# Hilscher Waf based build system V1.13.0.0

## Building libraries, applications and firmwares

DOC170404OI4EN | Revision 4 | English | Released | Public | 2023-10-19

# Table of Contents

# Chapter 1 Introduction

## 1.1 About this document

Waf is a Python-based framework for configuration, compiling and installing of software components. Based on this framework Hilscher has implemented a build system for generation of program libraries, applications and netX firmware.

This document describes the usage of the Hilscher Waf based build system and some internals. This document describes only features relevant for standard usage of the Hilscher Waf based build system. It does not describe rarely used Waf features (e.g. build visualization). Thus the Hilscher Waf based build system might be able to perform additional tasks and implement more feature than described here but: The functionality of any task or feature not described within this document cannot be guaranteed and might disappear on updated versions of Hilscher Waf based build system silently. It is strongly recommended to not rely on such undocumented functionality.

The Hilscher Waf based build system can be extended using additional modules. (E.g. Static Code Analysis Module, Asciidoc Module) These extensions might define additional command line arguments, commands, tasks or artefacts. Description of these modules is not part of this document but found in the corresponding extension module documentation.

## 1.2 List of revisions

| Revision | Date | Author | Revision |
|---|---|---|---|
| 4 | 2023-10-06 | AM | Migrate documentation to asciidoc |
| 3 | 2021-10-29 | AM | Document new commands for netX 90 / netX 4000. |
| | | | Document new command for documentation building. |
| | | | Document building .nxe/.nai/.nae/.mxf files. |
| | | | Document distribution of debug zip. |
| | | | Use netX 90 floating point coprocessor. |
| | | | Document new commands to set optimization flags. |
| | | | Document new sdram split offset settings. |
| 2 | 2018-01-12 | AM | Major rework of document. |

Table 1. List of revisions

## 1.3 Technical Information

The Hilscher Waf based build system was created on top of Waf build system V1.6.11 and still uses this version. In order to facilitate and ease distribution, a stripped down version of C-Python V2.7.3 was integrated into the Waf tool package provided by Hilscher.

## 1.4 Limitations

Currently there are no know limitations for the Waf build system.

## 1.5 References to documents

This document refers to the following documents:

| Ref | Document ID | Version | Document | Published by |
|---|---|---|---|---|
| [1] | - | V1.6.11 | The Waf Book | The Waf build system |

Table 2. References to documents

# Chapter 2 Overview about Waf

## 2.1 Hilscher Waf based build system structure

Normally, projects using the Waf build system package and provide all relevant waf scripts together with the project. Thus only a python interpreter is required to build the project. In contrast to that, Hilscher Waf based build system is split into two parts:

- the Waf build system using an integrated python interpreter which is part of the build tools provided by Hilscher and
- Hilscher specific extensions to the Waf build system.

Both parts together form the Hilscher Waf based build system which can be used to generate firmwares for Hilscher products or application software for various target types. Splitting the build system in this way allows to use a project specific version of the Hilscher extensions while preventing the need to contain the entire build system in every project. This helps to provide reproducible build of older projects.

The build system uses `wscript` files to define the build process of a project. These files declare targets, dependencies between targets and the structure of distributions to be generated. A project can also define custom build steps if needed. Precompiled libraries provided by Hilscher also ship a wscript file which allows for easy integration.

A typical project folder structure when used with Hilscher Waf based build system might look as follows:

**Snippet 1. Default Hilscher project structure**

```
Project/ (1)
Project/wscript (2)
Project/Components/A/{Includes,Sources,wscript} (3)
Project/Components/B/{Includes,Lib,wscript} (4)
Project/Documentation/D/{doc,wscript} (5)
Project/Targets/C/{Includes,Sources,wscript} (6)
Project/Tools/AsciiDoc (7)
Project/Tools/Waf (8)
```

(1) Base directory of the project

(2) Top level wscript of project containing global definitions & artefact distribution

(3) A source component

(4) A precompiled component

(5) A documentation to be generated

(6) A firmware target of the project

(7) Hilscher Waf extension for Asciidoc based documentation

(8) Hilscher Waf extensions / scripts

During execution of the build commands, waf will create additional folders within the project. These folder names are reserved for waf and can not be used for other purposes:

**Snippet 2. Files & folders generated by waf**

```
Project/build (1)
Project/dist (2)
Project/.lock-waf_* (3)
```

(1) Folder generated by Waf during the build process. It will contain all files generated during the build. (e.g. objects, libraries)

(2) Folder containing the artefacts to distribute. Will be generated during the `install` subcommand

(3) Lock file used by waf itself. (Normally hidden)

## 2.2 Build commands

The Waf build system divides the build process into subcommands parts which can be run separately or in single invocation as needed. Subcommands typically depend on successfull execution of the previous subcommand.

1. The first part is to collect command line options and parse the command line. This part is always performed and not associated a separate command. Waf will look for an `options()` function in the toplevel wscript and execute it to retrieve additional command line options defined by the project. Afterwards the command line will be parsed.
2. The **configure** subcommand configures the project by loading additional waf modules, locating required toolchains, tools, setting up environment variables or performing project specific tasks. Waf will execute the `configure()` function in the top-level wscript for this purposes. Afterwards the configuration will be stored in cache files for later use. Running the *configure* subcommand is a pre-requisite before most of the other commands can be used.
3. After beeing configured, the project can be build using the **build** command. To perform the build, Waf will execute the `build()` function defined in top-level wscript. The *build* command by itself is again splitted into three stages which are explained below.
4. An extension to the *build* command is the Hilscher specific **install** command. It is used generate artefact distributions. The structure of these distributions is defined in top-level wscript within the `build()` function. Since the *install* command is just an extension of *build* command, it can be run without running *build* command before, it will carray out all necessary build tasks as well.

The example Invoke waf to build a project shows a typical command-line to build a project with waf.

**Example 1. Invoke waf to build a project**

```
$ waf configure build
```

In order to optimize the build speed waf analyzes dependencies between build tasks and will skip tasks if their dependencies have not changed since the last build.

## 2.3 Environment objects

The Waf build system uses environments to store information about compiler & tool paths, compile flags and other parameters. Environments can inherit variables from other environments. Thus a variable set in a parent environment is also visibile in a child environment as shown in figure Environment relations

