues and Limitations

- None

User Guide

Overview

This user guide includes the following sections:

- Installing SLC
- Using SLC-CLI
- SLC Project Generation Examples

Install SLC-CLI

Install SLC-CLI

To install SLC-CLI with SLT-CLI, see [Install and Configure Packages].

Other Tools

SLC-CLI provides a number of options for generating project files. The compiler toolchain needed to build an application image with SLC is installed by SLT. See [Install and Configure Packages].

To flash the image to a target device, you need Simplicity Commander. Simplicity Commander lets you complete these essential tasks:

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

Simplicity Commander is installed with SLT and is one of the recommended packages, but you can also 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 the Simplicity Commander Reference Guide.

Use the SLC-CLI

Use the SLC-CLI

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

SLC-CLI Configuration

The SLC gets its configuration from a `.slconf` file, if it finds the file. If the `.slconf` file is specified or a default file is found in the working directory or project, depending on the nature of the comment, then the configuration options are used to provide the default values for any command line options. See SLT for more information.

In general, data in [core] and [slc] are considered, and option names can be any of the --double-dash forms of any option. This alternative to providing cli arguments on the command line itself applies to any option in any command. If multiple --double-dash names are used that refer to the same command but have different values in the configuration file or its included files, then the final value of the option is not defined. Do not do this.

SLC-CLI operations are based on the context of a specific SLC-compatible SDK. It is recommended to use an .slconf file for the slc configuration, as that is easier and more efficient, but manual configuration is possible and was required before .slconf support was added. When manually configuring SLC-CLI, it is recommended to first configure SLC-CLI to use a specific SDK by default.

1. Configure SLC-CLI to a specific Simplicity SDK location, for example:
   `slc configuration --sdk users\<NAME>\SimplicityStudio\SDKs\simplicity_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. See the release notes for your SDK version for compatible compilers.

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

```
slc configuration --sdk=~/SimplicityStudio/SDKs/simplicity_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. Supplying an empty string (--sdk="") clears this value. May be set to a relative value. If this value is set and an override path is provided in the generation operation, the override is used.
	`-e, --examine`	Prints the current configuration file and its location.
	`-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 and lib folders.

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`	Sets verbosity levels. Values: SILENT, ERROR, WARNING, INFO, DEBUG, TRACE
`--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. The output of slc-cli help provides the default location on your system.
`-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.

Work with Projects

This section assumes you have configured the Simplicity SDK 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\simplicity_sdk\app\bluetooth\example\bt_soc_empty\bt_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):

