

S C B I

A Build Driver v6.7

User's Guide

Pascal Obry

June 7, 2023

Contents

1	Introduction	4
2	Using	5
2.1	Module reference	5
2.2	Environment	7
2.2.1	Description	7
2.2.2	Environment files	9
2.3	Build plans	10
2.4	Standalone source archive	13
2.5	Command line	14
2.5.1	Description	14
2.5.2	Options	14
3	Creating new plug-ins	19
3.1	Hooks	19
3.1.1	Description	19
3.1.2	Calling order	26
3.2	Meta-module	28
3.3	Setting environment	30
3.4	Setting a proxy	31
3.5	Feature	32
3.6	Build kind	32
3.7	Checking installed tools	33
3.8	Inheritance	34
3.9	Native variant	34
3.10	Calling variant hooks	36
3.11	Getting configuration options	36
3.11.1	Configuration options with spaces	37
3.12	OS dependencies	38
3.13	SCBI Private Store	39

4	Support tools	40
4.1	Completion support	40
4.2	Linters	40
5	Use Cases	41
5.1	User's checkout	41
5.2	Analyzing build failures	42
5.2.1	Replay one specific step	43
5.2.2	Look at the sandbox status	43
6	The Sandbox	45
7	Complete example	46

1 Introduction

SCBI stands for Setup Config Build Install.

The goal is to automate the build procedures from sources. The main features are:

- Allow recursive build of dependencies
- Building from a VCS (currently supported *Git*, *Subversion* and *Mercurial*)
- Supports build variants
- Supports for cross compilation
- Building a specific branch or tag (Git only)
- Building from a tarball
- Building from a user's checkout for easy debug with controlled environment
- Support for build plans (modules with specific variant and/or version)

'toto'

The driver and plug-in modules are all written in `bash` scripts.

For developer of new plug-in modules in SCBI, all setup, source copy and installation are transparent actions, developers just need to fill functions to configure, build and install (generally just calling `configure` from `autotool` or `cmake` and `make`).

Those functions are very close to what would have been typed when building manually from the command line. The goal is to reduce as much as possible the gap between a standard manual build and the commands used on plug-in modules.

When properly setup, building any software should be as easy as:

```
$ scbi --plan=ver mysoft
```

We'll see how to achieve that in this document.

2 Using

2.1 Module reference

A module is described in an SCBI plug-in script (one plug-in in one file). It can be referenced in different places (in other plug-ins, build plan, etc...) with the following format which is called a *module reference*:

```
[for_module=]name[/variant][:version]
```

for_module

The name of the module depending on this module reference. This is optional and can be used to select a specific variant or version when the module name is used as a dependency on `for_module`. When specified on a build plan it is possible to use an alternative syntax, see `@for` see section 2.3 p.11.

name

The name of the module. This is the main reference for the module. The corresponding module script must be named identically and located in `$SCBI_DEFS`.

variant

This is a name to identify a build variant for the module. For example an application can be built with different configuration and or dependencies.

<ID>

The variant name. This will trigger specific hook routines from the module scripts.

native

A variant name that can be defined to use the distribution package. This name `native` is mandatory in this case and required when using the `auto` mode (see below). When a `native` variant is referenced the `build-depends` hook is not used (see section 3.9 p.34).

auto

A specific variant name which checks the dependencies of the defined `native` variant and use it if all dependencies are available (not necessary installed). The `auto` variant should not be defined in a plug-in script but only used in references. The `native` and `auto` variant can be used together to build a package or use the `native` one depending on the availability of the dependencies on the running OS. If used together with other variants, the `auto` variant must be the first one, that is just after the slash. For example:

```
module/auto.fast:v1.3
```

Finally, the `auto` variant can only be defined in a build plan or on the command line.

version

The version to build. This version string can be either:

<ID>

A reference to a VCS version (branch or a Git SHA-1 for example), or a tar version when starting with #.

dev

To take the source from a user's checkout.

skip

To specify that the module must be ignored.

force

To enable an otherwise skip module (possibly set to skip in a build plan).

The version can be set to `force` on the command line to tell that the module should not be skipped. On the command line it is always possible to force a build of a module to a specific variant or version. But if one need to build the default variant or version it is required to pass the `force` as version.

That is, the following command will never overwrite a build plan as it is a default reference to a module:

```
$ scbi -plan=xyz mod
```

But this will do:

```
$ scbi -plan=xyz mod:force
```

The following are examples of module references:

lcms2

Build `lcms2` from repository main (master/trunk) branch.

lcms2/fast

Build `lcms2` *fast* variant from repository main (master/trunk) branch.

gimp=lcms2/slow

When a direct dependency of project `gimp`, build `lcms2` *slow* variant from repository main (master/trunk) branch.

lcms2:v2.0

Build `lcms2` from repository using branch (or tag) *2.0*.

lcms2/fast:v2.0

Build `lcms2` *fast* variable from repository using branch (or tag) *2.0*.

lcms2:bfaf7b82ed

Build `lcms2` from repository using sources from SHA-1 *bfaf7b82ed* (Git only).

lcms2:#v1.0

Build `lcms2` using sources from tar version *1.0*.

lcms2:dev

Build `lcms2` using sources from user's repository checkout found in `$SCBI_GIT_REPO` (assuming `lcms2` is using Git). For a discussion about user's check-out see section 5.1 p.41.

lcms2:skip

Skip `lcms2` from the build. This is especially useful when set on a build plan or the command line to specify that the module is to be ignored.

2.2 Environment

An environment is a set of shell variables used to control the build environment. SCBI comes with defaults that can be used as-is, but one can fine tune the build environment using the following variables.

2.2.1 Description

SCBI_DEFAULT_ENV

The scbi default environment file to use.

SCBI_ENV_DIR

The directory from where the environment file has been loaded.

SCBI_VERSION

The scbi driver version string.
This version is v6.7.

SCBI_PLUGINS

The directories where the SCBI plug-ins are located. Default is `'$HOME/config/scbi'`.

SCBI_PLAN

The name of the build-plan to use if any.

SCBI_ROOT

The default root directory to use by SCBI.
Default is `'$HOME/dev'`.

SCBI_BDIR

The default directory for the build directory. This is also called the sandbox directory.

Default is '\$SCBI_ROOT/builds'.

SCBI_MODULES_DIR[<module-name>[@src|@build]]

An associative array giving the *install*, *source* or *build* directories for a given module name and variant. It is possible to get the *source* and *build* directories by adding respectively @src and @build at the end of the <module-name>. For example SCBI_MODULES_DIR[<module-name>@src] points to the source directory and SCBI_MODULES_DIR[<module-name>@build] point to the build directory. This associative array can be used in hooks config, build and install.

SCBI_PREFIX

The default global install directory. Each module after being installed into the sandbox are copied into this global installation directory.

Default is '\$SCBI_BDIR/install'. Each module can disable global installation (see section 3.1.1 p.26).

SCBI_GIT_REPO

The directories where the user's Git checkouts are to be found.

Default is '\$SCBI_ROOT/repositories/git'.