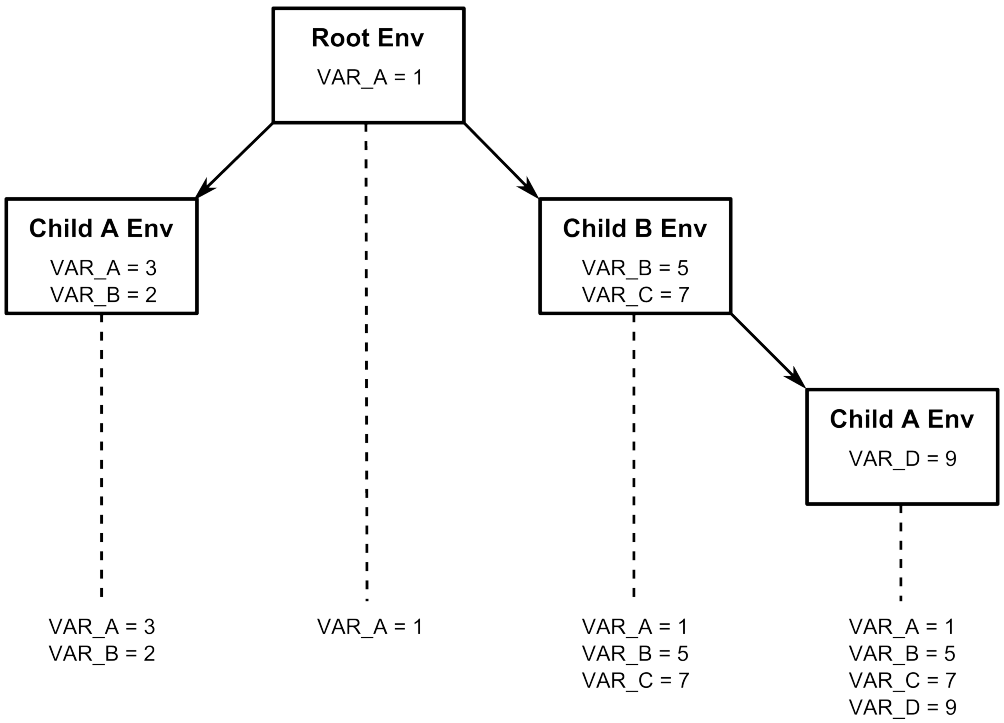


Figure 1. Environment relations

From that figure we can see, that if *VAR_A* is modified in Root Environment, this modification immediately affects all child environments *Child B* and *Child A* which do not override this variable with an own setting in this example.

The Waf build system will create environments for each loaded toolchain during processing the *configure* command. The contents of these environments will be stored in `build/c4che` folder for use by all other commands. (This is the reason why a project must be configured before any other command can be used) However, the relationship between the environments is not stored. Thus while a toolchain environment will incoporate changes of the `root` environment during processing a *configure* command it will not do so during processing of *build*, *install* or other commands. This is illustrated in figure Environment structure of Hilscher Waf build system.

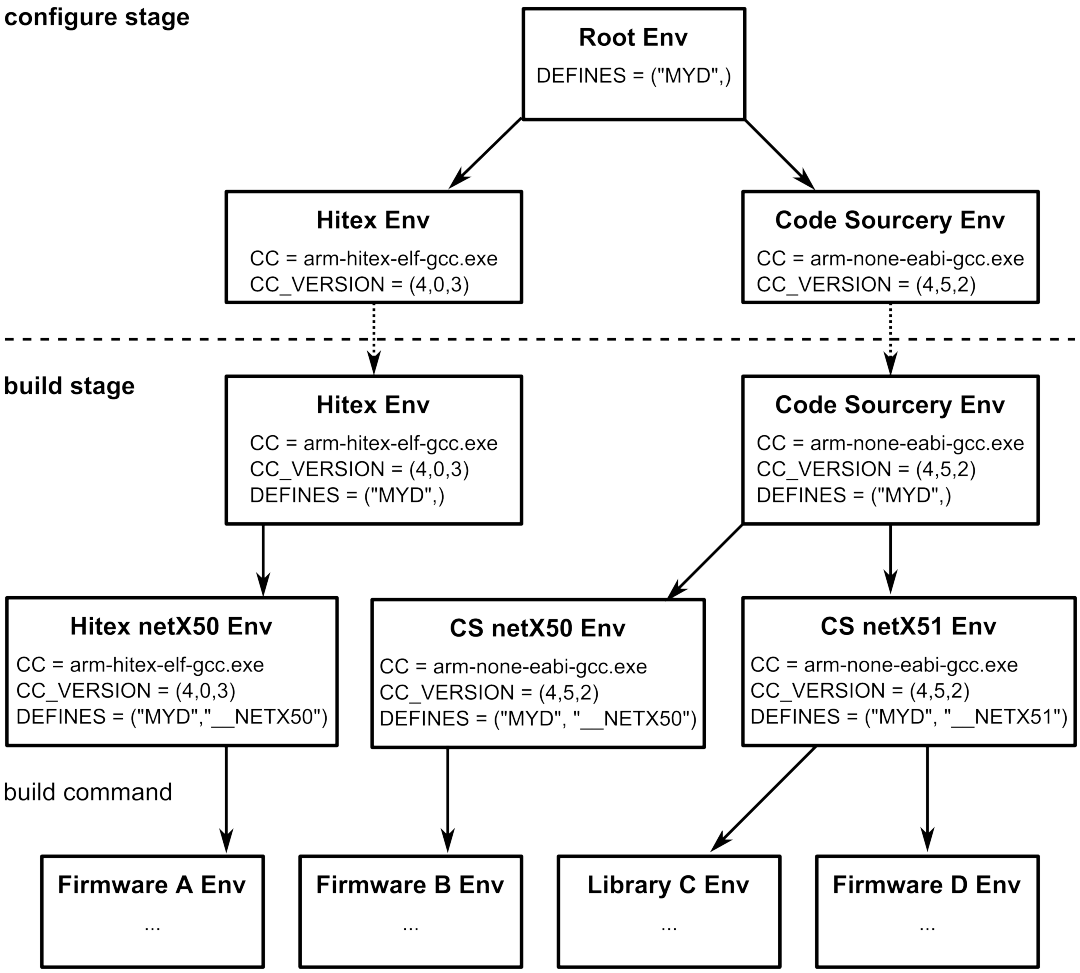


Figure 2. Environment structure of Hilscher Waf build system

Based on the toolchain environments created during the *configure* command, the Hilscher Waf based build system creates a tree of derived environments during processing build commands:

- An environment for a platform/target will be derived from the selected toolchain. For the same platform/target a separate environment is be created for each used toolchain.
- Each target is derived an environment from the selected platformF/target environment.

Thus a change/modification to the toolchain environment may propagate down to the derived targets if they are not overwritten at some level in between.

## 2.4 Build processing

The Waf build system organizes processing of a build command as follows:

1. At the beginning Waf build system will read the top-level wscript file and execute the `build()` function contained in this file. This function defines the targets to be build and may also recurse into subdirectories to find further wscript files and execute their `build()` functions accordingly. A target is defined by calling the corresponding functions and will create one or more **task generator** objects. The Hilscher Waf based build system will also create the target specific environments during this step. At the end of this step, after processing the wscript files, Waf build system has made a list of task generators.
2. The next step is to process the *task generator* by applying a selection of **task generator functions** to each of the *task*

*generators*. Waf build system uses the *features* parameter to decide which of the available *task generator functions* are applicable to a *task generator*. The *task generator functions* will create *tasks* (e.g. compile. link, etc) required to build the target associated with a *task generator*. They will also determine the dependencies of the tasks.

3. Finally Waf build system will run all **tasks** generated during the previous step. This typically involves calling external commands like compilers, linkers or other tools. Depending on a task and its dependencies, Waf build system might choose to not execute a task if its inputs or dependencies have not been changed.

**NOTE** Processing task generators and tasks happens in parallel using all available CPU cores of the build host. It might be desired to run only one task at a time to debug issues. This can be achieved by using the "-j" command line option as follows:

```
$ waf -j1 configure build
```

Due to this structure it is easy to extend existing build flows (e.g. static library, linked application) of Waf build system with own build steps and tasks since a *task generator function* defined in an extension can easily hook into them.

In order to allow seamless integration of components which might be providing additional artefacts for targets not used in current project or to handle missing dependencies the Hilscher Waf based build system implements an mechanism to automatically deactivate processing of task generators. This may happen in the following cases:

- a target uses a toolchain not loaded in the current project
- a target depends on a nother target (e.g. static library) which is not available or disabled in the current project

When a task generator is disabled by Hilscher Waf based build system, a warning printout is be generated but the build will not fail. However when a deactivated target is part of a distribution command, Hilscher Waf based build system will fail during processing of the *install* command to avoid incomplete distributions.