```
$ SISDK=~/SimplicityStudio/SDKs/simplicity_sdk
$ slc configuration --sdk=$SISDK --gcc-toolchain=/Applications/ARM
$ slc generate $SISDK/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
```

Project Operation Options

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

Option	Required	Example	Description
`-p, --project-file`	yes, may be a positional parameter instead	-p blink.slcp	The actual project file that any project operations will be working against.
`--with`	no	--with brd3200c,micriumos --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. 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.

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. Destination is not required. If not specified, then the slcp directory is used.
	`-d, --export-destination=DESTINATION`	Location for the generated project. If not provided, the .slcp or .slcw location is used. Generation location is used even for non-generating commands, as information there can still contribute to the project state.
	`-name, --project-name`	Overrides the project name in the slcp. This determines some output file names and the generated binary names.
	`-o, --output-type=OUTPUT_TYPE`	The output of the generation, effectively what kinds of files should be generated and for what tools/IDEs.
	`--force`	Forces operation to continue in the presence of validation errors.
	`--require-clean-project`	para:[If the slcp parser finds a potential problem and issues a warning, generation is halted.] para: [Examples: a project refers to a component not in the SDK , or a project uses deprecated metadata terms (like name instead of id for component listing, or name instead of project_name for the project's default name)]
	`--configuration=CONFIGURATION_OVERRIDE_MAP`	Provides the same function as ‘configuration’ in the .slcp but command line options take precedence. Takes a comma-delimited list of changes in the format “configuration_name:value”. The same configuration cannot be referenced more than once on the command line.
	`-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. Can be combined with -cpsdk.
	`-cpsdk, --copy-sdk-sources`	Copies all files referenced by the selected components and links any project sources. Can be combined with -cpproj.
	`-nocp, --no-copy`	Causes all files to be linked into the generation location. Only generated files will exist in the generation location. Cannot be used with any other copy operation.
	`-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.

Operation	Example/Arguments	Description
generate	`-d, --export-destination=DESTINATION`	Location for the generated project. If not provided, the .slcp or .slcw location is used. Generation location is used even for non-generating commands, as information there can still contribute to the project state.
	`-name, --project-name`	Overrides the project name in the slcp. This determines some output file names and the generated binary names.
	`-o, --output-type=OUTPUT_TYPE`	The output of the generation, effectively what kinds of files should be generated and for what tools/IDEs.
	`--force`	Forces operation to continue in the presence of validation errors.
	`--require-clean-project`	para:[If the slcp parser finds a potential problem and issues a warning, generation is halted.] para: [Examples: a project refers to a component not in the SDK , or a project uses deprecated metadata terms (like name instead of id for component listing, or name instead of project_name for the project's default name)]
	`--configuration=CONFIGURATION_OVERRIDE_MAP`	Provides the same function as ‘configuration’ in the .slcp but command line options take precedence. Takes a comma-delimited list of changes in the format “configuration_name:value”. The same configuration cannot be referenced more than once on the command line.
	`-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. Can be combined with -cpsdk.
	`-cpsdk, --copy-sdk-sources`	Copies all files referenced by the selected components and links any project sources. Can be combined with -cpproj.
	`-nocp, --no-copy`	Causes all files to be linked into the generation location. Only generated files will exist in the generation location. Cannot be used with any other copy operation.
	`-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.
	`-extmpl, --export-templates=EXPORT_TEMPLATES_FOLDER`	Location for custom export templates. Each exporter requires its own particular name for the files in this directory, but generally they are looking for the same files defined in the global exporter directory (<install>/developer/exporter_templates) using 'custom.<extension>' (e.g., custom.project.mak)

Operation	Example/Arguments	Description
generate	-name, --project-name	Overrides the project name in the slcp. This determines some output file names and the generated binary names.
	-o, --output-type=OUTPUT_TYPE	The output of the generation, effectively what kinds of files should be generated and for what tools/IDEs.
	--force	Forces operation to continue in the presence of validation errors.
	--require-clean-project	para:[If the slcp parser finds a potential problem and issues a warning, generation is halted.] para: [Examples: a project refers to a component not in the SDK , or a project uses deprecated metadata terms (like name instead of id for component listing, or name instead of project_name for the project's default name)]
	--configuration=CONFIGURATION_OVERRIDE_MAP	Provides the same function as ‘configuration’ in the .slcp but command line options take precedence. Takes a comma-delimited list of changes in the format “configuration_name:value”. The same configuration cannot be referenced more than once on the command line.
	-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. Can be combined with -cpsdk.
	-cpsdk, --copy-sdk-sources	Copies all files referenced by the selected components and links any project sources. Can be combined with -cpproj.
	-nocp, --no-copy	Causes all files to be linked into the generation location. Only generated files will exist in the generation location. Cannot be used with any other copy operation.
	-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.
	-extmpl, --export-templates=EXPORT_TEMPLATES_FOLDER	Location for custom export templates. Each exporter requires its own particular name for the files in this directory, but generally they are looking for the same files defined in the global exporter directory (<install>/developer/exporter_templates) using 'custom.<extension>' (e.g., custom.project.mak)
	-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.
	-tools, --config-tools=<tools>	Specifies the tools to be generated based on the file extension. If this is not provided, then all active tools will generate. You may list more than one tool extension separated by commas.

Operation	Example/Arguments	Description
generate	-o, --output-type=OUTPUT_TYPE	The output of the generation, effectively what kinds of files should be generated and for what tools/IDEs.
	--force	Forces operation to continue in the presence of validation errors.
	--require-clean-project	para:[If the slcp parser finds a potential problem and issues a warning, generation is halted.] para: [Examples: a project refers to a component not in the SDK , or a project uses deprecated metadata terms (like name instead of id for component listing, or name instead of project_name for the project's default name)]
	--configuration=CONFIGURATION_OVERRIDE_MAP	Provides the same function as ‘configuration’ in the .slcp but command line options take precedence. Takes a comma-delimited list of changes in the format “configuration_name:value”. The same configuration cannot be referenced more than once on the command line.
	-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. Can be combined with -cpsdk.
	-cpsdk, --copy-sdk-sources	Copies all files referenced by the selected components and links any project sources. Can be combined with -cpproj.
	-nocp, --no-copy	Causes all files to be linked into the generation location. Only generated files will exist in the generation location. Cannot be used with any other copy operation.
	-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.
	-extmpl, --export-templates=EXPORT_TEMPLATES_FOLDER	Location for custom export templates. Each exporter requires its own particular name for the files in this directory, but generally they are looking for the same files defined in the global exporter directory (<install>/developer/exporter_templates) using 'custom.<extension>' (e.g., custom.project.mak)
	-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.
	-tools, --config-tools=<tools>	Specifies the tools to be generated based on the file extension. If this is not provided, then all active tools will generate. You may list more than one tool extension separated by commas.

Operation	Example/Arguments	Description
generate	`--force`	Forces operation to continue in the presence of validation errors.
	`--require-clean-project`	para:[If the slcp parser finds a potential problem and issues a warning, generation is halted.] para: [Examples: a project refers to a component not in the SDK , or a project uses deprecated metadata terms (like name instead of id for component listing, or name instead of project_name for the project's default name)]
	`--configuration=CONFIGURATION_OVERRIDE_MAP`	Provides the same function as ‘configuration’ in the .slcp but command line options take precedence. Takes a comma-delimited list of changes in the format “configuration_name:value”. The same configuration cannot be referenced more than once on the command line.
	`-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. Can be combined with -cpsdk.
	`-cpsdk, --copy-sdk-sources`	Copies all files referenced by the selected components and links any project sources. Can be combined with -cpproj.
	`-nocp, --no-copy`	Causes all files to be linked into the generation location. Only generated files will exist in the generation location. Cannot be used with any other copy operation.
	`-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.
	`-extmpl, --export-templates=EXPORT_TEMPLATES_FOLDER`	Location for custom export templates. Each exporter requires its own particular name for the files in this directory, but generally they are looking for the same files defined in the global exporter directory (<install>/developer/exporter_templates) using 'custom.<extension>' (e.g., custom.project.mak)
	`-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.
	`-tools, --config-tools=<tools>`	Specifies the tools to be generated based on the file extension. If this is not provided, then all active tools will generate. You may list more than one tool extension separated by commas.
	`-o, --output-typ` e	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.

Operation	Example/Arguments	Description
generate	`--require-clean-project`	para:[If the slcp parser finds a potential problem and issues a warning, generation is halted.] para: [Examples: a project refers to a component not in the SDK , or a project uses deprecated metadata terms (like name instead of id for component listing, or name instead of project_name for the project's default name)]
	`--configuration=CONFIGURATION_OVERRIDE_MAP`	Provides the same function as ‘configuration’ in the .slcp but command line options take precedence. Takes a comma-delimited list of changes in the format “configuration_name:value”. The same configuration cannot be referenced more than once on the command line.
	`-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. Can be combined with -cpsdk.
	`-cpsdk, --copy-sdk-sources`	Copies all files referenced by the selected components and links any project sources. Can be combined with -cpproj.
	`-nocp, --no-copy`	Causes all files to be linked into the generation location. Only generated files will exist in the generation location. Cannot be used with any other copy operation.
	`-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.
	`-extmpl, --export-templates=EXPORT_TEMPLATES_FOLDER`	Location for custom export templates. Each exporter requires its own particular name for the files in this directory, but generally they are looking for the same files defined in the global exporter directory (<install>/developer/exporter_templates) using 'custom.<extension>' (e.g., custom.project.mak)
	`-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.
	`-tools, --config-tools=<tools>`	Specifies the tools to be generated based on the file extension. If this is not provided, then all active tools will generate. You may list more than one tool extension separated by commas.
	`-o, --output-typ` e	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.

Operation	Example/Arguments	Description
generate	`--configuration=CONFIGURATION_OVERRIDE_MAP`	Provides the same function as 'configuration' in the .slcp but command line options take precedence. Takes a comma-delimited list of changes in the format "configuration_name:value". The same configuration cannot be referenced more than once on the command line.
	`-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. Can be combined with -cpsdk.
	`-cpsdk, --copy-sdk-sources`	Copies all files referenced by the selected components and links any project sources. Can be combined with -cpproj.
	`-nocp, --no-copy`	Causes all files to be linked into the generation location. Only generated files will exist in the generation location. Cannot be used with any other copy operation.
	`-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.
	`-extmpl, --export-templates=EXPORT_TEMPLATES_FOLDER`	Location for custom export templates. Each exporter requires its own particular name for the files in this directory, but generally they are looking for the same files defined in the global exporter directory (<install>/developer/exporter_templates) using 'custom.<extension>' (e.g., custom.project.mak)
	`-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.
	`-tools, --config-tools=<tools>`	Specifies the tools to be generated based on the file extension. If this is not provided, then all active tools will generate. You may list more than one tool extension separated by commas.
	`-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=TIMEOUT_SECONDS`	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.

Operation	Example/Arguments	Description
generate	`-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. Can be combined with -cpsdk.
	`-cpsdk, --copy-sdk-sources`	Copies all files referenced by the selected components and links any project sources. Can be combined with -cpproj.
	`-nocp, --no-copy`	Causes all files to be linked into the generation location. Only generated files will exist in the generation location. Cannot be used with any other copy operation.
	`-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.
	`-extmpl, --export-templates=EXPORT_TEMPLATES_FOLDER`	Location for custom export templates. Each exporter requires its own particular name for the files in this directory, but generally they are looking for the same files defined in the global exporter directory (<install>/developer/exporter_templates) using 'custom.<extension>' (e.g., custom.project.mak)
	`-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.
	`-tools, --config-tools=<tools>`	Specifies the tools to be generated based on the file extension. If this is not provided, then all active tools will generate. You may list more than one tool extension separated by commas.
	`-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=TIMEOUT_SECONDS`	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.
	`-log, --log-only`	If set, all errors, warnings, and information is only printed to the process log files. Nothing is printed to the console.

Operation	Example/Arguments	Description
generate	-cpproj, --copy-proj-sources	Copies all files referenced by the project and links any SDK sources. Can be combined with -cpsdk.
	-cpsdk, --copy-sdk-sources	Copies all files referenced by the selected components and links any project sources. Can be combined with -cpproj.
	-nocp, --no-copy	Causes all files to be linked into the generation location. Only generated files will exist in the generation location. Cannot be used with any other copy operation.
	-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.
	-extmpl, --export-templates=EXPORT_TEMPLATES_FOLDER	Location for custom export templates. Each exporter requires its own particular name for the files in this directory, but generally they are looking for the same files defined in the global exporter directory (<install>/developer/exporter_templates) using 'custom.<extension>' (e.g., custom.project.mak)
	-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.
	-tools, --config-tools=<tools>	Specifies the tools to be generated based on the file extension. If this is not provided, then all active tools will generate. You may list more than one tool extension separated by commas.
	-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=TIMEOUT_SECONDS	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.
	-log, --log-only	If set, all errors, warnings, and information is only printed to the process log files. Nothing is printed to the console.
	-s, --sdk=SDK_PATH	Location of either the sdk folder containing the .slcs file, or the file itself. This is only required if no default sdk is set using 'slc configuration -s SDK_PATH'. If an sdk is supplied here and one is configured as default, this one takes precedence.