SCBI_SVN_REPO

The directories where the user's Subversion checkouts are to be found.

Default is '\$SCBI_ROOT/repositories/svn'.

SCBI_HG_REPO

The directories where the user's Mercurial checkouts are to be found.

Default is '\$SCBI_ROOT/repositories/hg'.

SCBI_TAR

The directories where the tarball are located.

Default is '\$SCBI_ROOT/.tar'.

SCBI_LOGS

The directories where the logs are located.

Default is '\$SCBI_ROOT/.logs'.

SCBI_PATCH

The directories where the patches are located.

Default is '\$SCBI_ROOT/.patches'.

SCBI_HOST

The host triplet identifying the running environment.

SCBI_TARGET

The target triplet for the builds.

Default is \$SCBI_HOST.

SCBI_JOBS

The number of simultaneous jobs to use to build. This variable is to be used in module scripts.

Default set to actual number of cores. This can be overwritten using `-jobs` option.

SCBI_HALF_JOBS

Half the simultaneous jobs to use to build. This variable is to be used in module scripts.

Default set to actual number of cores divided by 2. This can be overwritten using `--jobs` option.

2.2.2 Environment files

The environment can be set in the following files searched in this exact order:

\$PWD/.scbi-env

The local environment found in the current working directory.

\$PWD/../.scbi-env

The local environment found in the parent of the working directory. This environment can then be shared for a set of repositories.

\$HOME/.scbi-env

The local environment found in the HOME. This environment can then be shared by all builds.

\$SCBI_DEFS/.env

The default global environment used for all builds.

Also note that instead of the default `'.env'` (or `'.scbi-env'` for local environment files) a specific environment file can be specified with the `--env=<name>` command line option. (see section 2.5.2 p.14).

Example, to use environment `'.env-next'` to build `lcms2`:

```
$ scbi --env=next lcms2
```

Or using an environment variable:

```
$ export SCBI_DEFAULT_ENV=next
$ scbi lcms2
```

2.3 Build plans

A build-plan is a file listing for each module a specific version to build. It is equivalent to list all versions on the command line:

```
$ scbi lcms2:v2 aws:v1
```

But it is preferable to use a build plan to ensure a reproducible build and have all information properly recorded into a version control for example.

A build-plan is a file named `'.plan-<name>'` and listing the versions (see section 2.1 p.5 about module references) for each software:

File `'.plan-next'`:

```
| lcms2:v2
| aws:v1
```

A version specified for a meta-module is propagated to all aggregated modules and recursively to all aggregated meta-modules. Note that this specification is *soft*, meaning that it won't override any specific version specified for a module anywhere in the build plan or on the command line.

Some specific actions are supported with the following syntax:

To start a comment line.

@load <name>

To load another build-plan as a base for the new plan. Every descriptions after this will overwrite the loaded build plan.

This is a simple way to derive a build plan from another one.

File `'.plan-next'`:

```
| # the next line will load build plan standard
| @load standard
| lcms2:v2
| aws:v1
```

@on <discriminants> use <module-reference>

To set a module reference only if the running OS is matching the discriminants. The discriminants are separated with coma and can be the OS name (3 letters OS acronym supported), the OS version, the full OS name (debian, fedora, etc.) and the architecture. Use option --stat to see all supported OS and the list of discriminants for the running machine. If all discriminant are matching the module reference will overwrite previous definitions.

```
# Skip lcms2 on Debian-9
@on deb,9 use lcms2:skip
# Use v2 of lcms2 on Fedora
@on fed use lcms2:v2
```

@for <plug-in> use <modref>

To specify a specific variant and/or version of a plug-in when used as dependencies.

```
@for c-darktable use c-exiv2/auto:v0.27.5
```

@alias <plug-in1> use <plug-in2>

To set an alias for modules in dependencies. If <plug-in1> is not found while looking at dependencies then <plug-in2> will be tried. This may be handy to propose a public alternative to some private plug-in.

```
@alias private-plugin use alternative-plugin
```

[<barrier> ...] /[+-]variant:version

To create a group. The barrier if present must evaluate to true or the group is ignored.

A group can be used in a build plan to group together all modules with the same version and/or variant. To define a group:

[<barrier>

Start with a square bracket and an optional barrier. The barrier is a condition starting with = or /= and the name of a variant or discriminant which will be searched on the specified module's plan variants or a set of OS discriminants.

```
--plan=name/variant1.variant2
```

The barrier can have multiple variants or discriminants separated with a coma:

```
= deb,9
```

All of them must be set for the barrier to be true.

Finally the barrier can have two groups of values, for a = condition only one of the group can match (OR) for the barrier to evaluate to TRUE. For a

`/=` condition both groups must match (AND) for the barrier to evaluate to TRUE.

```
= deb,9 deb,11
```

Will be true on GNU/Debian 9 and 11.

```
/= deb,9 deb,11
```

Will be true on all versions of GNU/Debian except version 9 and 11.

The group will be evaluated only if the barrier evaluate to true otherwise it will be skipped. If the barrier is not specified the group will always be evaluated (i.e. the barrier is by default TRUE).

Modules listed in the group without variant and version will inherit the current variant and version (if any).

] /`[+-]`variant:version

Ends with `/[+-]variant:version` which is the variant and version to apply to all modules in the group. Both `variant` and `version` are optional. If the variant starts with a plus (+) it is added to all modules in the group instead of replacing it. If the variant starts with a minus (-) it is removed from all modules in the group instead of replacing it.

Without a barrier, will always be open:

```
lcms2:v2
[
  moduleA
  moduleB
] /speed:v3
```

With a barrier based on a variant (will be open if the build plan is for example `--plan=name/variant1.variant2`):

```
lcms2:v2
[ = variant1
  moduleA
  moduleB
] /speed:v3
```

With a barrier based on discriminants (will be open on Debian 10):

```
lcms2:v2
[ = deb,10
  moduleA
  moduleB
] /speed:v3
```

And finally, to build one just need to pass the build plan as option:

```
$ scbi --plan=next lcms2 aws
```

Note that if a module is set as `skip` on the build plan, to build the default version one need to force it on the command line:

```
$ scbi --plan=next lcms2:force aws
```

A set of variants separated with a dot can be specified with the build plan. In the following example we set the `mpi` variant of the build plan `next`. This will activate all groups having the following barrier `"= mpi"`.

```
$ scbi --plan=next/mpi lcms2:force aws
```

It is possible to list the final build plan (that is all modules with their corresponding version) using `--list-plan` option. See 2.5.2.

A build-plan can be set in the following files searched in this exact order:

\$PWD/.scbi-plan-<name>

A local build plan found in the current working directory.

\$SCBI_DEFS/.plan-<name>

A global build-plan.

2.4 Standalone source archive

Using the `--standalone` option it is possible to build a fully self contained source archive. This archive embed all necessary sources and scripts to rebuild the software without the need to get access to repositories or source tar files.

On the root of the archive a build script is provided, so starting a build from such standalone source archive boil done to:

Uncompress

```
$ tar xf sa-archive.tgz
```

Enter the directory

```
$ cd sa-archive
```

Start the build

```
$ ./build
```

By default the archive is created into `/tmp`, but it is possible to change the location setting the `SCBI_TMP` environment variable. This may be needed as such archive can be very large.