## 2.5 File system representation

Waf build system internally uses *Node* objects to represent the file system. A *Node* is created for each file or directory declared or located during the build process. Node objects are organized in trees as the file system is. A *Node* may refer to a non-existing file or directory. This is typically the case for files and directories to be generated during the build. (e.g. object files)

# Chapter 3 Usage

This chapter describes typical use cases and their implementation using Hilscher Waf based build system. For a reference description of the various commands used in the following sections please refer to section [section-command-reference].

## 3.1 Running Waf

The Waf build system must be invoked from the folder which contains the project's top-level wscript file. The build process is influenced by environment variables and command line arguments.

**NOTE** Waf build system can be extended with additional modules. These modules may evaluate othe environment variables and define additional command line arguments. For a description of these please refer to the corresponding module documentation.

### 3.1.1 Environment variables

Hilscher Waf based build system may evaluate several environment variables in order to locate toolchains and other build tools. Depending on the toolchains used to build the project it is mandatatory to set these variables properly. Ohtwerise the *configure* command will fail to locate the executables.

#### PATH

The *PATH* variable is commonly used on most systems to define the search path of executables. This variable should be configured to include the folder of the Waf tool provided by hilscher. Depending on the project the variable should also include folder paths to python (`python.exe`) and subversion (`svn.exe`) executables. Optionally, the *PATH* variable may also include paths to Hitex(`arm-hitex-elf-gcc.exe`) and CodeSourcery (`arm-none-eabi.exe`) toolchains. These will be used as fallbacks if the specific vars are not set.

#### CS_PATH

The *CS_PATH* variable is used by Hilscher Waf based build system to determine the location of the CodeSourcery toolchain. The variable shall point to the base folder of the CodeSourcery toolchain, that is Hilscher Waf based build system will look for `bin/arm-none-eabi-gcc.exe` within than folder. If the variable is not set, the CodeSourcery toolchain will be looked up using the *PATH* environment variable.

#### GCC_ARM_PATH

The *GCC_ARM_PATH* variable is used to locate the GNU Arm Embedded Toolchain. The variable shall point to the base folder of the toolchain, Hilscher Waf based build system will look for `bin/arm-none-eabi-gcc.exe` within than folder. The toolchain can only be used if the variable is set.

#### PATH_LLVM_XPIC

The Hilsxher xPIC toolchain is located by Hilscher Waf based build system using the *PATH_LLVM_XPIC* environment variable. The variable shall point to the base directory of the toolchain. This directory contains `bin/xpic-llvm.exe`. If the variable is not set, the xPIC toolchain will be looked up useing the *PATH* environment variable.

### 3.1.2 Command invocation

The syntax of invoking the Hilscher Waf based build system tool is as follows:

```
waf.bat [options] command [another_command [...]]
```

The command argument is mandatory while the options cen be omitted if not needed. The commands are executed in the order they have been specified at the command line. The following commands are understood by Hilscher Waf based build system:

#### configure

The *configure* command will prepare the toolchain environments to be used by the other commands. A project must be always configured once before any of the build commands can be used. It is strongly recommended to always run the *configure* prior using build commands to avoid unexpected issues due to changed wscript files.

#### build

The *build* command generates (compile, link etc.) all artifacts defined by Depending on the build condition, the output is generated within the `build/${condition}` subfolder of the project.

### install

The *install* command is modified in Hilscher Waf based build system such that the artifacts are distributed in the `dist` subfolder according to the distribution definitions made in the wscript files

### clean

The clean command removes all build artifacts within the `build` subfolder

### list

The *list* command is an extension of Hilscher Waf based build system which will print a list of all targets defined in the project to the command line.

The build process of Hilscher Waf based build system can be further customized using optional command line arguments as follows:

### --conditions=<debug|debugrel|release>

With this option it can be choosen if a *debug*, *debugrel* or *release* build is performed. Debug symbols are generated in all three cases but, the *debug* and *debugrel* build may activate project specific debug code. In addition, the *debug* case also disables any compiler optimizations. This provides better debugability at the cost of more stack use, bigger code size and lower performance. (default: *release*)

### --destdir=destination_directory

Adjust the destination director for distribution folder. (default: *dist*)

### --dump-environment=variables,...

With this option, Hilscher Waf based build system will generate a text file along with the build artefact, which contains the value of selected variables used in a build setp. The information will be broken down to each single task executed. Commonly used variables are: CFLAGS, CXXFLAGS, LINKFLAGS, DEFINES, LIB, INCLUDES and STLIB.

### --generate-report=reports,...

Hilscher Waf based build system can generate reports related to build process or build artifacts. This parameter is a comma separated list of values selecting which reports shall be generated. Currently the following values can be used:

#### all

Generate all known reports

#### size

Generate a report of the allocated size of functions, global variables and size statistics of an elf file. This is useful when the firmware size shall be optimized.

### -j jobs, --jobs=jobs

Specifies how many processes should be run in parallel. Whenever possible Waf build system tries to run jobs in parallel to speed up the build process. (default: number of cores of the system)

### --targets=target,...

This option can be used to build only a subset of the targets defined in the project. Waf build system will only build the specified targets and their dependencies. (default: build all targets)

**Example 2. Example waf invocations**

Generate a debug build without any optimizations:

```
$ waf.bat --conditions=debug configure build
```

Generate a release build and distribution:

```
$ waf.bat configure install
```

## 3.2 Preparing a project for Waf

This section describes how to setup a project for uses with the Hilscher Waf based build system. While some of the steps are identical to standard Waf build system, some Hilscher specific settings have to be made. The folder/file structure used in the following description is based on Hilscher definitions for the development. These might not fully apply in all cases.

The following listing shows which steps are required to prepare the project for use with the Waf build system:

1. Add the Hilscher Waf based build system scripts to your project. For convience its is recommended to locate them in subfolder `Tools/Waf`. A typical Waf script folder will look like:

```
MyProject/Tools/Waf
MyProject/Tools/Waf/__init__.py
MyProject/Tools/Waf/hilscher_dist.py
MyProject/Tools/Waf/hilscher_doc.py
MyProject/Tools/Waf/hilscher_extras.py
MyProject/Tools/Waf/hilscher_firmware.py
MyProject/Tools/Waf/hilscher_netx.py
MyProject/Tools/Waf/hilscher_toolchains.py
MyProject/Tools/Waf/...
```

   It is good practice to integrate the Waf tool folder via a change management system and using an external (subversion) or submodule (git).

2. A top level *wscript* file with path `MyProject/wscript` must be created as entry point for Waf build system. It will typically initialize common settings, load toolchains and recurses into subdirectories to locate and load other *wscript* files. The initial content of this file might be as follows

   **Example 3. Basic top-level wscript file content**

```
out = 'build' # waf build folder name

# Space separated string or list of strings of toolchain modules
# to load. Variable used later on when loading modules
toolchain_modules = """
   hilscher_toolchain_ecoscentric
"""

def options(opt):
   # prepare waf.bat command line arguments from modules
   # additional extensions might be added here
   opt.load('hilscher_netx hilscher_firmware hilscher_extras', tooldir='Tools/Waf')
   opt.load(toolchain_modules)

def configure(conf):
   # load basic Hilscher waf extension modules
   # additional extensions might be added here
   conf.load('hilscher_netx hilscher_firmware hilscher_extras', tooldir='Tools/Waf')
   opt.load(toolchain_modules)

   # load subdirectory wscripts and run their configure() function
   # if available. Do not recurse into “dist” or “build” subfolder. They
   # are generated upon build and might contain generated wscripts
   conf.autorecurse(mandatory = False, excl= ["dist", "build"])

def build(bld):
   # load subdirectory wscripts and run their build() function
   # fails if wscript files without build() function is found
   # Do not recurse into “dist” or “build” subfolder. They
   # are generated upon build and might contain generated wscripts
   bld.autorecurse(excl= ["dist", "build"])
```

3. For libraries and targets of the project itself wscript files need to be created as needed. These files define the artefacts of the project and how they are generated. If the default Hilscher project file system s tructure is used this might look

as follows:

```
MyProject/wscript
MyProject/Components/MyApp/Includes
MyProject/Components/MyApp/Sources
MyProject/Components/MyApp/wscript
MyProject/Components/eCosPro/...
MyProject/Components/...
MyProject/Targets/MyFirmware/Sources
MyProject/Targets/MyFirmware/wscript
MyProject/Targets/MyFirmware/netx90_usecase_a.ld
MyProject/Targets/MyFirmware/netx90_ecos_iflash.xml
```

Hilscher supplied code or pre-compiled libraries must be placed within the project structure as well. (E.g. rcX operating system, protocol stack libraries) These are usually shipped with *wscript* files which declare the libaries to Waf so the can be referenced later.