Operation	Example/Arguments	Description
generate	`-cpsdk, --copy-sdk-sources`	Copies all files referenced by the selected components and links any project sources. Can be combined with -cpproj.
	`-nocp, --no-copy`	Causes all files to be linked into the generation location. Only generated files will exist in the generation location. Cannot be used with any other copy operation.
	`-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.
	`-extmpl, --export-templates=EXPORT_TEMPLATES_FOLDER`	Location for custom export templates. Each exporter requires its own particular name for the files in this directory, but generally they are looking for the same files defined in the global exporter directory (<install>/developer/exporter_templates) using 'custom.<extension>' (e.g., custom.project.mak)
	`-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.
	`-tools, --config-tools=<tools>`	Specifies the tools to be generated based on the file extension. If this is not provided, then all active tools will generate. You may list more than one tool extension separated by commas.
	`-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=TIMEOUT_SECONDS`	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.
	`-log, --log-only`	If set, all errors, warnings, and information is only printed to the process log files. Nothing is printed to the console.
	`-s, --sdk=SDK_PATH`	Location of either the sdk folder containing the .slcs file, or the file itself. This is only required if no default sdk is set using 'slc configuration -s SDK_PATH'. If an sdk is supplied here and one is configured as default, this one takes precedence.
	`-tpl, --template-output`	Causes all files to be generated as in the template style. This is useful for pre-made examples. This turns on no_copy and therefore cannot be used with copy sources.

Operation	Example/Arguments	Description
generate	`-nocp, --no-copy`	Causes all files to be linked into the generation location. Only generated files will exist in the generation location. Cannot be used with any other copy operation.
	`-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.
	`-extmpl, --export-templates=EXPORT_TEMPLATES_FOLDER`	Location for custom export templates. Each exporter requires its own particular name for the files in this directory, but generally they are looking for the same files defined in the global exporter directory (<install>/developer/exporter_templates) using 'custom.<extension>' (e.g., custom.project.mak)
	`-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.
	`-tools, --config-tools=<tools>`	Specifies the tools to be generated based on the file extension. If this is not provided, then all active tools will generate. You may list more than one tool extension separated by commas.
	`-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=TIMEOUT_SECONDS`	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.
	`-log, --log-only`	If set, all errors, warnings, and information is only printed to the process log files. Nothing is printed to the console.
	`-s, --sdk=SDK_PATH`	Location of either the sdk folder containing the .slcs file, or the file itself. This is only required if no default sdk is set using 'slc configuration -s SDK_PATH'. If an sdk is supplied here and one is configured as default, this one takes precedence.
	`-tpl, --template-output`	Causes all files to be generated as in the template style. This is useful for pre-made examples. This turns on no_copy and therefore cannot be used with copy sources.

Operation	Example/Arguments	Description
generate	`-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.
	`-extmpl, --export-templates=EXPORT_TEMPLATES_FOLDER`	Location for custom export templates. Each exporter requires its own particular name for the files in this directory, but generally they are looking for the same files defined in the global exporter directory (<install>/developer/exporter_templates) using 'custom.<extension>' (e.g., custom.project.mak)
	`-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.
	`-tools, --config-tools=<tools>`	Specifies the tools to be generated based on the file extension. If this is not provided, then all active tools will generate. You may list more than one tool extension separated by commas.
	`-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=TIMEOUT_SECONDS`	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.
	`-log, --log-only`	If set, all errors, warnings, and information is only printed to the process log files. Nothing is printed to the console.
	`-s, --sdk=SDK_PATH`	Location of either the sdk folder containing the .slcs file, or the file itself. This is only required if no default sdk is set using 'slc configuration -s SDK_PATH'. If an sdk is supplied here and one is configured as default, this one takes precedence.
	`-tpl, --template-output`	Causes all files to be generated as in the template style. This is useful for pre-made examples. This turns on no_copy and therefore cannot be used with copy sources.

Operation	Example/Arguments	Description
generate	-extmpl, --export-templates=EXPORT_TEMPLATES_FOLDER	Location for custom export templates. Each exporter requires its own particular name for the files in this directory, but generally they are looking for the same files defined in the global exporter directory (<install>/developer/exporter_templates) using 'custom.<extension>' (e.g., custom.project.mak)
	-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.
	-tools, --config-tools=<tools>	Specifies the tools to be generated based on the file extension. If this is not provided, then all active tools will generate. You may list more than one tool extension separated by commas.
	-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=TIMEOUT_SECONDS	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.
	-log, --log-only	If set, all errors, warnings, and information is only printed to the process log files. Nothing is printed to the console.
	-s, --sdk=SDK_PATH	Location of either the sdk folder containing the .slcs file, or the file itself. This is only required if no default sdk is set using 'slc configuration -s SDK_PATH'. If an sdk is supplied here and one is configured as default, this one takes precedence.
	-tpl, --template-output	Causes all files to be generated as in the template style. This is useful for pre-made examples. This turns on no_copy and therefore cannot be used with copy sources.
	--with=COMPONENT_LIST	Accepts a comma separated list of component IDs to be added to the project. Use a ':' operator for instance-capable components to separate instance names after the component ID (for example, 'pwm:led0:led1' for led0 and led1 instances with pwm). If a component is to come from an extension, a semicolon should be used first to specify the id of the extension such as 'pwm;arc:led0:led1' if with-ing a component called pwm from an extension called 'arc' with instances 'led0' and 'led1'. The version of the extension is not supplied but is inferred from the project state.

Operation	Example/Arguments	Description
generate	`-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.
	`-tools, --config-tools=<tools>`	Specifies the tools to be generated based on the file extension. If this is not provided, then all active tools will generate. You may list more than one tool extension separated by commas.
	`-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=TIMEOUT_SECONDS`	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.
	`-log, --log-only`	If set, all errors, warnings, and information is only printed to the process log files. Nothing is printed to the console.
	`-s, --sdk=SDK_PATH`	Location of either the sdk folder containing the .slcs file, or the file itself. This is only required if no default sdk is set using 'slc configuration -s SDK_PATH'. If an sdk is supplied here and one is configured as default, this one takes precedence.
	`-tpl, --template-output`	Causes all files to be generated as in the template style. This is useful for pre-made examples. This turns on no_copy and therefore cannot be used with copy sources.
	`--with=COMPONENT_LIST`	Accepts a comma separated list of component IDs to be added to the project. Use a ':' operator for instance-capable components to separate instance names after the component ID (for example, 'pwm:led0:led1' for led0 and led1 instances with pwm). If a component is to come from an extension, a semicolon should be used first to specify the id of the extension such as 'pwm;arc:led0:led1' if with-ing a component called pwm from an extension called 'arc' with instances 'led0' and 'led1'. The version of the extension is not supplied but is inferred from the project state.

Operation	Example/Arguments	Description
generate	-tools, --config-tools=<tools>	Specifies the tools to be generated based on the file extension. If this is not provided, then all active tools will generate. You may list more than one tool extension separated by commas.
	-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=TIMEOUT_SECONDS	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.
	-log, --log-only	If set, all errors, warnings, and information is only printed to the process log files. Nothing is printed to the console.
	-s, --sdk=SDK_PATH	Location of either the sdk folder containing the .slcs file, or the file itself. This is only required if no default sdk is set using 'slc configuration -s SDK_PATH'. If an sdk is supplied here and one is configured as default, this one takes precedence.
	-tpl, --template-output	Causes all files to be generated as in the template style. This is useful for pre-made examples. This turns on no_copy and therefore cannot be used with copy sources.
	--with=COMPONENT_LIST	Accepts a comma separated list of component IDs to be added to the project. Use a ':' operator for instance-capable components to separate instance names after the component ID (for example, 'pwm:led0:led1' for led0 and led1 instances with pwm). If a component is to come from an extension, a semicolon should be used first to specify the id of the extension such as 'pwm;arc:led0:led1' if with-ing a component called pwm from an extension called 'arc' with instances 'led0' and 'led1'. The version of the extension is not supplied but is inferred from the project state.
	--without=COMPONENT_LIST	Accepts a comma separated list of component IDs to be removed from the project. You may use a ':' operator with instance-capable components to specify the instance to remove. If no ':' is provided, all instances are removed. This uses ';' semantics as described above for '--with' to remove components belonging to extensions. --without runs after the '--with' command, so if a component appears in both lists it will not be added.

Operation	Example/Arguments	Description
generate	-o, --output-typ e	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=TIMEOUT_SECONDS	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.
	-log, --log-only	If set, all errors, warnings, and information is only printed to the process log files. Nothing is printed to the console.
	-s, --sdk=SDK_PATH	Location of either the sdk folder containing the .slcs file, or the file itself. This is only required if no default sdk is set using 'slc configuration -s SDK_PATH'. If an sdk is supplied here and one is configured as default, this one takes precedence.
	-tpl, --template-output	Causes all files to be generated as in the template style. This is useful for pre-made examples. This turns on no_copy and therefore cannot be used with copy sources.
	--with=COMPONENT_LIST	Accepts a comma separated list of component IDs to be added to the project. Use a ':' operator for instance-capable components to separate instance names after the component ID (for example, 'pwm:led0:led1' for led0 and led1 instances with pwm). If a component is to come from an extension, a semicolon should be used first to specify the id of the extension such as 'pwm;arc:led0:led1' if with-ing a component called pwm from an extension called 'arc' with instances 'led0' and 'led1'. The version of the extension is not supplied but is inferred from the project state.
	--without=COMPONENT_LIST	Accepts a comma separated list of component IDs to be removed from the project. You may use a ':' operator with instance-capable components to specify the instance to remove. If no ‘:’ is provided, all instances are removed. This uses ‘;’ semantics as described above for ‘--with’ to remove components belonging to extensions. --without runs after the ‘--with’ command, so if a component appears in both lists it will not be added.

Operation	Example/Arguments	Description
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, summarize	`summarize --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.

Operation	Example/Arguments	Description

graph	`graph --project blink.slcp`
	`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.
		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.
		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.

	`--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 -t emlib -d project_sdk_extension -a neo_politan`	Clones sdk components for developers who wish to use a component in a slightly modified form. Takes care of ensuring all referenced files and other data is copied so that the cloned components are truly self-contained. To use the resulting clone, place it in a new sdk extension.
	`--destination`	The location that the component will be cloned into. The component's slcc and directly required files will be cloned to this location.
	`--target=TARGET_COMPONENT_ID`	The component to be cloned. Referenced files will be copied along with the .slcc, and the project search paths will be updated and saved. Clone data is automatically added based on the current system's date.
	`--author "Fozzy Bear"`	If the optional author option is specified, the cloned component's author field in the .slcc is updated.

Operation	Example/Arguments	Description
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 if 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.
sbom	`--all-components=<allSdkComponents>[,<allSdkComponents>...]`	Add every component from a comma separated list of input SDK or extension path to the SBOM. This is a utility command that is fully equivalent to `--with=<every_component>`. This cannot be specified with the `-p` command.
	`--cyclone-json`	The CycloneDX JSON output format.
	`--cyclone-xml`	The CycloneDX XML output format.
	`--spdx-default`	The SPDX default (tag:value) output format.
	`--spdx-json`	The SPDX JSON.

Work with SDKs and Extensions

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

SDK Options

These options are shared among all SDK commands.

Option	Example	Description
-s --sdk	`-s '/sdk/sisdk'`	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.

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.

Operation	Example/Arguments	Description
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.).
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. 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. 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.

Create SDK Extensions

This section discusses what constitutes an SDK extension and how to create one, as well as collating any information from the primary specification that pertains to SDK extensions. Note that this document is updated separately from the specification. In the event that the specification and this document do not align, the specification supersedes this document.

What Makes Up an SDK Extension?

To be used as an SDK extension, all you need initially is an `.slce` file and a folder that contains it. This becomes the container for the SDK extension.

Additionally, if you intend to install an SDK extension into an SDK manually (as opposed to letting the Simplicity Studio UI handle it), make sure the folder containing the `.slce` file is in a folder called `extension` inside of the SDK you intend to install it into.

Create the Empty SDK Extension

First, you will create an empty SDK extension for the sole purpose of testing that the integration of the SDK extension into the SDK is successful. So you can test as you go, you will create an SDK extension such that you do not need to go to the Simplicity Studio UI to install it.

Name your base folder anything you like.

2. Ensure your base folder is in the extension directory of the SDK. If the folder does not exist, create it. By default, it is likely you will not have it if you have yet to install extensions into a particular SDK.
    1. Because you will be using the `slc cli` later to verify your SDK extension is installed properly, make sure you are creating your SDK extension in the same SDK that is configured with `slc configuration --sdk .`
    2. If you are performing these steps with an sdk you downloaded using the Simplicity Studio installation, you will typically find the sdk in your user home folder. Navigate to SimplicityStudio → SDKs → simplicity_sdk to find your installed sdk. You can also see where you installed the sdk from Simplicity Studio by checking the installation manager. Go to Launcher View, select Install at the top, and select the SDKs tab. Then, find the Simplicity SDK - 32-bit and Wireless MCUs installation card and check the path indicated there.

    This is an example folder structure:

```
+ - SimplicityStudio
    |                                + - SimplicityStudio
    |
    + - SDKs
      |
      + - simplicity_sdk
        |
        + - extension
          |
          + - your_extension
```

    Note: This is extension singular, not extensions plural.

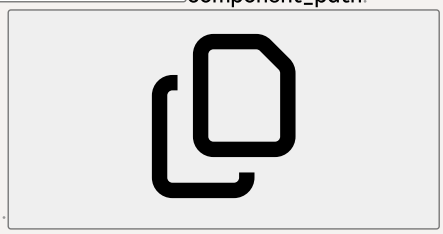
3. Create an `.slce` file at the base folder you wish to use. If starting from scratch completely, your folder may be empty.
4. Substituting <text> where appropriate, enter the following into the `.slce` file.

    .slce minimum working data

```
id: "<your_extension_id>"
label: "                         id: "<your_extension_id>"
label: "<your_extension_label>"
version: <your_extension_version>
sdk:
  id: "simplicity_sdk"
version: "2025.6.1"
component_path:
 - path: <your_component_path>
```

    - <your_extension_id> is a unique id for this SDK extension and how SLC internally recognizes it as distinct from another vendor. The SLC specification for id indicates which characters are allowed. Generally, this means no spaces, all lowercase, and _ (underscore character) are permitted.
    - <your_extension_label> a human-readable name for the SDK extension. This may contain spaces.
    - <your_extension_version> the starting version. This must follow semantic versioning rules – for example, 1.0.0, 0.0.1.
    - <your_component_path> is the folder at the same level as the `.slce` file where the SDK extension can find your `.slcc` files. If you do not know, the `.slcc` file defines an installable component into an SLC project. You may define multiple paths here as well. This procedure assumes the path is . , as in