The file `build-deps` is generated into the root directory and contains the `apt` command with all external (OS) dependencies needed to build the software.

2.5 Command line

2.5.1 Description

The SCBI command line:

```
$ scbi [options] module-ref1 [module-ref2 ... module-refn]
```

For a description of the module reference, see section 2.1 p.5.

It is possible to list the module currently supported by running `scbi` with `--help` or `--stat` option or with `[TAB]` key if completion support is activated (see section 4.1 p.40).

2.5.2 Options

The following options are supported:

-h | --help

List options and usage information.

-v | --version

Display version information of the SCBI driver and versions from plug-ins distributions coming from other sources¹.

--prefix=<dir>

The installation directory to use instead of the default '\$SCBI_PREFIX'.

--build-dir=<dir>

The root build directory to use instead of the default '\$SCBI_BDIR'.

-e | --env=<name>

Use environment file `.env-<name>` instead of the default `.env`. (see section 2.2 p.7)

--plan=<name>[/variant]

Use build plan file `.plan-<name>`. (see section 2.3 p.10) with an optional variant.

--enable-<name>

Enable feature *name*, (see section 3.5 p.32).

-t | --target=<name>

Specify the target to build for, the default is same as the host. This is needed for cross-compilation only.

¹Note that for this to be supported the plug-ins distribution must have generated and installed a file name `'.scbi*version.txt'` next to the plug-ins.

- S | --no-setup**
Skip the setup step.
- W | --no-wrapup**
Skip the wrapup step.
- I | --no-install**
Skip the install step.
- u | --update**
Update the repositories before building.
- s | --setup**
Do the setup step (default)
- c | --config**
Do the config step (default)
- b | --build**
Do the build step (default)
- i | --install**
Do the install step (default)
- clean-install**
Clean installation directory
- w | --wrapup**
Do the wrapup step (default).
- tar**
Check for tar only if explicitly specified in the module reference.
- tests[:only] [=OPTIONS]**
Do a standard build cycle and run the testsuite if any. If suboption `only` is specified only run the testsuite (no build is checked/done). The optional `OPTIONS` parameters is passed as 4th parameter of the tests hook (see section 3.1.1 p.26)
- d | --deps[:warning]**
Check all dependencies and build them if necessary. Do not fail when a missing external dependencies is found when sub-option `warning` is set.
- a | --archive**
Build binary archive when build is successful.
- standalone**
Create a standalone source archive

- l | --log: [yes|no|on-failure]**
Control the log level to display. The default is `on-failure` which displays the command log only in case of failure.
- f | --force**
Do not check if build id is matching, force a rebuild.
- p | --purge[:only]**
Remove module build directory, then if `only` is not specified build the module. The simple `-p` option corresponds on `-purge`.
- shell: [dev|sandbox|sb|tests|use]**
Start a shell (`$SHELL`) with the environment variables set. This is meant to be used to build or test in the very same environment used by SCBI while building the given module on the command line.
 - sandbox**
Specifying `sandbox` or `sb` move the shell working directory into the module's build directory `sandbox`. This mode is meant to debug (or test some other options) the build of the given module. Not only the environment is setup from all dependencies but also the build environment of the specified module is setup. That is, in this mode the environment is exactly the one SCBI sees when starting the configuration, build or install steps.
 - tests**
As above but setup also the tests environment (that is all environment of the tests dependencies).
 - dev**
As `sandbox`, but also move the shell working directory into the module user's checkout. Alternatively one can specify the module's version `dev` instead.
 - use|no sub-option**
The shell stays in the current working directory and setup the environment for the module to be used by another one. That is the given module is a dependency (and all dependents modules recursively) for the work to be done.
- run:"command args..."**
Run the given command with the module's environment variables set. This is meant to be used to run a command which is depending on some libraries or binaries that are in the sandbox.
- list-depends: [direct|full]**
Recursively list module's dependencies. Either list all dependencies recursively (the default) or only the direct dependencies of the given module.

- list-externals: [only]**
 Recursively list externals and modules dependencies. If sub-option `only` is specified it only displays the externals.
- list-no-version**
 Do not display modules or externals versions.
- list-filter: [all|run|dev]**
 Restrict the listing of dependencies or externals to either `all` (default) dependencies, the `dev` dependencies needed to build the software or the `run` dependencies needed to execute the software. Those last dependencies may be computed automatically from the `dev` dependencies listed in plugin modules.
- list-plan**
 List the final build plan that will be used. This is after loading main plan, inheriting from other plan `@load` and resolving the command line modules specification.
- module-version**
 Display module's version given in the command line or in the build plan.
- dry-run**
 Only list modules to be handled.
- stat: [full]**
 Display some statistics about of current setup. With sub-option `full` also gives stats about module's build time.
- n | --no-patch**
 Do not apply patches.
- j | --jobs=<n>**
 Max compilation jobs, set variable `SCBI_JOBS` which can be used in module scripts. (see section 2.2.1 p.9).
- safe**
 Build in safe mode. If a module needs to be built the previous build directory is removed. This ensures that the old configuration and/or build artifacts won't interact in any way with new build.
- plugins=<directory>**
 Specify a non default location for the modules and plug-ins directory. This overrides the `SCBI_DEFS` environment.
- store <KEY> <VALUE>**
 Set or retrieve private information based on `KEY`, `VALUE` pair. (see section 3.13 p.39)

A completion support for bash is provided. If activated many options will give hints about possible values like listing plans for `--plan=` option or listing the supported modules.

3 Creating new plug-ins

To create a new plug-in for SCBI you need to create a single `bash` script (one per module) into `$SCBI_DEFS` and named as the plug-in.

The plug-in script describes, using different hooks, the steps for building. One for configuring, one for building and one for installing. These are the mains steps generally used to build all software.

The different steps are described by a simple `bash` code snippet. And some hooks are just descriptions to retrieve artifacts (like VCS sources or tar file).

In the following section all the supported hooks are described. Then the calling order conducted by SCBI is described.

3.1 Hooks

The hooks must be prefixed by the name of the plug-in. Each of them are optional.

3.1.1 Description

<module>-vcs

Describe the module source VCS.

parameter NONE

return array

1. `proxy` : name of a proxy, or *default* to use default system one. (see section 3.4 p.31).
2. `options` : comma separated list of options, possible values are *none*, *no-recursive*, *lfs*
3. `protocol` : *git*, *svn*, *hg*
4. `URL` : repository location with protocol
5. `DIR` (optional): name of the directory if the repository is hosting multiple projects. This is mostly needed for Subversion non standard repository layout. If `URL` is pointing to the tree when `trunk`, `branches` and `tags` directories are found, then SCBI will check for the actual sources in `<root>/trunk/$DIR` or `<root>/branches/<VER>/$DIR` for example.

