

1. [Quick start](#)
2. [The long theory](#)
 1. [Configuration objects](#)
 2. [Inheriting](#)
 3. [Inheritance and default fields values](#)
 4. [Variables](#)
 5. [What was done in quick start](#)
3. [Fields](#)
 1. [Library](#)
 2. [Toolchain](#)
 3. [Build environment](#)
4. [Adding other component libraries to soclib-cc search path](#)

Quick start

SoCLib's configuration file is used by [soclib-cc](#) to find your tools paths. You may override:

- libraries: SystemC implementation to use (its paths, ...), tlm, ...
- toolchain: Compiler and compiler flags
- build env: toolchain, libraries and other flags (where objects reside, ...)

Let's suppose we want to override SystemC's path, we can write the following `~/soclib/global.conf`:

```
config.libsystemc_22 = Library(
    parent = config.systemc,
    dir = "/home/me/tools/systemc/2.2"
)

config.foo = BuildEnv(
    parent = config.build_env,
    libraries = [config.libsystemc_22],
)

config.default = config.foo
```

Now let's suppose we would like to add another configuration where we use SystemCass. We don't want compiled objects to mix-up, so we'll set another repository for built files.

```
config.libsystemcass = Library(
    parent = config.systemc,
    dir = "/home/me/tools/systemc/cass",
    libs = config.systemc.libs + ["-Wl,-rpath,%(libdir)s", "-ldl", "-fopenmp"],
)

config.systemcass = BuildEnv(
    parent = config.default,
    repos = "repos/systemcass_objs",
    libraries = [config.libsystemcass],
)
```

Now if we want to compile a platform with SystemCass, the only thing to do is to tell it to `soclib-cc`:

```
$ soclib-cc -t systemcass
```

The argument after `-t` is the configuration name, attribute set to `config` in this line:

```
config.systemcass = BuildEnv( ....
```

The only configuration names that can be passed to `-t` are the ones associated to `BuildEnvs`.

The long theory

Configuration objects

SoCLib's configuration file is using inheritance in order to be able to share parameters among different similar instances.

There are 3 base configurations objects to define:

- `Toolchain` to define a compiler suite
- `Library` to define a library, like a SystemC implementation
- `BuildEnv` to define a build environment. This one must reference one instance of each of the above.

You must define a set of these classes. There is a default configuration. It is composed of 3 default configuration classes:

- `config.systemc`.
 - ◆ It is a `Library`.
 - ◆ It expects the environment variable `$SYSTEMC` to point to your actual SystemC installation directory
- `config.toolchain`.
 - ◆ It is a `Toolchain`.
 - ◆ It uses the default compiler (`gcc` & `g++`)
- `config.build_env`.
 - ◆ It is a `BuildEnv`.
 - ◆ It uses the two previous ones.

Inheriting

Inheritance is written using `parent =` as follows:

```
config.my_new_toolchain = Toolchain(  
    parent = config.toolchain,  
    cflags = ....  
)
```

`config` is a global object defined by configuration system. It holds current configuration status.

Inheritance and default fields values

Using a `parent` is optional. If you use `parent =`, all the parent's fields are used as default values for the newly created configuration. Thus you only have to override custom fields.

You can also use no `parent`, then all fields are needed, see below for the list.

Variables

`soclib-cc`'s `-t arg` option will change used configuration. It will make configuration system look for `config.arg`. You should have defined it before.

What was done in quick start

```
# Defining a SystemC implementation inheriting everything
# from default SystemC declaration
config.libsystemc_22 = Library(
    parent = config.systemc,
    dir = "/home/me/tools/systemc/2.2"
)

# Then defining a new default configuration,
# inheriting from default build environment
config.sc22 = BuildEnv(
    parent = config.build_env,
    libraries = [config.libsystemc_22],
)
config.default = config.sc22

# Now with SystemCASS

# Declare a new SystemC implementation
config.libsystemccass = Library(
    parent = config.systemc,
    dir = "/home/me/tools/systemc/cass",
)

config.scass = BuildEnv(
    parent = config.default,

    # This defines a new path to store compiled objects to
    # See 'fields' section below
    repos = "repos/systemccass_objs",

    # and here we tell this configuration use the SystemC implementation
    # declared above.
    libraries = [config.libsystemccass],
)
```

Fields

You may put "%(name)s" anywhere in strings used for expansion, this will expand to value of name attribute in the same class. See systemc definition below.

cflags, <mode>_cflags, libs, <mode>_libs are all optional.