```
component_path:
 - path: .
```



```
- path: .
```

- The sdk definition is typically `simplicity_sdk` because that is the only SLC aware Silicon Labs sdk at this time. You may need to change the version number if you have a later version of the Simplicity SDK.
  - If you want to learn your `simplicity_sdk` version without opening Simplicity Studio, open the `simplicity_sdk.slcs` file inside the sdk folder and check the `sdk_version` metadata.

5. You now have a valid `.slce` file. If you already have some `.slcc` files and they are targeted by `component_path`, skip to step 6. Otherwise:
   1. Make a new `.slcc` file in the same directory as the `.slce` (or in one of the directories pointed to by `component_path`).
   2. Add a bare minimum amount of text for the `.slcc` to be considered valid.
      For example:

```
id: neopolitan_icecream
label: Neopolitan
```

```
id: neopolitan_icecream
label: Neopolitan Icecream
package: "ext-comp"
description: Neopolitan Icecream
category: Melting|Icecream
```

```
quality: alpha
```

      The above is an absolute minimum set of required keys for an `.slcc` file to be considered a valid component.
6. Ensure that your `slc` command line is installed and available.
7. Ensure that `slc configuration` is set to use the sdk you are installing the SDK extension for.
8. Run `slc signature trust -extpath <path_to_your_extension_sdk>` so it is trusted. Otherwise, none of its contents will be parsed, and will therefore not be found in later steps. This should be the path to the *folder containing the .slce* , not to the `.slce` itself.
   Note: This is different from `slc signature trust -extid <your_extension_id>:<your_extension_version>` which installs trust for your SDK extension based on its `id` and `version` regardless of where it ends up moving. The former method will trust the SDK extension location allowing you to rename or reversion it without losing trust. If you are following this procedure and manually installed the SDK extension in with your Simplicity SDK, you will likely vastly prefer the `-extpath` option.
9. Run `slc signature trust --sdk <path_to_the_simplicity_sdk>` if you have not yet trusted your SDK.
10. Run `slc examine <your_slcc_component_id> -ext <your_extension_id>:<your_extension_version>` where the `id` is either the dummy `id` you created for the test component (in the above example, that would be `neopolitan_icecream`), or the `id` of some component that should exist in your SDK extension. The `-ext` part tells `examine` to look in a specific SDK extension and you must supply both the `id` and `version` so it knows where to look (since the same `id` could be used in different places).
11. The SLC command line will report the component is found and the information about it will be displayed on the console. If it does not appear, recheck the above steps. Otherwise, you have installed your SDK extension.

Note: The most common error here is not trusting the SDK extension. If it is not trusted, you will not receive any notification, it will just not load and not find the component.

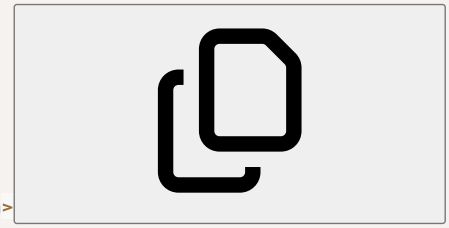
Make a Project Use an SDK Extension (Command Line)

Normally, you can add an SDK extension to a project using the Component Selector in the Simplicity Studio User Experience (UX). Without that, you must do this manually. You add an SDK extension to a project by modifying its `.slcp` file to refer to the SDK extension. Components within that SDK extension are included using a special syntax.

1. Make a new project. This is beyond the scope of this document, but you can take an existing Simplicity Studio project you have or use `slc generate -np` on an example project in the SDK to create a new one to work with if you do not have one already. Refer to the relevant `slc` documentation on how to find and generate an example.
2. Inside the `.slcp` file, add a section as shown below. Add to the same level as other root metadata tags:

```
sdk_extension:
 - id: <your_extension_id>
```

sdk_extension:
- id: <your_extension_id>

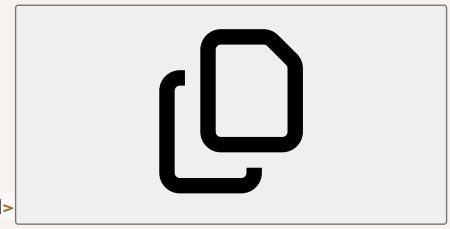


```
version: <your_extension_version>
```

3. This tells the project that it will use that SDK extension with specifically that version. You may define multiple SDK extensions to use in a project, but for a given `id`, you may only choose one version.
4. Now, tell the project to use one of the components in the SDK extension. There is a `component` key at the root of the yaml with a list of components selected in a project. Observe the difference between referring to a component from the SDK and a component from the SDK extension as shown below.