Example:

```

function lcms2-vcs ()
{
    echo default
    echo none
    echo git
    echo http://github.com/mm2/Little-CMS
}

```

It is possible specify a specific `vcs` hook in an environment file by two means:

SCBI_<module>_VCS_VARIANT

If environment variable `SCBI_<module>_VCS_VARIANT` is set then the hook `<module>-${SCBI_<module>_VCS_VARIANT}-vcs` is looked for before the generic `<module>-vcs`. This is specific to module name `<module>`.

SCBI_VCS_VARIANT Same as above for the all modules.

And finally the function `scbi-rewrite-vcs` can be defined. This function takes the five values returned by the `vcs` hook and must return the five values in the same order but possibly rewritten.

<module>-tar

Describe the tar filename and location of the tar containing the module sources.

parameter \$VERSION

return array

1. `proxy` : name of a proxy, or *default* to use default system one. (see section 3.4 p.31).
2. `options` : comma separated list of options to be used on the command line for the tool used (see protocol below). Use *none* if no option are needed.
3. `protocol`: *scp*, *cp*, *wget*, *curl*, *none*.
4. `URL`: location where the tarball can be downloaded.
5. `filename` : name of the archive with a possible \$VERSION in the pattern.

Example:

```

function cmake-tar ()
{
    local VERSION=$1

    echo default
    echo --no-check-certificate
    echo wget
}

```

```

    echo http://some.server.fr/sources
    echo cmake-$VERSION.tar.gz
}

```

It is possible specify a specific tar hook in an environment file by two means:

SCBI_<module>_TAR_VARIANT

If environment variable SCBI_<module>_TAR_VARIANT is set then the hook <module>-\${SCBI_<module>_TAR_VARIANT}-tar is looked for before the generic <module>-tar. This is specific to module name <module>.

SCBI_TAR_VARIANT Same as above for the all modules.

And finally the function scbi-rewrite-tar can be defined. This function takes the five values returned by the tar hook and must return the five values in the same order but possibly rewritten.

<module>-[common|<variant>]-patches

Returns the patch files to apply. REF is the actual module version making it possible to apply a patch selectively.

parameter \$PREFIX \$TARGET \$VARIANT \$KIND \$REF

return array

1. echo <patch-file-1>
2. echo <patch-file-2>
3. [[\$REF =~ "7.8"]] && echo <patch-for-7.8>
4. ...

<module>-version

Returns the current version of the installed module.

parameter \$PREFIX \$TARGET \$VARIANT

return version or \$VERSION_OK or \$VERSION_ERROR

1. the version as a string corresponding to the current installed version.
Can be \$VERSION_OK or \$VERSION_ERROR if the actual version cannot be determined.

<module>-external-env

Used to compute the module's build identifier and will trigger a rebuild of the module if the value returned by this hook changes. This may be needed if the module must setup its sources depending on some external environment. The return value has no semantic, it must just reflect a change of value, generally just returning the value of the controlling environment variable is sufficient.

parameter \$PREFIX \$TARGET \$VARIANT

return value

1. echo \$SOME_VAR

<module>-[common|<variant>]-build-depends

Lists all modules required to build the actual module. These are generally some tools like `cmake`, `swig` or `gcc` which are only needed to build but not needed at runtime. A dependent module can be specified as a full module-reference (see section 2.1 p.5) so with variant and/or version even if this is a generally a bad idea. To keep the module generic it is better to use the simple module name here and have a build-plan to specify a specific version and/or variant.

parameter \$PREFIX \$TARGET \$VARIANT

return array

1. echo <module1>
2. echo <module2>
3. ...

Example:

```
function aws-build-depends ()
{
    echo gnat
}
```

<module>-[common|<variant>]-tests-depends

Lists all modules required to run the test-suite for the module. These are generally some tools or frameworks like `cunit` which are only needed to run the tests.

parameter \$PREFIX \$TARGET \$VARIANT

return array

1. echo <module1>
2. echo <module2>
3. ...

Example:

```
function aws-tests-depends ()
{
    echo libcunit1
}
```

<module>- [common | <variant>] -depends

Lists all modules required to build the actual module and to use/execute it. These are either other modules handled by SCBI or distribution packages. A dependent module can be specified as a full module-reference (see section 2.1 p.5) so with variant and/or version even if this is generally a bad idea. To keep the module generic it is better to use the simple module name here and have a build-plan to specify a specific version and/or variant.

parameter \$PREFIX \$TARGET \$VARIANT

return array

1. echo <module1>
2. echo <module2>
3. ...

Example:

```
function aws-depends ()
{
    echo xmlada
    echo asis
}
```

<module>- [common | <variant>] -env

Set the environment to use this module as a dependency. This is the exported environment. The know how to set the environment using the helper routines see section 3.3 p.30.

parameter \$PREFIX \$TARGET \$VARIANT

1. add-to-var <name> <value>
2. ...

return NONE

Example:

```
function libgegl-env()
{
    local PREFIX=$1
    local TARGET=$2

    add-to-var PKG_CONFIG_PATH $PREFIX/lib/pkgconfig
    add-to-var PATH              $PREFIX/bin
}
```

<module>-[common|<variant>]-build-env

Set the environment needed to build this module. Those settings are not exported and only visible and used for the module. The `build-env` hook sees only the environment of the build dependencies.

parameter \$PREFIX \$TARGET \$VARIANT

1. `export <name>=<value>`
2. ...

return NONE

<module>-setup

To put in place the sources just before configuration step. This is generally not needed as the sources are setup by `schi` using the information provided by the `tar` or `vcs` hook. This may be needed in some complex case or if the sources are generated based on the running environment.

parameter \$PREFIX \$TARGET \$VARIANT

return NONE

<module>-out-of-tree

Whether to build out of the source tree or not.

parameter \$PREFIX \$TARGET \$VARIANT

return true | false

Example:

```
function lcms2-out-of-tree ()
{
    echo false
}
```

<module>-only-explicit-build

Whether to build when found in dependencies or only when the plug-in is passed on the command line. If function returns `true` then it will be built only when on the command line. In all cases it will be used to get the environment when found in dependencies.

parameter \$PREFIX \$TARGET \$VARIANT

return true | false

Example:

```

function lcms2-only-explicit-build()
{
    echo false
}

```

<module>-config

To configure the sources before starting the build. This is generally a call to GNU `configure` or `cmake`. The source tree is always accessible as `../src`.

parameter \$PREFIX \$TARGET \$VARIANT

1. `./configure --prefix=$PREFIX ../src`

return NONE

<module>-[common|<variant>]-config-options

Returns a list of options for the configuration step. The `config` (see above) hook is then supposed to be using the `get-config-options` routine to get the options to pass to the configuration tools like `cmake` or `configure` (see section 3.11 p.36).

parameter \$PREFIX \$TARGET [\$VARIANT]

return array

1. `echo <option1>`
2. `echo <option2>`
3. ...