- For building a static library (e.g. a component of the project) the following *wscript* listing can be used as a starting point:

**Example 4. Template wscript for static library**

```
def build(bld):
    # compute a platform prefix for the target
    lib_prefix = bld.get_name_prefix(
        toolchain = "ecoscentric",
        platform  = "netX90"
    )

    bld.stlib(
        target        = "myapp", # library file will be named libmyapp.a
        name          = lib_prefix + "myapp", # task generator name
        toolchain     = "ecoscentric",
        platform      = "netX90",
        description   = "My Application library",
        source        = [
            "Sources/MyApp_Body.c",
            "Sources/MyApp_Resources.c",
            "Sources/MyApp_Process.c",
            "Sources/MyApp_Functions.c",
        ],
        includes      = [
            "Includes"
        ],
        defines       = [
            "MY_DEFINE=1"
        ],
        use = [
            'armv7em-ecoscentric-ecospro-eabi/*/ecos_sdk'
        ],
        export_includes = [
            "Includes"
        ],
    )
```

- The next listing shows how a firmware to be run on a netX can the build

**Example 5. Template wscript for firmware**

```
def build(bld):
    bld.firmware(
        target         = 'myfw.nxi',
        toolchain      = 'ecoscentric',
        platform       = 'netx90',
        source         = [
```

```
        'Sources/config.c',
    ],
    use = [
        'armv7em-ecoscentric-ecospro-eabi/*/myapp',
        'armv7em-ecoscentric-ecospro-eabi/*/ecos_target',
    ],
    features    = 'buildstamp libsused',

    linkerscript = 'netx90_usecase_a.ld',

    netx_type    = 'NETX90',
    hboot_xml    = 'netx90_ecos_iflash.xml',
)
```

## 3.3 General wscript rules

When writing a *wscript* file, the following rules should be followed for convience and interoperability:

1. *wscript* files are written in the python script language as Waf build system itself is. Thus special care must be taken regarding the indentation of a statement. In python language the beginning or end of a block (function, conditional, loop) is marked by indentation. All statements belonging to a block must be using the same indent after the block has been opened by the corresponding python keyword. The amount of indent does not matter, for readability four spaces are considered good practice.

```
def build(bld):

    bld.stlib() (1)
     bld.firmware() (2)
```

(1) Block indent by first line to be 4 spaces

(2) Wrong ident of 5 spaces used

Lines following an opening brace are considered part of the statement of the opening brace itself until the corresponding closing brace is encountered. Here indent has no special meaning. Opening braces must occur on the same line as the statement they belong to, they must not be moved to the next line.

```
def build(bld):

    bld.stlib( (1)
      target = "mylib" (2)
     )

    bld.firmware() (3)
```

(1) Opening brace

(2) Ident does not matter here since we're in a brace

(3) Ident does matter and must be 4 spaces again

If required lines can be continued using the backslash ('\') at the end of the line

2. Multiple build commands within a single *wscript* file must be put into a single `build()function. It is not possible to define multiple functions with same name within a single wscript file.

   **WARNING** Beware that double function names are not considered as a syntax error in python. The last definition found just overwrites all previous definitions.

3. Paths specified within a wscript file are always relative to the directory containing the wscript file itself.
4. Waf build system supports modular projects with separate wscript files for components and/or targets. Waf build system does not implicitly search and load all wscript files within a project. For that purpose the functions [section-function-recurse] or [section-function-autorecurse] are to be used.

## 3.4 Selecting / loading toolchains

Hilscher Waf based build system supports building targets which use different toolchains at the same time. This is implemented by storing the paths, flags and variables associated with a toolchain in an *environment*. These toolchain *environments* are created during the *configure* command. It is part of the project configuration, which toolchains should be loaded and made available for the build process. This is achieved by loading the corresponding toolchain module as follows:

```
def configure(conf):
    conf.load('hilscher_netx hilscher_firmware hilscher_extras', tooldir='Tools/Waf') (1)
    conf.load('hilscher_toolchain_ecoscentric') (2)
```

(1) At first load the generic Hilscher modules

(2) Load a hilscher toolchain module

Since toolchain modules might also define additional command line arguments, they should be loaded in `options()` function as well. To avoid a mismatch between options and configure commands, it is sensible to use a variable as shown in example Loading toolchains

**Example 6. Loading toolchains**

```
project_toolchains = """ (1)
  hilscher_toolchain_ecoscentric
  hilscher_toolchain_codesourcery
"""

def options(opt):
    opt.load('hilscher_netx hilscher_firmware hilscher_extras', tooldir='Tools/Waf')
    opt.load(project_toolchains) (2)

def configure(conf):
    conf.load('hilscher_netx hilscher_firmware hilscher_extras', tooldir='Tools/Waf')
    conf.load(project_toolchains) (3)
```

(1) Create whitespace separated string with toolchains to load (Can be also array of strings)

(2) Load the toolchains to get additional command line arguments

(3) Load & detect the toolchains

The toolchain modules provided by Hilscher Waf based build system itself are listed in table Toolchains provided by Hilscher Waf based build system. An external extension or the project itself might define additional toolchains.

| Name | Module | Description |
|---|---|---|
| ecoscentric | hilscher_toolchain_ecoscentric | eCosCentric toolchain to be used with eCos operating system |
| codesourcery | hilscher_toolchain_codesourcery | CodeSourcery toolchain used with netX50, netX51, netX52, netX100 and netX500. Typically in conjunction with rcX operating system |
| gccarmemb | hilscher_toolchain_gccarmemb | GNU arm embed toolchain. A generic ARM toolchain typically used with netX90 APP side |
| llvm-xpic | hilscher_toolchain_xpic | LLVM based toolchain provided by Hilscher Gesellschaft für Systemautomation mbH. For use with xPIC found on some of the netX devices |
| hitex | hilscher_toolchain_hitex | Legacy toolchain from Hitex used in old projects with netX50/netX100/netX500 |

Table 3. Toolchains provided by Hilscher Waf based build system

## 3.5 Compiling a static library

A static library is generated using the build command `stlib()`. Static libraries can be an artefact of the project to be used by other projects or used within the project itself to group parts of the application. They also help to avoid multiple compilation of same file within one project if the files are used by multiple targets of a project and compiled with same

settings. The following example shows how to declare a static library in a *wscript* file.