Library

name

Name of the library (what it implements), "systemc" is currently the only specified value.

vendor

Provider of the library, used for some quirks in soclib-cc. "modelsim" and "OSCI" are currently the only specified values.

libs

Link flags. default: ['-L%(libdir)s', '-lsystemc']

cflags

Cflags. default: ['-I%(dir)s/include']

release_cflags

cflags used for a "release" build, ie everyday build.

release_libs

Fields

linking arguments for a "release" build.

debug_cflags

cflags used for a "debug" build, ie when there is a bug to nail down.

debug_libs

linking arguments for a "debug" build.

prof_cflags

cflags used for a "profiling" build, ie performance test build.

prof_libs

linking arguments for a "profiling" build.

Default config.systemc? example:

```
config.systemc = Library(  
    name = 'systemc',  
    vendor = 'OSCI',  
    libs = ['-L%(libdir)s', '-lsystemc', '-lpthread'],  
    cflags = ['-I%(dir)s/include'],  
  
    # libs and cflags are implemented a generic way, now we  
    # have to provide "libdir" and "dir"  
  
    libdir = '%(dir)s/lib-%(os)s',  
    dir = "${SYSTEMC}",  
  
    # Again, libdir uses 'dir' and 'os', thus we have to define "os"  
    # here we use the value provided by a function.  
    os = _platform(),  
)
```

Toolchain

tool_<KEY>

executable name and/or full path for a given tool.

Used <KEY>s are: CXX, CC, CXX_LINKER, CC_LINKER, LD, VHDL, VERILOG.

prefix

a string prepended to all toolchain tools.

cflags

global cflags.

libs

global linking arguments.

release_cflags

cflags used for a "release" build, ie everyday build.

release_libs

linking arguments for a "release" build.

debug_cflags

cflags used for a "debug" build, ie when there is a bug to nail down.

debug_libs

linking arguments for a "debug" build.

prof_cflags

cflags used for a "profiling" build, ie performance test build.

prof_libs

linking arguments for a "profiling" build.

max_processes

Maximum simultaneous compilation processes run (same as -j command-line flag)

- Cflags used for compilation will be `cflags + mode_cflags`

- Libs used for compilation will be `libs + mode_libs`
- *mode* is selected in current build environment, or on command line (flag `-m`)

Example:

```
config.toolchain_64 = Config(
    base = config.toolchain,
    tool_CC = 'ccache gcc-4.2',
    tool_CXX = 'ccache g++-4.2',
    tool_CC_LINKER = 'gcc-4.2',
    tool_CXX_LINKER = 'g++-4.2',
    tool_LD = 'ld'
    max_processes = 3,
    cflags = config.toolchain.cflags + ['-Os', '-m64'],
    libs = config.toolchain.libs + [
        '-liberty',
        '-L/sw/lib', '-lz', '-m64'
    ],
)
```

Build environment

This is the one you may specify from command line with `-t`. By default, this is the configuration set to `config.default` last. The following fields must be set.

`toolchain`

A Toolchain to use for compilation

`libraries`

A list (enclosed in []) of `Library` to be used for compilation and linkage of programs.

`repos`

Path where object files are stored, it may be absolute or relative to current path (where `soclib-cc` is run)

`max_name_length`

Maximum file name length for the file system `repos` is located in. If object file has a longer name, it is hashed to get a shorter one, around 12 chars.

`cache_file`

A path to a file where to store metadata file indexation cache. Default is `"%(repos)s/soclib_cc.cache"`, i.e. a path relative to `repos`.

Adding other component libraries to soclib-cc search path

Soclib-cc searches metadata files in soclib's module directories. This default behavior can be tweaked to add other paths on search list. Simply call `addDescPath`:

```
config.addDescPath("/path/to/my/components")
```

This method may be called more than once to add more directories.