<module>-build

The command to build the application. This is most of the case a simple call to `make`.

parameter \$PREFIX \$TARGET \$VARIANT

1. `make`

return NONE

Example:

```

function lcms2-build()
{
    local PREFIX=$1
    local TARGET=$2

    make -J$SCBI_JOBS
}

```

<module>-install

The command to install the application. This is most of the case a simple call to make with target install.

parameter \$PREFIX \$TARGET \$VARIANT

1. make install

return NONE

<module>-wrapup

This is to do whatever is needed after the installation step.

parameter \$PREFIX \$TARGET \$VARIANT

return NONE

<module>-prefix

The directory for the installation of the module. The returned value can be set to NONE to skip the installation step. If this hook is not defined the module will be installed into \$SCBI_PREFIX.

parameter \$PREFIX \$TARGET \$VARIANT

return string

1. echo <install_dir>
or
2. echo NONE

<module>-tests

Set of commands to run the tests for the module.

parameter \$PREFIX \$TARGET \$VARIANT [\$OPTIONS]

return NONE

3.1.2 Calling order

Some of the hooks are called by the main SCBI driver when needed as they are returning information of some artifacts. Those hooks do not have specific calling order:

- vcs
- tar
- version
- external-env

- out-of-tree
- prefix
- env
- build-env
- depends
- build-depends

For all other hooks there is a specific calling order which is:

1. setup
2. config
3. build
4. install
5. wrapup

It must be noted that for those steps we have generic internal version and pre, post corresponding steps:

module-pre-<step>

The generic pre step handled internally by scbi driver. Must not be redefined.

<module>-pre-<step>

The pre step action done before the main step.

<module>-<step>

The default step.

module-<step>

The generic step handled internally. Must not be redefined.

<module>-post-<step>

The post step action.

module-post-<step>

The generic post step handled internally. Must not be redefined.

For example, the generic internal setup step (`module-setup`) is used by the `scbi` driver to setup the sources (creating the build environment and populating the proper sources to compile) using the `vcs`, `version`, `tar` and `out-of-tree` hooks.

The `depends`, `build-depends`, `env`, `build-env` and `config-options` can have a `common` variant which is cumulative with the corresponding variant hooks or with the default hook. If a variant hook is present then the default hook is not called (we stop at the first `<variant>` call below:

1. `<module>-common-cross-<step>`
2. `<module>-<variant>-cross-<step>`
3. `<module>-cross-<step>`
4. `<module>-common-<step>`
5. `<module>-<variant>-<step>`
6. `<module>-<step>`

The `cross` hooks are used when the target is different than the host.

For all other hooks there is no `common` variant. Again if a variant hook is present the default hook is not called (we stop at the first `<variant>` call below:

1. `<module>-<variant>-cross-<step>`
2. `<module>-cross-<step>`
3. `<module>-<variant>-<step>`
4. `<module>-<step>`

3.2 Meta-module

A meta-module is a module listing a set of modules to build. This can be used to aggregate some related modules that are generally built together but not depending on each others.

Other modules can depend on a meta-module.

A module defining the hook `<module>-modules` (possibly with variants) becomes a meta-module.

`<module>-[common|<variant>]-modules`

Lists the aggregated modules.

parameter \$PREFIX \$TARGET [\$VARIANT]

return array

1. echo <module1>
2. echo <module2>

<module>-[common|<variant>]-propagate-version

Whether the meta-module version is to be propagated to aggregated modules.

parameter \$PREFIX \$TARGET [\$VARIANT]

return true | false

And can list a build-plan to use for building all listed modules:

<module>-plan

Returns the build-plan to use.

parameter NONE

return string

1. echo <plan-name>

Example:

```
function gnatcoll-plan()
{
    echo stable
}

function gnatcoll-modules()
{
    echo gnatcoll-core gnatcoll-bindings gnatcoll-db
}
```

A module listed in a meta-module can itself be a meta-module. A meta-module can define only a limited set of hooks, that is:

- [variant]-setup
- [variant]-propagate-version
- [variant]-build-depends
- [variant]-plan
- [variant]-modules

- [variant]-prefix
- [variant]-wrapup

It is possible to specify a version for a meta-module even if there is no sources as this version will be propagated to all aggregated modules and recursively for all meta-modules.

3.3 Setting environment

The five helper routines are to be used to set environment variables in `env` and `build-env` hook. They permit a fine control about the way the variables are set and avoid duplicate values.

define-var [PREPEND|APPEND|ADD|SET|UNSET] <VARIABLE> <VALUE>

Prepend, append, add or set a variable depending on the first action parameter. Note that a single character action is supported as alias.

PREPEND | P

Prepend <VALUE> in <VARIABLE>, remove previous value if it exists. This ensures that the value is found after the current ones if needed.

APPEND | A

Append <VALUE> in <VARIABLE>, remove previous value if it exists. This ensures that the value is found before the current ones.

ADD

Add <VALUE> in <VARIABLE> if it is not already present. The place of the value in the variable is not important, it may be at the start, middle or end.

SET | S

Set <VARIABLE> to <VALUE> removing all previous values. After this call we always have a single value set.

UNSET | U

Unset <VARIABLE>. After this call <VARIABLE> is not set.

prepend-to-var <VARIABLE> <VALUE>

short hand for `define-variable PREPEND <VARIABLE> <VALUE>`.

append-to-var <VARIABLE> <VALUE>

short hand for `define-variable APPEND <VARIABLE> <VALUE>`.

add-to-var <VARIABLE> <VALUE>

short hand for `define-variable ADD <VARIABLE> <VALUE>`.

set-var <VARIABLE> <VALUE>

short hand for define-variable SET <VARIABLE> <VALUE>.

unset-var <VARIABLE>

short hand for define-variable UNSET <VARIABLE>.

Example:

```
function mymodule-build-env()
{
    set-var MYMODULE_USE_CLANG no
}

function mymodule-env()
{
    add-to-var CMAKE_PREFIX_PATH /some/path
    prepend-to-var PATH /priority/dir/bin
}
```

3.4 Setting a proxy

When a Web protocol is used to retrieve a tar archive with the sources or a VCS it may be necessary to pass through a proxy. In the tar and vcs hook description the name of a proxy can be given. This name corresponds to a bash function named proxy-<name> that is used to set the proxy.

proxy-<name>

Set a proxy.

parameter NONE

return NONE

1. export https_proxy=...

A typical setting is:

```
function proxy-enterprise()
{
    export http_proxy=http://myproxy.org:3128
    export https_proxy=http://myproxy.org:3128
    export ftp_proxy=http://myproxy.org:3128
    export no_proxy=localhost,.net
}
```

This routine can be set into the environment file (see section 2.2 p.7) or in the meta-module (see section 3.2 p.28) description depending on use cases.

Also, note that this setting will be available only during the download of the archive file or the VCS and won't be polluting other module download not requiring a proxy.

3.5 Feature

It is possible to enable a feature that can be checked in the plug-ins using the command line `--enable-<name>`. This option activate the feature named `name`. This is a convenience function and it is up to the user to decide about the semantic of the enabled feature. For SCBI driver this is only a flag set or not.

In the module plug-ins the feature can be checked with `is-enabled` function:

```
is-enabled <name> [then-value] [else-value]
```

This function set the return code to 0 (true) if activated and returns `yes` by default or `then-value` if set. If the feature is not activated the function set the return code to 1 (as for errors) and returns the empty string by default or `else-value` if set.