### Example 7. Compiling a static library

The declaration must occur within the `build()` function in the wscript file. Besides the static library itself an additional *SDK* task generator should be defined if the static library is a dependency of other libraries:

```
def build(bld):
    bld.sdkcomponent(
        name = "mylib_sdk", (1)
        description    = "My Static library SDK",
        export_includes = ['Includes'],
    )

    lib_prefix = bld.get_name_prefix( (2)
        toolchain = "ecoscentric",
        platform  = "netx90"
    )

    bld.stlib(
        name          = lib_prefix + "mylib", (3)
        target        = "mylib", (4)
        toolchain      = "ecoscentric",
        platform       = "netx90",
        description    = "My Static library",
        source         = [
            "Sources/MyLib_Source1.c",
            "Sources/MyLib_Source2.c",
            "Sources/MyLib_Source3.c",
        ],
        includes       = [
            "Includes"
        ],
        defines        = [

        ],
        use = [
            'mylib_sdk', (5)
            'armv7em-ecoscentric-ecospro-eabi/*/ecos_sdk'
        ],
    )
```

(1) The name of the SDK task generator for *use* in other targets. SDKs should not include a toolchain/platform prefix if possible

(2) Compute the Hilscher specific toolchain/platform prefix

(3) The name of the library task generator for *use* in other targets. Static library targets usually have a toolchain/platform prefix in this name

(4) The name of library file (will become *mylib.a* here)

(5) Reference our own sdk

If a static library depends on another static library, the SDK of that library is to be referenced to get include paths and defines exported by that library. This is shown in example Compiling static library depending on another static library

### Example 8. Compiling static library depending on another static library

A static library might depend on other libraries. Such dependencies are declared with the *use* argument. Its value is a list of names of other task generators which this task generator depends on.

```
def build(bld):
    lib_prefix = bld.get_name_prefix(
        toolchain = "ecoscentric",
```

```
    platform  = "netx90"
)

bld.stlib(
    target        = "myapp",
    name          = lib_prefix + "myapp", # task generator name
    toolchain      = "ecoscentric",
    platform       = "netx90",
    description    = "My Application library",
    source         = [
        "Sources/MyApp_Source1.c",
        "Sources/MyApp_Source2.c",
        "Sources/MyApp_Source3.c",
    ],
    includes       = [
        "Includes"
    ],
    defines        = [
        "MY_DEFINE=2"
    ],
    use = [
        'mylib_sdk', (1)
        'armv7em-ecoscentric-ecospro-eabi/*/ecos_sdk'
    ],
)
```

(1) Use the SDK from *mylib* since *myapp* depends on it

**NOTE** In most cases of building static libraries the *use* parameter should refer to SDK task generators only. Referencing another static library task generator directly will have an impact on the final linking step and might cause unexpected linker errors.

## 3.6 Link a executable file

Using the command `bld.program()` a task generator for linking an executable file can be declared. Such a file might be used with a debugger or directly executed if it was compiled for the host.

**NOTE** In most cases at Hilscher it makes more sense to directly Generate a firmware. This will implicitly generate the executable file

The following example shows how to link an executable file for netX51 and rcX V2.1 .Link executable for netX51

```
def build(bld):
    lib_prefix_rcx21 = bld.get_name_prefix(
        toolchain = "codesourcery",
        suffix="rcX_V2.1" (1)
    )

    bld.program( (2)
        target      = 'myfirmware', (3)
        platform    = 'netx51',
        toolchain   = "codesourcery",
        description = 'My firmware for netX51',
        source      = [
            "Sources/config.c",
        ],
        use= [
            'arm-none-eabi/*/rcX_V2.1/mylib',
            'arm-none-eabi/*/rcX_V2.1/myapp',
            'arm-none-eabi/*/rcX_V2.1/rcx_mid',
            'arm-none-eabi/*/rcX_V2.1/rcx_vol',
            'arm-none-eabi/*/rcX_V2.1/rcx_netx51',
            'arm-none-eabi/*/rcX_V2.1/rcx',
        ],
```

```
    linkerscript    = "netX51_sdram.ld",
)
```

1. rcX based targets use an extra suffix in the target prefix string
2. A task generator not further referenced does not need a name
3. output file will be named named myfirmware.elf

## 3.7 Declaring pre-compiled components

In some case it might be required to integrate pre-compiled third-party components into a project. E.g libraries provided by external sources. For this use case, the libraries and header files should be placed in a directory structure as shown in the following listing

```
MyProject/Components/ExtCom/Includes
MyProject/Components/ExtCom/Includes/ExtCom.h
MyProject/Components/ExtCom/Lib/libextcom.a
MyProject/Components/ExtCom/wscript
```

With a *wscript* which declares the precompiled libraries as shown in example Declaring a precompiled library

**Example 9. Declaring a precompiled library**

```
def build(bld):
    bld.sdkcomponent( 1
        name = "extcom_sdk",
        description     = "My pre-compiled library SDK",
        export_includes = ['Includes'],
        use             = [],
    )

    bld.externalcomponent(
        name          = "arm-none-eabi/*/extcom",
        target        = "extcom", 2
        path          = "Lib/",   3
        description    = "My pre-compiled library ",
        version       = "4.0.0.3", 4
        export_includes = ["Includes",],
    )
```

1. The should be always an SDK task generator for precompiled libraries
2. The name of the external library without suffix
3. The directory the external library resides in
4. The version of the external library

**NOTE** If the pre-compiled library is also created with Hilscher Waf based build system there is no need to manually craft a precompiled component. Hilscher Waf based build system can automatically generate a suitable precompiled component structure as build artifact.

## 3.8 Generate a firmware

The generation of a firmware image depends on various requirements of the platform, operating system and system design. Therefore there are different arguments to be used depending on the firmware target.

Since the build process of a firmware is strongly coupled to the linking process of the executable, the `bld.firmware()` command is derived from the `bld.programm()` step and replaces it. In contrast to `bld.programm()`, the target parameter in `bld.firmware()` command specifies the name of the firmware file. Hilscher Waf based build system will use the extension of this filename to determine the kind of firmware to build. Also the name of the linked executable is derived from the firmware file name by replacing the extension with `.elf`.

The following sections provide an overview about the different firmware usecases and how to write a *wscript* file to build a corresponding firmware.

### 3.8.1 Loadable firmware (nxf) for netX 10 / netX 50 / netX 51 / netX 52 / netX 100 / netX 500

A loadable NXF firmware is used in a two stage boot process. After Power On the netX rom loader loads and executes the second stage loader. The second stage loader can be loaded from Serial Flash, Parallel Flash or even from DPM by the Host. The second stage loader initializes some of the netX hardware (e.g. SDRAM). Afterwards it locates a NXF firmware file in the file systems on permanent storage devices (Flash-based boot mode) or expects a NXF firmware via DPM (RAM-based boot mode). If a firmware was found or provided, the firmware will be loaded and executed.

Loadable firmware files are identified by file extension “.nxf”. The following example listing shows how a NXF firmware file can be generated.

**Example 10. Building a loadable NXF firmware**

```
def build(bld):
    bld.firmware(
        target      = 'myfirmware.nxf', (1)
        platform    = 'netx51',
        toolchain   = "codesourcery",
        description = 'My loadable firmware for netX51',
        source      = [
            "Sources/config.c",
        ],
        use= [
            'arm-none-eabi/*/rcX_V2.1/mylib',
            'arm-none-eabi/*/rcX_V2.1/myapp',
            'arm-none-eabi/*/rcX_V2.1/rcx_mid',
            'arm-none-eabi/*/rcX_V2.1/rcx_vol',
            'arm-none-eabi/*/rcX_V2.1/rcx_netx51',
            'arm-none-eabi/*/rcX_V2.1/rcx',
        ],
        linkerscript    = "netX51_sdram.ld",
        BOOTBLOCKERFLAGS = [ (2)
            '-ct', 'netx51_52',
            '-s', 'SPI_gen_10',
            '-d', 'SD_MT48LC2M32B2-7IT',
        ],
    )
```

(1) Name of the firmware file. The extension ".nxf" requests building a loadable NXF firmware. The executable will be called myfirmware.elf

(2) Boot-Image creation flags for use by ROM loader. These values are usually not evaluated by Second Stage Loader.