```
component:
 - id: emlib
```

component:
- id: emlib
- id: neopolitan_icecream



```
from: <your_extension_id>
```

   - `emlib` comes from the Simplicity SDK. Any component from the SDK needs only the `id` field. You do not need to have this specific component. This is just an example of what using a component from the main SDK looks like compared to the SDK extension.
   - `neopolitan_icecream`, or whatever `id` of a component in your sdk extension, comes from that SDK extension. As such, you need to tell SLC that this component comes from a different location. Note that the version is not included next to the `from` field. This is why `sdk_extension` field exists earlier. `sdk_extension` indicates what specific SDK extension to pick out. You cannot have multiple SDK extensions with the same `id` even if the versions are different, in the same project. The component listing will always refer to whatever version of that SDK extension is being brought in at the time.
5. You can now use `slc summarise` on the project to view and ensure that your project is seeing the component.

Use Your SDK Extension with the Simplicity Studio IDE

The Simplicity Studio Integrated Development Environment (IDE) provides a User Interface (UI) for adding SDK extensions from anywhere on your system. However, if you are actively developing SDK extensions, it is

important to be aware of a few limitations. This section discusses how to add your custom SDK extension to Simplicity Studio in this way and what to look out for as an SDK extension developer.

First, if you have followed the above procedures up to this point and used a project that is already part of a Simplicity Studio workspace, it is already connected. You need only launch Simplicity Studio and open the project. The SDK extension will already be installed and you can browse components within it.

If you followed the above procedures but did not use a project in a Simplicity Studio workspace, follow these steps

Use an already installed and trusted SDK extension with a project:

1. Create a new project or pick an existing one using the same `sdk` you installed the SDK extension for.
2. Open the `.slcp` file.
3. Navigate to the Component Selector by clicking the Software Components tab.
4. Search for a component in your SDK extension.
5. After you find it, Simplicity Studio prompts you to add the SDK extension to the project before you install the component.
6. Install the component and your project is now connected to that SDK extension.

For the above steps, you do not need to install the SDK extension at a preference level to be globally accessible. By placing it in the Simplicity SDK's `extension` folder, you already did that manually and Simplicity Studio has detected that.

However, if you did not install the SDK extension or you are using a different Simplicity SDK that does not have the SDK extension, your flow will be a bit more involved. If you wish to use your SDK extension with a different SDK or you have distributed it and another SDK wishes to use it, follow these steps:

1. In Simplicity Studio go to Preferences > Simplicity Studio > SDKs and select the Simplicity SDK Suite to which the SDK extension will be added. Click Add Extension....
2. Click Browse and navigate to the root folder of the new SDK extension and click Select Folder.
3. The SDK extension should be displayed in the Detected SDK Extension window with the correct name, version, and path. Click OK and then Trust and Apply and Close.
   Important: Be aware that installing an SDK extension like this will copy the SDK extension from wherever it was into the proper folder structure and rename it so it is detected as an SDK extension. Because this is a *copy*, changes you make to your original SDK extension will not be reflected until you remove and add the SDK extension again. This is why the earlier part of the above procedures recommend starting from the correct installation location of the SDK you want to test with because this shields you from this complication.
4. You can now follow the above steps in Using an already installed and trusted SDK extension with a project.

Daemon Mode

The SLC CLI can be run as a daemon. In this mode, the application is launched and remains open to service further commands. This reduces the time to service any given command, especially those which require loading the sdk (which is most of them). To run a command in daemon mode, add `--daemon` to any command, after the slc but before the command itself. For example, `slc --daemon generate neo_politan.slcp -d gen_destination`. If the daemon is not currently running, it will start. If it is, it will merely connect.

The daemon mode was built to scale with Continuous Integration and Continuous Development (CI/CD) systems that may be running many operations at once. As such, it is enough to just add `--daemon` to any command, and SLC CLI will automatically handle what it needs to ensure that if multiple calls are all attempting to start the daemon, only one will win and service every call.

Daemon Mode Considerations

- The SDK that is loaded when the daemon first launches, as configured by slc configuration, is the assumed default for all future commands until the daemon is shut down.
- When using the daemon for CI/CD flows:
  - It is recommended that every command references the --sdk directly and does not rely on any configuration.

- Run all slc signature trust commands with the --daemon option.

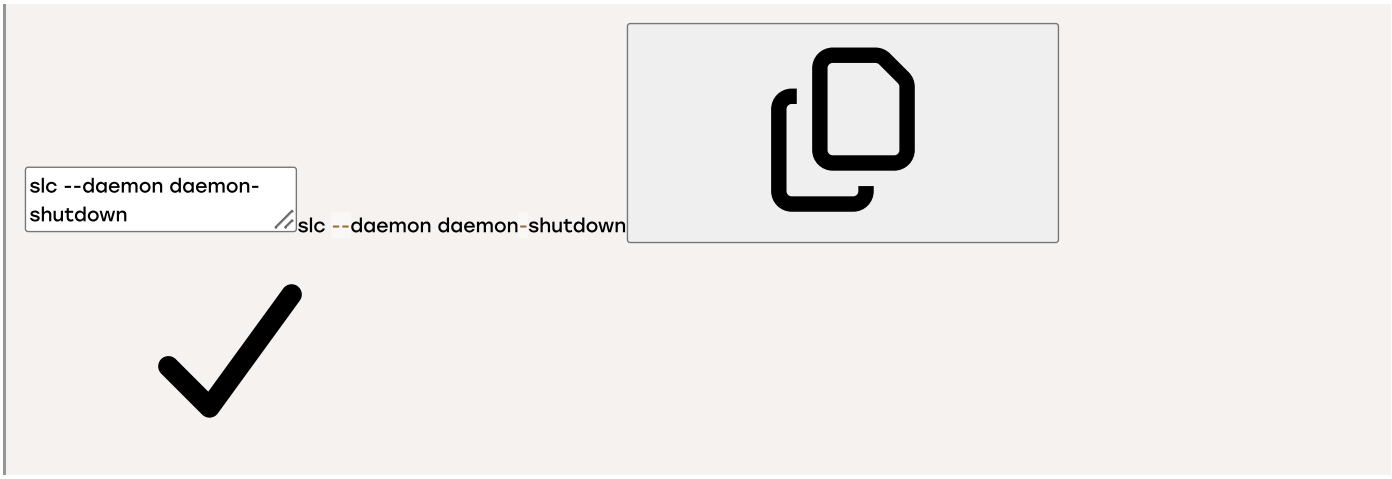
Shut Down

The daemon will automatically shut down in two scenarios (outside of crashes):

- If a bug occurs where a daemon is launched but not connected to, the daemon process will end after 30 seconds.
- After the daemon has serviced at least one command, and it has been more than 30 minutes since the last command. This timeout can be adjusted.

The daemon can be manually shut down by issuing the following command:

```
slc --daemon daemon-shutdown
```



Daemon Options Reference

The following are supported daemon-specific options that apply to the daemon connections themselves. As with `--daemon` they should be used before the slc command.

Option	Example	Description
--do-not-launch-daemon		Will attempt to connect to an existing daemon, but will fail if none is present (in other words, it will not attempt to start up a daemon if one is not available).
--daemon-timeout=TIMEOUT		How long, in minutes, the daemon will stay alive after servicing a request before it automatically shuts down.
--daemon-only		If set, the daemon will not attempt to use a direct call to fallback. Under normal circumstances, if something goes wrong attempting to connect to a daemon, then SLC CLI is launched directly to service whatever command it was given. This fallback may be considered more of a problem than a solution in some cases, such as system testing. If this is set, the launcher will never attempt to do a direct launch of SLC CLI, and fail immediately if anything goes wrong attempting to either start, or connect to, a Daemon.
--reload-sdk		Reload the SDK from the input location from disk. Users should call this if the SDK has changed on disk and SLC is currently running in Daemon mode.

Workarounds

When SLC CLI has issues with daemon mode, it will try to fall back to servicing the command as if daemon mode was not requested. In the event of instability, try the following:

- Check the user directory's .uc/cli folder for data directories (cli ones should be cli_pdata.) Delete all of the data directories. Normally, the launcher should delete .running files that are stale (empty and never getting filled in) or refer to dead process ids, but if a daemon has hung in an unexpected way and simply cannot respond, this is the easiest way to remove it from consideration.
- Check if there are any stray .lock files in any data directories, and delete them if they exist.

Project Generation Examples

Project Generation Examples

Silicon Labs Configurator (SLC) supports specifying all of its arguments in `.slconf` files and this improves portability for project generation. Any `slc` argument can be specified in the `.slconf` file. The long form of the argument name is used without the leading `--`. The format of the `.slconf` file is detailed in SLT Configuration File Specification.

Key benefits of using a `.slconf` file include:

- Shorter command lines
- Project generation portability
- Reproducibility
- Removal of requirements to trust SDKs

> Notes on portability Key points for ensuring a project and its supporting files are portable are that they contain no absolute paths and that the specified paths work across the supported operating systems (Windows, macOS, and Linux (Ubuntu)). For maximum portability, the `.slconf` files should be located such that relative paths can be used to specify the SDK and the project or workspace file (only one is specified). A directory structure that meets those requirements is shown in the examples that follow. For portability across operating systems the relative paths should contain forward slashes (/) only - even on Windows. `slc` supports using forward slashes on Windows from a command prompt or PowerShell terminal.

Steps for Creating a .slconf File

Project File Creation Step

Create a project-level `.slconf` file with the following elements:

- SDK and SDK extension paths: The path to the SDK and any SDK extension is specified using the `sdk-package-path` argument which takes a list of the SDK and extensions in a single statement. Any packages specified in this statement are implicitly trusted by `slc`.
- Target board (or part): It is recommended to use a Silicon Labs development board when creating the initial project so that peripheral components will have valid pin definitions. After initial project creation, the board can be replaced with the desired target part and then the existing pin definitions can be modified for the customer hardware. Specify the board in a `with` statement. (`with = [ brd4403b,]`)
- Output type: Specify the project generator to use for the integrated development environment (IDE) or build system. More than one generator can be specified as a comma-separated list. (`output-type = "vscode"`)
- Project Import mode: This is a boolean argument for if the SDK source files should be copied into the generated project or specified as links in the build system files. (`copy-sources = "TRUE"`)
- Toolchain type: Specify one toolchain that will be used to build the project. Options are `gcc`, `iar`, and `llvm`. (`toolchain = "gcc"`)
- New project indicator: This is a boolean argument for if a new instance of the project should be created or not. This should be `"TRUE"` when creating a project from an SDK example project or workspace file and `"FALSE"` for previously created projects. (`new-project = "TRUE"`)
- Workspace file or slcp project file path: This should be the relative path from the `.slconf` file location to the workspace or slcp project file and it should be specified using forward slashes (`/`). Examples:

```
project-file = "../home/.silabs/slt/installs
```

```
// project-file = "../home/.silabs/slt/installs/conan/p/simpleb526998f4a4d/p/app/bluetooth/example/bt_soc_empty/bt_soc_empty.slcp"
```

```
workspace = "../home/.silabs/slt/installs
```

```
// workspace = "../home/.silabs/slt/installs/conan/p/simpleb526998f4a4d/p/platform/bootloader/sample-apps/workspaces/bootloader-apploader/bootloader-apploader.slcw"
```

- Other software components: Software components not pulled in by the project or workspace file can be specified using a `with` statement. For example, an IO Stream component specified with the `bt_soc_empty` project. (`with = [ "brd4403b","iostream_eusart:vcom"]`)
- Toolchain: A `[toolchain]` section can be included to specify a path to the toolchain used by the project. However, because `slt` is typically used during the CMake build to locate installed toolchains (and this overrides the toolchain in the `.slconf` file), the `[toolchain]` section is of limited use. If the toolchain needs to be overridden with one not installed using `slt`, define the environment variable `ARM_GCC_DIR` with the desired toolchain path.

Generation Step

The generation command line when using a `.slconf` file is simple because the slconf file (`--slt-config`) and the output location (`-d`) are specified. Optionally a generator timeout can be specified for projects that use an adapter pack that takes a long time to generate, such as a Zigbee or Matter project.

Examples:

```
slc generate --slt-config=slconf/MatterLightSwitchOverThreadSolution_1019a_vs.slconf -d=g/MatterLightSwitchOverThreadSolution_1019a_vs -lfewp --generator-timeout=800
```

```
slc generate --slt-config=slconf/bt_soc_empty_4403b_vs.slconf -d=g/bt_soc_empty_4403b_vs
```

Build Step

The build step consists of invoking CMake from the project `cmake_gcc` folder or the workspace `_cmake` folder. Examples are given in the next section.

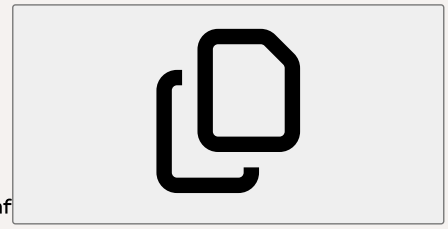
Examples

For these examples, the contents of the user `.silabs` folder created from installing the recommended packages with `slt` was copied to a `home` folder outside of the Users folder. This was done for ease of

demonstration, so that the `USERNAME` folder is not involved in the examples. If this is done any packages updated with `slt` would have to be copied again. The folder structure used for these examples is ("g" is the destination folder used by the generate command):

```
- C:\slc-test\
 - g
```

- C:\slc-test\
  - g
  - home
    - .silabs
      - sdm
      - slt
  - slconf
    - bootloader-apploader_2902a_vs.slconf
    - bootloader-apploader-workspace_2901a_vs.slconf
    - bt_soc_blinky_4402a_vs.slconf
    - bt_soc_empty_4403b_vs.slconf
    - MatterLightSwitchOverThreadSolution_1019a_vs.slconf



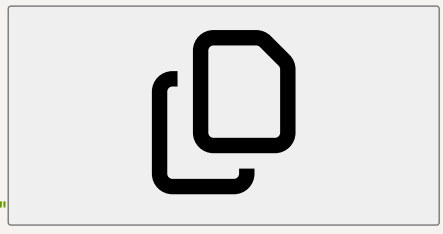
Example 1: Bluetooth Empty Project (bt_soc_empty_4403b_vs.slconf)

.slconf Contents

```
[toolchain]
```

[toolchain]

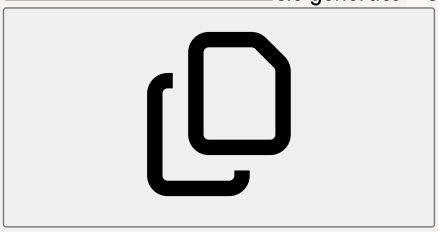
```
[slc]
sdk-package-path = [ "../home/.silabs/slt/installs/conan/p/simpleb526998f4a4d/p",]
with = [ "brd4403b","iostream_eusart:vcom"]
output-type = "vscode"
project-file = "../home/.silabs/slt/installs/conan/p/simpleb526998f4a4d/p/app/bluetooth/example/bt_soc_empty/bt_soc_empty.slcp"
copy-sources = "TRUE"
new-project = "TRUE"
```



```
toolchain = "gcc"
```

Generation Command Line

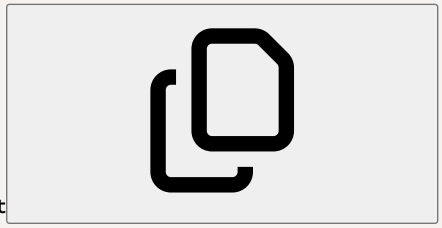
```
slc generate --slt-config=slconf/bt_soc_empty_4403b_vs.slconf -d=g/bt_soc_empty_4403b_vs
```



Build Command Line

Run the following from the `g/bt_soc_empty_4403b_vs/cmake_gcc` folder:

```
cmake --workflow --preset project
```

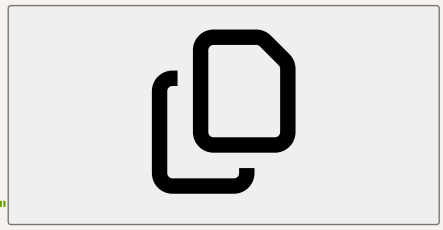


Example 2: Bluetooth Blinky Project (bt_soc_blinky_4402a_vs.slconf)

.slconf Contents

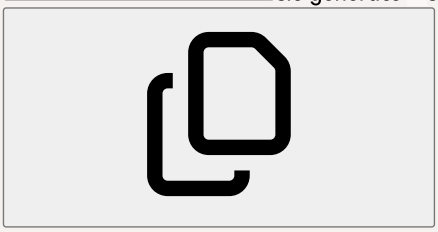
```
[toolchain]

[slc]
sdk-package-path = [ "../home/.silabs/slt/installs/conan/p/simpleb526998f4a4d/p",]
with = [ "brd4402a",]
output-type = "vscode"
project-file = "../home/.silabs/slt/installs/conan/p/simpleb526998f4a4d/p/app/bluetooth/example/bt_soc_blinky/bt_soc_blinky.slcp"
copy-sources = "TRUE"
new-project = "TRUE"
toolchain = "gcc"
```



Generation Command Line

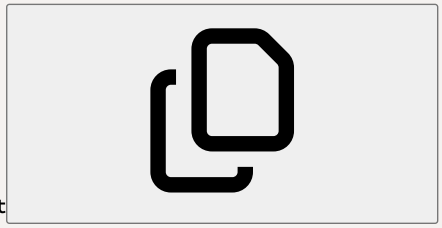
```
slc generate --slt-config=slconf/bt_soc_blin
```
slc generate --slt-config=slconf/bt_soc_blinky_4402a_vs.slconf -d=g/bt_soc_blinky_4402a_vs



Build Command Line

Run the following command from the `g/bt_soc_blinky_4402a_vs/cmake_gcc` folder:

```
cmake --workflow --preset project
```
cmake --workflow --preset project

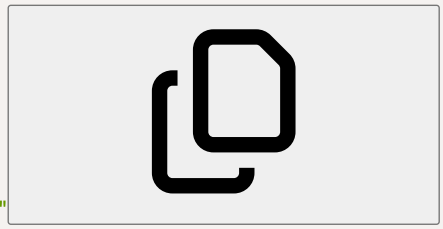


Example 3. Bootloader Apploader Project (bootloader-apploader_2902a_vs.slconf)

.slconf Contents

```
[toolchain]
```
[toolchain]

```
[slc]
sdk-package-path = [ "../home/.silabs/slt/installs/conan/p/simpleb526998f4a4d/p",]
with = [ "brd2902a",]
output-type = "vscode"
project-file = "../home/.silabs/slt/installs/conan/p/simpleb526998f4a4d/p/platform/bootloader/sample-apps/bootloader-apploader/bootloader-apploader.slcp"
copy-sources = "TRUE"
new-project = "TRUE"
toolchain = "gcc"
```



Generation Command Line

```
slc generate --slt-config=slconf/bootloader-apploader_2902a_vs.slconf -d=g/bootloader-apploader_2902a_vs
```

Build Command Line

Run the following command from the `g/bootloader-apploader_2902a_vs/cmake_gcc` folder:

```
cmake --workflow --preset project
```

Example 4. Bootloader Workspace (bootloader-apploader-workspace_2901a_vs.slconf)

.slconf Contents

```
[toolchain]

[slc]
sdk-package-path = [ "../home/.silabs/slt/installs/conan/p/simpleb526998f4a4d/p",]
with = [ "brd2901a",]
output-type = "vscode"
workspace = "../home/.silabs/slt/installs/conan/p/simpleb526998f4a4d/p/platform/bootloader/sample-apps/workspaces/bootloader-apploader/bootloader-apploader.slcw"
copy-sources = "TRUE"
new-project = "TRUE"
toolchain = "gcc"
```

Generation Command Line

```
slc generate --slt-config=slconf/bootloader-apploader-workspace_2901a_vs.slconf -d=g/bootloader-apploader-workspace_2901a_vs
```

Build Command Line

Run the following command from the `g\bootloader-apploader-workspace_2901a_vs\bootloader-apploader-workspace_cmake` folder:

```
cmake --workflow --preset project
```

Example 5. Matter Workspace (MatterLightSwitchOverThreadSolution_1019a_vs.slconf)

This example is using Matter SDK Extension 2.7.0 and to get the project to build the environment variable `ARM_GCC_DIR` was defined to point to an installation of the GCC ARM 14.2.1 toolchain.

.slconf Contents

```
[toolchain]

[slc]
sdk-package-path = [ "../home/.silabs/slt/installs/conan/p/matte7710ee5c0d1f2/p", "../home/.silabs/slt/installs/conan/p/simpleb526998f4a4d/p", "../home/.silabs/slt/installs/conan/p/wisece6a05cd369ee2/p",]
with = [ "brd1019a",]
output-type = "vscode"
workspace = "../home/.silabs/slt/installs/conan/p/matte7710ee5c0d1f2/p/slc/solutions/light-switch-app/series-3/light-switch-app-thread-bootloader.slcw"
copy-sources = "TRUE"
new-project = "TRUE"

toolchain = "gcc"
```

Generation Command Line

```
slc generate --slt-config=slconf/MatterLight
```

```
slc generate --slt-config=slconf/MatterLightSwitchOverThreadSolution_1019a_vs slconf -d=g/MatterLightSwitchOverThreadSolution_1019a_vs -lfewp --generator-timeout=800
```

Build Command Line

Run the following command from the `g\MatterLightSwitchOverThreadSolution_1019a_vs\light-switch-app-thread-bootloader_cmake` folder:

```
cmake --workflow --preset project
```

Project Migration

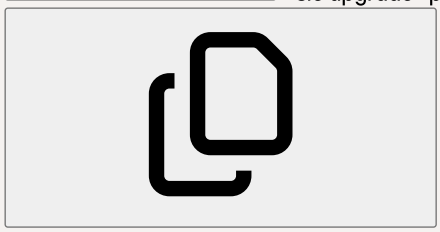
Project Migration

Use the Upgrade command in SLC CLI to migrate a project based on Simplicity SDK 2025.6 or earlier to 2025.12 or later. The upgrade process updates the project to align with changes to the new folder structure in the SDK 2025.12+. Follow these steps to complete the migration.

> Note: Running both commands performs an in-place project migration and upgrade.

1. Upgrade the project to the Simplicity SDK 2025.12.

```
> slc upgrade -p project.slcp --sdk-package-path <sisdk_2025.12_location>,<extension_location>
```



2. Regenerate the project files after the upgrade.

```
> slc generate -p project.slcp --sdk-package-path <sisdk_2025.12_location>,<extension_location>
```