Some example, for a single feature check:

```
| if [[ $(is-enabled installer) ]]; then
```

For checking two features:

```
| if [[ "$(is-enabled_appli)" = "yes" ]] \  
|    || [[ "$(is-enabled_installer)" = "yes" ]];  
| then
```

Or setting a specific compiler option, example of a config-options hook:

```
| echo -DCMAKE_CXX_FLAGS="-march=$(is-enabled_safe-arch_  
|    x86_64_native) "
```

3.6 Build kind

One of the most generic build variant that must be handled is the Release vs Debug one. To support this more easily without requiring to create a specific variant hook there is a simple support using the function `get-build-type`:

```
$ get-build-type <config> <default>
```

The `<config>` represents the configuration tool (CMAKE or CONFIGURE for example). But it can be whatever string.

This function will return the build-kind for the calling module and system type (cmake or configure) and if not specified the default value is returned.

The actual build kind for the given module can be set using some environment variables in the following order:

SCBI_<module>_BUILD_TYPE

Build kind for this specific module.

SCBI_<CONFIG>_BUILD_TYPE

Build kind for the builder kind.

And then this can be used in the config hook simply with:

```
function lcms2-config()
{
    local PREFIX=$1
    local TARGET=$1

    local BUILD_TYPE=$(get-build-type CMAKE Release)

    cmake -DCMAKE_INSTALL_PREFIX=$PREFIX \
          -DCMAKE_BUILD_TYPE=$BUILD_TYPE \
          ../src
}
```

With the above example, if no variables are set in the environment to set the build type the default `Release` will be used. To force a `Debug` build one can set in the environment file or in the current shell:

```
| export SCBI_CMAKE_BUILD_TYPE=Debug
```

Which will force a `Debug` build for all modules using `cmake`, or:

```
| export SCBI_lcms2_BUILD_TYPE=Debug
```

With will force a `Debug` build for the `lcms2` module only. In both cases, the corresponding module must have been coded to use the `get-build-type` routine as seen above. Note that the `export` must use *underscores* instead of dashes in module name as dashes are forbidden in variable name.

3.7 Checking installed tools

It may be needed to check for the existence of a tool installed on the machine. It may not be a direct dependency but a tool needed for a specific need. For example the need to have `makeself` installed to create a self installer.

Two helper routines are provided for this:

check-tool

Take the name of a tool as parameter, returns status code different than 0 if not found.

<i>parameter</i>	tool-name
<i>return</i>	NONE

Example:

```
check-tool makeself
if [ $? != 0 ]; then
    echo Required tool makeself missing
    exit 1
fi
```

check-tools

Take a list of tools as parameter, returns status code different that 0 if not found.

parameter tool-name1 [tool-name2...]
return NONE

Example:

```
check-tool wget rsync
if [ $? != 0 ]; then
    echo Required tools wget and/or rsync missing
    exit 1
fi
```

3.8 Inheritance

It is possible to create a plug-in module by inheriting from the definition of another module. To do this the first line on the module should be:

```
| inherit-plugin <name>
```

The <name> above is the name of an existing module from which we want to inherit all the current hooks.

After this line it is possible to redefine any hook as it would be done without inheritance. This may be handy if a module need to redefine only the `vcs` or `proxy` hook for a specific context.

3.9 Native variant

For a plug-in it is sometimes needed to have a native variant to use the OS libraries instead of building from sources. It is possible to do so by creating the variant hooks manually. All hooks must be nop action except the `depends` hooks which may list the OS libraries and possible the `env` hook which may need to define some environment variables.

The support routine `with-variant` can be used to automate this action:

```
with-variant native [<packages>:[v]|NONE] \
  --env 'cmd1_ 'cmd2'
```

The first parameter is the variant name. The followings are the OS package names or NONE. An optional environment section can be opened with `--env`. The value NONE will create an empty depends hook and can be used if the actual OS dependency has been installed manually on the machine. A possible reason is that the library has not been packaged by the OS maintainers. The `[v]` tag can be used to check that install `<package>` has the version specified for the plug-in.

For example:

```
with-variant native pckA:[v] pckB \
  --env 'set-var_ROOT_DIR_/usr'
```

The command above will automatically create the following hooks when called from 'natv' module script and if natv has version 1.2 in build plan:

```
function natv-native-build-env ()
{
  true
}

function natv-native-env ()
{
  set-var ROOT_DIR /usr
}

function natv-native-build-depends ()
{
  true
}

function natv-native-depends ()
{
  echo pckA:1.2
  echo pckB
}

function natv-native-config ()
{
  true
}

function natv-native-build ()
{
  true
}
```

```

}

function natv-native-install ()
{
    true
}

function natv-native-prefix ()
{
    echo NONE
}

```

3.10 Calling variant hooks

The support routine `call-variant-hook` can be used to call a set of hook in a module based on the variants.

```

| call-variant-hooks hname $PREFIX $TARGET $VARIANT \  

| [$P1 $P2...]

```

Will call all hooks in the current plug-in (<plugin>) with arguments `$PREFIX`, `$TARGET`, `$VARIANT` and optional parameters `$P<n>` in the form:

```

| <plugin>-<variant>-hname

```

Where <variant> is all possible values found in `$VARIANT`.

3.11 Getting configuration options

It is often needed to set configuration options based on the variant we want to build. This is for example to activate an external module.

```

| get-config-options $PREFIX $TARGET $VARIANT

```

Will call all hooks in the current plug-in (<plugin>) named:

```

| <plugin>-common-config-options  

| <plugin>-<variant>-config-options

```

The `common` hook is designed to contain all the common options to be used regardless of the selected variant.

For example:

```

function mod-mpi-config-options ()
{
    echo -DMPI=ON
}

function mod-openssl-config-options ()
{
    echo -DOpenMP=ON
}

function mod-common-config-options ()
{
    echo -DPREFIX=/my/path
}

function mod-config ()
{
    local PREFIX=$1
    local TARGET=$2
    local VARIANT=$3

    local OPTIONS="$(get-config-options_/usr_lin_$VARIANT) "

    cmake $OPTIONS ../src
}

```

If the current module reference is `mod/mpi` the `cmake` command will be:

```
| cmake -DPREFIX=/my/path -DMPI=ON ../src
```

And if the module reference is `mod/mpi.openmp` the `cmake` command will be:

```
| cmake -DPREFIX=/my/path -DMPI=ON -DOpenMP=ON ../src
```

3.11.1 Configuration options with spaces

If an option has spaces in it, the split will not be correct. This is a common problem and the solution is not trivial.

```
echo -DLINK_LIBRARIES="-lmpi_cxx -lmpi"
```

Then the `OPTIONS` string as described above won't work because of the way the shell will split the option on spaces by default. And so the actual options passed to `cmake` will be:

- `-DLINK_LIBRARIES="-lmpi_cxx`

- -lmpi

The solution is to use an array and modifying the field separator temporarily:

```
function mod-config()
{
    local PREFIX=$1
    local TARGET=$2
    local VARIANT=$3

    local IFS=$'\n'

    local OPTIONS=$(get-config-options /usr lin $VARIANT)

    cmake ${OPTIONS[*]} ../src
}
```

3.12 OS dependencies

The hook depends list dependencies to other plug-ins or external libraries/tools. As the name of the external dependency packages may be different depending on the Linux distribution and version it is possible to use an OS independent naming.

```
| os@-<name>
```

The `os@-<name>` is then search against a table to find the corresponding OS package name.