A loadable NXF firmware contains a boot header identical to a bootable firmware image. Therefore the integrated bootblocker tool must be is also run and the BOOTBLOCKERFLAGS argument must be specified.

### 3.8.2 Loadable firmware (nxi) for netX 90 / netX 4000

The loadable firmware for netX 90 / netX 4000 is flashed into internal or external Flash memories of netX 90 / netX 4000. It is directly booted by the ROM code of these chipsets and needs to obey a defined layout.

Such loadable firmware file is identified by file extension “.nxi”. The following listing shows how a loadable firmware file for netX90 can be generated:

**Example 11. Generate NXI firmware**

```
def build(bld)
    bld.firmware(
        target      = 'myfirmware.nxi', (1)
```

```
    platform    = 'netx90',
    toolchain   = "gccarmemb",
    description = 'My loadable firmware for netX90',
    source      = [
       "Sources/main.c",
    ],
    use= [
       'armvem-none-eabi/*/eCos/mylib',
       'armvem-none-eabi/*/eCos/myapp',
       'armvem-none-eabi/*/eCos/ecos_sdk',
       'armvem-none-eabi/*/eCos/ecos_target',
    ],
    linkerscript = "netx90_ecos_usecase_a.ld",
    netx_type    = 'NETX90', (2)
    hboot_xml    = 'netx90_ecos_iflash.xml', (3)
 )
```

(1) Name of the firmware file. The extension ".nxi" requests building a loadable NXI firmware. The executable will be called myfirmware.elf

(2) Parameter for hboot compiler to select chipset type

(3) Hboot layout definition for NXI

The loadable NXI firmware for netX 90 / netX 4000 are hboot compatible structures which are generated by the hboot compiler. This is performed as part of the build process and requires specification of arguments *netx_type* and *hboot_xml*. The HBOOT xml layout file must mate with the linkerscript.

### 3.8.3 Loadable firmware (nxi + nxe) for netX 90

The loadable NXI/NXE firmware for netX 90 is flashed into the internal and external flash memories of netX 90. It is directly booted by the chipsets ROM code and needs therefore obey a defined layout.

This loadable firmware is made of two files, a file with extension ".nxi" to be flashed to internal Flash memory and a file ".nxe" to be flashed to external Flash memory. The following example shows how such a firmware can be built

**Example 12. Generate NXI/NXE firmware**

```
def build(bld):
   bld.firmware(
      target      = 'myfirmware.nxi', (1)
      platform    = 'netx90',
      toolchain   = "gccarmemb",
      description = 'My loadable firmware for netX90',
      source      = [
         "Sources/main.c",
      ],
      use= [
         'armv7em-none-eabi/*/eCos/mylib',
         'armv7em-none-eabi/*/eCos/myapp',
         'armv7em-none-eabi/*/eCos/ecos_sdk',
         'armv7em-none-eabi/*/eCos/ecos_target'
      ],
      linkerscript    = "netx90.ld",
      netx_type  = 'NETX90',          (2)
      hboot_xml  = ['netx90_nxi.xml',   (3)
                    'netx90_nxe.xml',   (4)
      ]
   )
```

(1) Name of the firmware file. The extension ".nxi" requests building a loadable NXI firmware. The executable will be called myfirmware.elf

(2) Parameter for hboot compiler to select chipset type

3 Hboot layout definition for NXI

4 Hboot layout definition for NXE. Setting this parameter requests build a NXI/NXE firmware.

The loadable NXI firmware for netX 90 / netX 4000 are hboot compatible structures which are generated by the hboot compiler. This is performed as part of the build process and requires specification of arguments *netx_type* and *hboot_xml*. The HBOOT xml layout files must mate with each other and with the linkerscript.

### 3.8.4 Maintenance firmware (mxf) for netX90

Building a maintenance firmware is identical to building an nxi file as described in section Loadable firmware (nxi) for netX 90 / netX 4000 but using the ".mxf" file extension instead.

### 3.8.5 Application firmware (nai + nae) for netX 90

The NAI/NAE firmware is run on the application side of the netX 90. While the `.nai` file is flashed into internal flash (INTFLASH2) the `.nae` file is optionally written to external flash of a netX 90-based hardware. On booting, the chipset's rom loader will relocate the content of the external Flash to SDRAM memory while the internal Flash content will be used without change. Therefore the part of the firmware located in internal flash needs to be linked with proper addresses and offsets while the part of the firmware located in external flash must be linked to SDRAM memory.

The start offset of the ".nae" image within the external flash memory can be configured in order to partition the external flash memory as needed.

The following listing shows how to build a NAI firmware with Hilscher Waf based build system.

**Example 13. Generate NAI/NAE firmwware**

```
def build(bld):
    bld.firmware(
        target      = 'appfw.nai', (1)
        platform    = 'netx90',
        toolchain   = "gccarmemb",
        description = 'My application firmware for netX90',
        source      = [
            "Sources/main.c",
        ],
        use= [
            'armv7em-none-eabi/*/mylib',
            'armv7em-none-eabi/*/myapp',
        ],
        linkerscript = "netx90_app.ld",
        netx_type    = 'netx90_rev1', (2)
    )
```

1 Name of the firmware file. The extension ".nai" requests building a NAI firmware. The executable will be called appfw.elf

2 Parameter for hboot compiler to select chipset type

For building a combined NAI/NAE firmware, additional parameters must be specified A detailed description of these arguments are provided in section [section-commands-generate_netx90_app_image].

### 3.8.6 Loadable module (nxo) for netX 100 / netX 500

A loadable module is a dynamically linked firmware module which is loaded and executed by the operating system running on the netX. In order to use a loadable module, the netX must be running a base firmware containing the operating system which in turn can load, link and execute the firmware module.

Generation of a loadable module is a more complex process which requires various information and a special prepared linker file. The following example provides a brief introduction how to generate a loadable module.

**Example 14. Generate NXO module**

```
def build(bld):
   bld.module(
      target      = 'mymodule.nxo', (1)
      platform    = 'netx100',
      toolchain   = "codesourcery",
      description = 'My loadable firmware for netX51',
      source      = [
         "Sources/config.c",
      ],
      fileheader_source = 'Sources/FileHeader.c',
      taglist_source    = 'Sources/Taglist.c',
      use= [
         'arm-none-eabi/*/rcX_V2.1/mylib',
         'arm-none-eabi/*/rcX_V2.1/myapp',
         'arm-none-eabi/*/rcX_V2.1/rcx_module_netx100',
         'arm-none-eabi/*/rcX_V2.1/c_module',
      ],
      linkerscript    = "nxo.ld",
   )
```

(1) Name of the module file. The extension ".nxo" requests building a loadable module. The executable will be called mymodule.elf

**NOTE** Loadable modules are usually not linked with default operating system libraries. Instead special module libraries must be used. Due to the recursive nature of the *use* argument it might still happen that default operating system libraries are used in linking step. This will break the linking step of a loadable module. Such a problem arises if a static library referenced by the module depends on an operating system library instead of the operating systems SDK. Therefore it is strongly recommended that static libraries never depend on other static libraries but only on their associated SDK task generators. This avoids such linking issues

### 3.8.7 Bootable firmware images for netX 10 / netX 50 / netX 51 / netX 52 / netX 100 / netX 500

**WARNING** This procedure is considered a low level build step. Using the `bld.firmware()` command defined above strongly recommended.

A bootable firmware image is loaded by loader implemented in netX ROM code. It is typically loaded from serial flash but can also loaded from MMC card formatted with FAT file system, the parallel flash, the DPM or other supported boot sources of the netX.

**NOTE** To boot from an MMC card, the boot able firmware must be placed as file `netx.rom` in the root directory of the card.

Historically the bootable firmware image was created by a tool called bootblocker. This functionality has been fully integrated into Hilscher Waf based build system itself. The name of the command and its arguments are still based on the historic roots of this tool. The bootimage itself is created from a previously build elf image.

**WARNING** In order to build a working boot image from an executable file, the executable must obey some rules. These have impact on the linker script.

The following listing shows an example how to generate a bootable firmware image for an NX-IO board. Details about bootimage/bootblocker arguments can be found in bootblocker documentation.

**Example 15. Generate bootable firmware image**

```
def build(bld):
   bld.bootimage(
      target = 'netx.rom', (1)
      description = 'Boot image for NX-IO',
      use         = ['arm-none-eabi/*/rcX_V2.1/myfirmware'], (2)
```

```
    BOOTBLOCKERFLAGS = [ 3
        '-ct', 'netx500',
        '-s', 'SPI_gen_10',
        '-d', 'SD_MT48LC2M32B2',
    ],
)
```

1 Name of the bootable image file to generate

2 Name of a previously used `bld.program()` task generator to build the executable of the firmware

3 Flags passed to bootblocker tool

## 3.9 Build a distribution

The artefacts build in a project can be deployed in a structured manner within distribution directory by invoking waf with `install` command. By default the `dist` subdirectory of the project is used as destination for distribution output. The location of the distribution directory can be changed with command line arguments when invoking Waf build system.

In order to assists with generation of the project distribution, special build functions have been defined according to Hilscher definitions for distribution folders. These distribution commands are normally part of the top-level wscript file of the project but could be added to any wscript as well. Some usecases of them are described in the following sections

### 3.9.1 Make a pre-compiled library distribution

Pre-compiled library distributions are used when project artefact's like static libraries and public header files are to be integrated into other project. For that purpose the artefacts to be distributed are placed in a directory structure and a wscript declaring the artefacts is generated. This procedure can be automated as shown in the following example listing:

**Example 16. Distribute Pre-Compiled libraries**

```
def build(bld):
    bld.distribute_lib(
        install_path = 'Library', 1

        use = [ 2
            'arm-none-eabi/*/rcX_V2.1/mylib_sdk',
            'arm-none-eabi/*/rcX_V2.1/mylib',

            ('arm-none-eabi/*/rcX_V2.1/mylib_secret', None, 'no_debug_info'), 3

            'arm-none-eabi/*/extcom_sdk', 4
            'arm-none-eabi/*/extcom',
        ],
        dist_includes = [ 5
            'MyLib_Public.h',
        ],
    )
```

1 The distribution will be generated in folder *<destdir>/Library*

2 Reference artefacts to distribute. Hilscher Waf based build system will take care to structure them as defined

3 `mylib_secret` will be removed debug information in distribution

4 It is also possible to re-distribute pre-compiled libraries

5 List of header files to distribute. Will be looked up using the include paths of task generators referenced in *use*

### 3.9.2 Make a firmware file distribution

Firmware distributions require a more flexible distribution structure. Typically firmware files need to be placed in a project specific directory structure. This can achieved by using an extend syntax of the use argument as shown in the following example

**Example 17. Generate a firmware distribution**

```
def build(bld):
    bld.distribute_firmware(
        install_path = 'Firmware', (1)

        use = [ (2)
            ('myfirmware.nxf', 'NXIO/netx100/'),
        ],
        dist_includes = [ (3)
            'MyFw_Public.h',
        ],
    )
```

(1) The distribution will be generated in folder *<destdir>/Firmware*.

(2) Reference artefacts to distribute. A tuple must be used where the second parameter specifies the destination and Hilscher Waf based build system will locate the file in *<distdir>/Firmware/<dest>*. Destination can be directory or file.

(3) List of header files to distribute. Will be looked up using the include paths of task generators referenced in *use*

In addition to distributing the firmware files itself, the used-libs firmware meta information file will be copied as well, if the firmware is built with the *libsused* feature.

### 3.9.3 Make a debug information distribution

In addition to a firmware distribution, Hilscher Waf based build system can also generate a distribution with debug information and source code to provide all necessary guts to allow debugging firmwares. This can achieved as shown in following example:

**Example 18. Generate a debug distribution**

```
def build(bld):
    bld.distribute_debug(
        install_path = 'Debug', (1)
        use_debug    = [ (2)
            'myfirmware.nxf',
        ],
        use_source   = [ (3)
            'arm-none-eabi/*/rcX_V2.1/mylib',
            'arm-none-eabi/*/rcX_V2.1/myapp',
        ]
    )
```

(1) The distribution will be generated in folder *<destdir>/Debug*.

(2) List of firmwares to distribute .elf and .map files for. All firmwares will be placed in compressed archive *<destdir>/Debug/Debug.zip*.

(3) List of components to distribute source files for. The source of all referenced components will be placed in compressed archive *<destdir>/Debug/Source.zip* while preserving the directory structure.

To save disk space the files will be placed in compressed archives (ZIP) which reflect the directory structure of the project.

### 3.9.4 Add arbitrary files to distribution

In many case it is required to add additional files like device description files, example source code to a projects distribution. For that purpose the build functions `bld.install_files()` and `bld.install_as()` are available. They can be used

as shown in the following listing

**Example 19. Add arbitrary files to distribution**

```
def configure(conf):
    conf.setenv('')
    conf.env['MY_VERSION'] = '1.2.3.4' (1)

def build(bld):
    bld.install_files(
        "Firmware/DeviceDescription", (2)
        [
            'Misc/MyDevice.xml',
            'Misc/MySecondDevice.xml',
        ],
    )

    my_version = "V1.2.3.4"

    bld.install_as(
        'Docs/ReleaseInfo.docx',
        'Firmware/ReleaseInfo ${MY_VERSION}.docx', (3)
    )
```

(1) Set a variable in all environments containing the project version

(2) The files will be copied to folder *<destdir>/Firmware/DeviceDescription*.

(3) The file will be copied as *<destdir>/Firmware/ReleaseInfo V1.2.3.4.docx*.

If it is required to automatically scan for files to add to a distribution, an ant glob syntax compatible file search command is available. It can be used as follows:

**Example 20. Distribute arbitrary files from a directory**

```
def build(bld):
    device_description_nodes = \ (1)
        bld.path.ant_glob("DeviceDescription/**/*.xml") + \
        bld.path.ant_glob("DeviceDescription/**/*.bmp") + \
        bld.path.ant_glob("DeviceDescription/**/*.ico")

    bld.install_files( (2)
        "Firmware/DeviceDescription",
        device_description_nodes
    )
```

(1) Scan directory *DeviceDescription* and subfolders for all xml, bmp and ico files.

(2) Install all files found in a flat structure in *<destdir>/Firmware/DeviceDescription*.