A distribution acronym is represented by a 3 letters name (deb, ubt, rhl for example). See output of the `-stat` option for the currently supported distributions.

An alias table is searched into the following places and in this specific order:

1. `.pkgs-<distrib>` into `$SCBI_DEFS` (default ' `/.config/scbi`')
2. `.scbi-pkgs-<distrib>` into `$SCBI_BDIR` (build directory)
3. `.scbi-pkgs-<distrib>` into `$PWD` (current directory)

In complement to the files searched based on distribution acronym, the major and minor version of the OS is also used.

For example on Debian 10.6 the searched files will be, and in that order:

1. `.pkgs-deb`
2. `.pkgs-deb-10`
3. `.pkgs-deb-10-6`

So `.pkgs-deb-10` can add new dependencies or override some dependencies that are set in `.pkgs-deb`.

3.13 SCBI Private Store

It is sometimes needed to use some private information (like software keys) to activate a module. Such information cannot be stored directly into a plug-in which could be made public. To cope with this specific case SCBI comes with support for storing pair of information into an external private store.

The option to set or retrieve a value is `--store`.

- On the command line:

To set a key:

```
$ scbi --store LICKY 89AXCDPPP6BGG
```

To list all stored keys:

```
$ scbi --store
```

To get value associated with a key:

```
$ scbi --store LICKY
```

- Inside a plug-in script:

To set a key:

```
scbi-store-set-key LICKY 89AXCDPPP6BGG
```

To get value associated with a key:

```
scbi-store-get-key LICKY
```

All keys are stored into the file `'$SCBI_PLUGINS/.store'` (if using the default environment `'$HOME/.config/scbi/.store'`).

4 Support tools

4.1 Completion support

A script is provided to support bash completion for scbi. The script is `'bash_completion.d/scbi'` and must be copied into `'/etc/bash_completion.d/'` OS directory.

Note that for this to work, the bash completion package must be installed, on Debian based distributions this is achieved with:

```
$ apt install bash-completion
```

4.2 Linter

The tool `scbi-lint` can be used to check for common plug-in errors.

For example:

```
$ scbi-lint lcms2
```

It will report:

- Function not recognized as hook name (warning),
- Lists all variants,
- Check that parameters `PREFIX`, `TARGET`, etc. are declared with proper parameter number,
- Check that the number of expected parameters is correct,
- Reports undeclared hook parameter (warning),
- Reports parameters used but not declared (error).

The option `--error` will only report errors.

5 Use Cases

5.1 User's checkout

This is certainly the most important use case. A complex software is built with SCBI, but a bug (due to configuration, build or dependencies) is found in one module. We need to debug this module and fix it, so the complete cycle would be:

- Check-out the module
- Compile manually in debug mode
- Debug the application
- Fix the code
- Commit the changes
- Push, integrate the new changes (possibly in a new branch)

SCBI does support this scenario directly by allowing a user's checkout to be used instead of the sources in the sandbox. It is highly discouraged to modify the sources in the sandbox which should stay untouched by users. Beware, depending on the SCBI options, the modifications in the source may be overwritten, so lost.

The recommended steps are (we assume the current VCS for the module is Git):

Checkout the module

The user need to manually checkout the module. This can be done in any directory usually used to do development:

```
$ cd $HOME/dev
$ git clone http://github.com/mm2/Little-CMS
```

Setup SCBI enviroment

The variable `SCBI_GIT_REPO` must be modified to point to the directory containing the user's Git checkouts. This is done by creating a new environment file, for example `'.env-debug'` (see section 2.2.2 p.9):

```
$ export SCBI_GIT_REPO=$HOME/dev
```

If the default value is used this step is not needed.

Create a build plan

The build plan will list all modules that must be compiled using a user's checkout. For example `'.plan-debug'`:

```
| lcms2:dev
```

Remember, `dev` is the special version referencing a user's checkout. (see section 2.1 p.5).

Modify the module's sources

As a standard development step we modify the sources in the local checkout to fix the issue:

```
$ cd $HOME/dev/Little-CMS
$ emacs somefile.c
```

When this is setup, building the new software (Little-CMS in this example) with a development check-out, one can just do:

```
$ scbi --env=debug --plan=debug --deps lcms2
```

We can further simplify by adding the build-plan reference in the environment file `'.env-debug'`:

```
| export SCBI_GIT_REPO=$HOME/dev
| SCBI_PLAN=debug
```

And then the new equivalent command line is:

```
$ scbi --env=debug --deps lcms2
```

If a single software with no dependencies are to be debugged it is possible to just call SCBI with a `dev` reference for the module. In this case there is no need to create an environment file nor a build-plan and we can simply call:

```
$ scbi lcms2:dev
```

Mercurial and Subversion VCS are also supported, the variables for each are:

Git	<code>SCBI_GIT_REPO</code>
Mercurial	<code>SCBI_HG_REPO</code>
Subversion	<code>SCBI_SVN_REPO</code>

5.2 Analyzing build failures

When a build fails it may be needed to look at the full logs and the actual commands executed for the failing step (configure, build, install...).

5.2.1 Replay one specific step

It is possible to replay a single command and display the full log to analyze what went wrong:

configuration

```
$ scbi --config --log:yes lcms2
```

build

```
$ scbi --build --log:yes lcms2
```

The files to be edited are in the sandbox module's directory:

```
'$SCBI_BDIR/lcms2/build/src'
```

Then one of the configuration commands above can be restarted. *When the fix is found it must be applied in the module's repository outside of the sandbox or it will be lost.*

Again, note that restarting an SCBI command without specifying a specific step (like --config or --build) will reset all sources from the reference repository in the sandbox.

It is possible to replay the configure, build or install step just by running the corresponding .cmd file. Use the Bash -x option to display the commands as they are executed.

5.2.2 Look at the sandbox status

The SCBI driver does record all needed information in the sandbox directory '`<module>/build/.log/`'.

The files created in this directory are:

'env.cmd'	The environment set to configure, build or install the module.
'config.cmd'	The commands executed to configure.
'config.log'	The full log issued during the configuration.
'build.cmd'	The commands executed to build.
'build.log'	The full log issued during the build.
'install.cmd'	The commands executed to install.
'install.log'	The full log issued during the installation.

In all the log files one can find a trace of every command executed. All trace line are prefixed with `_TRACE:.` The `'config.log'` contains the full command used to configure the module and the `'build.log'` the full command used to build the module.

As an example, the build command file for `c-cmake` plug-in looks like this:

```
# c-cmake-build
(
  cd <...>/c-cmake/build/build
  source <...>/scbi-runtime
  [ -f ../.log/env.cmd ] && source ../.log/env.cmd

  source ../.log/c-cmake

  c-cmake-build <...>/c-cmake/build/install \
                x86_64-linux-gnu default
)
```

Note that this script is executed in a sub-shell and so won't pollute the working shell environment.

Note also that the sources in the sandbox can be overwritten at any time. So the sources should not be edited directly on the sandbox. When the issue is found the recommended solution is to use a user's checkout, see section 5.1 p.41.

6 The Sandbox

We describe in this section the sandbox structure. Yet, it is generally not needed to go inside this hierarchy of directories except for debugging a new module being written (see section 5.2 p.42).

```
[$SCBI_BDIR]  
├── .scbi.runtime           : scbi runtime  
├── .vcs  
│   ├── [<module>]         : sources checkout  
│   ├── .patches  
│   ├── .tar  
│   │   ├── <module>-<version>-src.tar.gz  
│   │   └── [.logs]       : global log directory with history  
│   │       ├── [[YYYY-MM-DD]]  
│   │       │   └── *.log  
│   └── [<module>]  
│       ├── build →        : symlink to current/latest target build dir  
│       ├── build-id-<variant> : build id to check for changes  
│       ├── src →          : symlink to sources (vcs or tarball)  
│       ├── tar →          : symlink to extracted tarball  
│       ├── tarsrc         : extract of tarball  
│       ├── source-id      : version of the source (tar or vcs)  
│       ├── install-<target>-id : version of the module installed globally  
│       ├── vcs →          : symlink to repository (.vcs or user's checkout)  
│       └── [<target>-<variant>]  
│           ├── .log        : commands & output for each steps  
│           ├── [build]     : build directory  
│           ├── build-id    : sha1 to identify the build  
│           ├── [install]  : installation directory  
│           ├── manifest    : list of all files in install above  
│           └── src          : rsync of ../src
```

7 Complete example

Many scripts come with `scbi`, they are all good examples to learn about different features.

As a complete example here, let's build latest GIMP version. The plug-in module script is called `'gimp'` (placed into `'$SCBI_PLUGINS/.config/scbi'`) and all hook functions in this script are prefixed by `gimp-`.

Note that this was compiling at the time this doc was written, but some dependencies on the OS may required the script to be updated.

- We describe the VCS, sources are handled in Git:

```
function gimp-vcs()
{
    echo default
    echo none
    echo git
    echo https://github.com/GNOME/gimp
}
```

- We won't build GIMP out-of-tree, that is the build will be done in the source directory:

```
function gimp-out-of-tree()
{
    echo false
}
```

- We list all dependencies, most of them are standard Debian packages except `libmypaint`, `libmypaint-brushes` and `libegl`.

```
function gimp-depends()
{
    local PREFIX=$1
    local TARGET=$2
    local VARIANT=$3

    echo gtk-doc-tools
    echo libbabl-dev
    echo libgegl-dev
    echo libgexiv2-dev
    echo libpoppler-glib-dev
    echo python-gtk2-dev
    echo python-cairo-dev
    echo mypaint
```

```

echo libmng-dev
echo libxpm-dev

echo libmypaint
echo mypaint-brushes
echo libgegl
}

```

- We configure GIMP, note that the environment for the three dependent modules are set in the corresponding plug-in module:

```

function gimp-config()
{
    local PREFIX=$1
    local TARGET=$2
    local VARIANT=$3

    ./autogen.sh --prefix=$PREFIX
}

```

- We build GIMP:

```

function gimp-build()
{
    local PREFIX=$1
    local TARGET=$2
    local VARIANT=$3

    make -j$SCBI_JOBS
}

```

- We install it, this is the installation into the sandbox install directory for this module:

```

function gimp-install()
{
    local PREFIX=$1
    local TARGET=$2
    local VARIANT=$3

    make install
}

```

- The global installation is set to point to '/opt/gimp':

```

function gimp-prefix()
{

```

```

    echo /opt/gimp
}

```

As seen above, `gimp` module depends on three other modules and we do need some configuration from those dependencies during the configuration. The exported environment for the three dependencies are:

- Hook `libmypaint-env`:

```

function libmypaint-env()
{
    local PREFIX=$1
    local TARGET=$2
    local VARIANT=$3

    add-to-var PKG_CONFIG_PATH \
        $PREFIX/lib/pkgconfig
}

```

- Hook `mypaint-brushes-env`:

```

function mypaint-brushes-env()
{
    local PREFIX=$1
    local TARGET=$2
    local VARIANT=$3

    add-to-var PKG_CONFIG_PATH \
        $PREFIX/share/pkgconfig
}

```

- Hook `libegl-env`:

```

function libegl-env()
{
    local PREFIX=$1
    local TARGET=$2
    local VARIANT=$3

    add-to-var PKG_CONFIG_PATH \
        $PREFIX/lib/pkgconfig
    add-to-var PATH \
        $PREFIX/bin
}

```

The full description for those three modules are provided with scbi.

For this version of GIMP we need specific versions of the dependencies. A build-plan `.plan-gimpdev` is used to describe this:

```
| libmypaint:v1.3.0  
| mypaint-brushes:v1.3.0
```

That is, we need version 1.3.0 of `libmypaint` and `mypaint-brushes`.

At this point we can build GIMP with a simple:

```
$ scbi --plan=gimpdev --deps gimp
```

Index

- build kind, 32
- build plan, 10
- call-variant-hooks, 36
- check-tool, 33
- completion, 40
- environment, 7
 - file, 9
- feature, 32
- get-config-options, 36
- helpers
 - add-to-var, 30
 - append-to-var, 30
 - call-variant-hooks, 36
 - check-tool, 33
 - define-var, 30
 - get-build-type, 32
 - get-config-options, 36
 - inherit-plugin, 34
 - is-enabled, 32
 - prepend-to-var, 30
 - set-var, 30
 - unset-var, 30
 - with-variant, 34
- hook
 - build, 25
 - build-depends, 22
 - build-env, 24
 - calling order, 26
 - config, 25
 - config-options, 25
 - cross, 28
 - depends, 23
 - env, 23
 - external-env, 21
 - install, 26
 - modules, 28
 - only-explicit-build, 24
 - out-of-tree, 24
 - patches, 21
 - plan, 29
 - post-, 27
 - pre-, 27
 - prefix, 26
 - propagate-version, 29
 - setup, 24
 - tar, 20
 - tests, 26
 - tests-depends, 22
 - vcs, 19
 - version, 21
 - wrapup, 26
- inherit-plugin, 34
- inheritance, 34
- is-enabled, 32
- linter, 40
- meta-module, 28
- module reference, 5
- native variant, 34
- os dependencies, 38
- private store, 39
- proxy, 31
- source archive, 13
- user's checkout, 41
 - SCBI_GIT_REPO, 42
 - SCBI_HG_REPO, 42
 - SCBI_SVN_REPO, 42
- variable
 - add, 30
 - append, 30
 - prepend, 30

- set, 30
 - unset, 30
- variant
 - auto, 5
 - native, 5, 34
- version
 - dev, 6
 - force, 6
 - skip, 6
- with-variant, 34