## 3.10 Preparing a version header file

The Hilscher Waf based build system can parse the version information associated with a task generator / artefact from a header file. The version header file to parse is specified via the `version_include` argument accepted by most of the build functions as follows:

**Example 21. Retrieve version information from header**

```
def build(bld):
    bld.stlib( (1)
        ...
        version_header = 'MyLib_Version.h' (2)
        ...
```

```
    )
```

1. `stlib()` uses only for reference here, can be applied to any build function
2. Set *task generator's* version from *MyLib_Version.h*

The version header will be looked up in all include directories defined by the *task generator* itself and in all include directories exported by *task generators* used/referenced by means of *use* argument. The version header file will be parsed for preprocessor defines looking like common Hilscher version defines and major, minor build and revision number are extracted accordingly. From this information the `version` argument of the *task generator* is set using the format *<major>.<minor>.<build>.<revision>*. A template listing for parseable version header format is shown in the following example.

**Example 22. Template version header**

```
#ifndef XXX_VERSION_H_
#define XXX_VERSION_H_

#define XXX_VERSION_MAJOR    1 (1)
#define XXX_VERSION_MINOR    1
#define XXX_VERSION_BUILD    5
#define XXX_VERSION_REVISION 0

#endif /* XXX_VERSION_H_ */
```

1. The define ending with `_VERSION_MAJOR` will be considered the major number

### 3.10.1 Use floating point instructions

The netX 90 APP CPU is equipped with a single precision floating point coprocessor. In order to build your code such that it uses the coprocessor the platform must be set to `netx90_app_softfp` or `netx90_app_hardfp` depending on your needs:

- Using platform **netx90_app_softfp** will build code which uses floating point instructions within a function but passes all function parameters as if no floating point coprocessor is available. Such code can be linked with any existing/pre-compiled code which also uses software floating point calling convention.
- In contrast to that **netx90_app_hardfp** will build code which uses floating point instructions within a function and also use the coprocessor registers when passing function parameters. Such code can only be linked with code which has been also build with this calling convention. The code will be more efficient since it can use additional registers. For such targets, the name prefix changes to "armv7em-none-eabihf"

An example listing of how to build your application with hard floating points is shown in the following listing

**Example 23. Build target with full floating point support**

```
def build(bld):
   bld.firmware(
      target      = 'myfirmware.nai',
      platform    = 'netx90_app_hardfp',
      toolchain   = "gccarmemb",
      description = 'My loadable firmware for netX90',
      source      = [
         "Sources/main.c",
      ],
      use= [
         'armv7em-none-eabihf/*/mylib',
         'armv7em-none-eabihf/*/myapp',
      ],
      linkerscript = "netx90_app.ld",
      netx_type    = 'netx90_rev1'
   )
```

## 3.11 Use custom optimization settings

When compiling with *release* or *debugrel* conditions, waf will by default use size optimized compilation setting. In some cases this might not be desired and the target requires other settings.

### 3.11.1 Project wide per toolchain optimization setting

Using function `conf.set_toolchain_optimize_flags()` optimization flags can be set on project level for each toolchain. The function must be used within the configure phase as shown in the following example:

**Example 24. Set optimization settings for entire project**

```
def configure(conf):
   conf.load('hilscher_netx hilscher_firmware hilscher_extras hilscher_dist', tooldir='Waf')
   conf.load('hilscher_toolchain_codesourcery')

   conf.set_toolchain_optimize_flags(
      toolchain = 'codesourcery',
      conditions = 'release', (1)
      cflags='-O3',
      cxxflags='-O3')

   conf.set_toolchain_optimize_flags(
      toolchain = 'codesourcery',
      conditions = 'debugrel',
      cflags='-O3',
      cxxflags='-O3')
```

(1) Set optimization settings for *release* condition.

### 3.11.2 Target specific optimization settings

The optimization settings can also be configured individually per target. This will override the project wide settings. Per target optimization flags are specified using function function `bld.set_target_optimize_flags()` follows. If the target declaration has no *name* attribute, the value of its *target* attribute can be used:

**Example 25. Set optimization settings for a target**

```
def build(bld):
   lib_prefix_rcx21 = bld.get_name_prefix( toolchain = "codesourcery",
      suffix="rcX_V2.1")

   bld.stlib(
      target = "myapp",
      name   = lib_prefix_rcx21 + "myapp",
      ...
   )

   bld.set_target_optimize_flags( (1)
      name = lib_prefix_rcx21 + "myapp", (2)
      conditions = 'release',
      cflags='-O3',
      cxxflags='-O3'
   )

   bld.set_target_optimize_flags(
      name = lib_prefix_rcx21 + "myapp",
      conditions = 'debugrel',
      cflags='-O3',
      cxxflags='-O3'
   )
```

(1) Optimization settings should be configured right after declaring the *task_generator*.

2 Name of the *task generator* to change the optimization settings for..

# Chapter 4 Migration from previous version

The following sections outline changes made to previous waf versions in order to support with migration.

## 4.1 Hilscher Waf based build system V1.13.0.0

### New features

#### WAF-253

Support building targets for *ARMv8-A* architecture using toolchain `ecoscentric` and platform `armv8a` or `netxxxlmpw`.

#### WAF-259

A new distribution command `bld.distribute_debug()` is availabe to distribute executables (.elf) and source code required to debug a firmware. See Make a debug information distribution for more details.

#### WAF-262

The path to *gcov* tool from GNU compiler collection is discovered during toolchain loading and stored in environment variable `GCOV`.

### Changes

#### WAF-257, WAF-258

Object files, libraries and executables are **always** build with **debug symbols enabled**. This is independ of condition *debug*, *debugrel* or *release*. If the debug symbols should be removed from a library for distribution, this has to be explicitly requested in `bld.distribute_lib()` command as explained in section Make a pre-compiled library distribution.

#### WAF-253

In former versions, the toolchains to be loaded had to be declared in the top-level wscript as follows:

```
top = '.'
out = 'build'

REQUIRED_TOOLCHAINS = ['codesourcery', 'ecoscentric']

def options(opt):
    ...
```

This declaration is **no more supported**. Instead a toolchain has to be loaded as intended by Waf build system via explicit loading of the corresponding modules after loading the core modules of Hilscher Waf based build system:

```
top = '.'

def options(opt):
    opt.load('hilscher_netx hilscher_firmware hilscher_extras', tooldir='Tools/Waf')
    opt.load("hilscher_toolchain_codesourcery hilscher_toolchain_ecoscentric")

def configure(conf):
    conf.load('hilscher_netx hilscher_firmware hilscher_extras', tooldir='Tools/Waf')
    opt.load("hilscher_toolchain_codesourcery hilscher_toolchain_ecoscentric")
```

More details about than can be found in section Selecting / loading toolchains.

#### WAF-252

Input & output files are passed to compiler using relative paths to allow for more include paths to be passed / *use* components to be referenced.

# Chapter 5 Command reference

## 5.1 Common arguments

Many of the commands defined by Hilscher Waf based build system have similar arguments. This section describes these common arguments to avoid redundant descriptions afterwards. Refinements to the arguments are described for each command separately within its reference section.

Build functions are used within the `build()` function of a *wscript* file to declare build steps. These commands are small wrapper functions hiding the generation of *task generators* associated with the corresponding build steps. The following common arguments are accepted by all build functions. However, not all arguments might be applicable for a particular build function and thus are ignored silently. For better readability, empty arguments should be omitted

```
def build(bld):
    bld.<cmd>(
        export_includes = ['include_path',...],
        export_defines  = ['define=value',...],
        cflags = ['cflag',...],
        cxxflags = ['cxxflag',...],
        defines = [ 'define=value',...],
        description = 'description_string',
        displaygroup = 'displaygroup_name',
        features = 'feature ...',
        hidden_from_list = True | False,
        includes = ['include_path',...],
        install_path = 'path',
        linkerscript = 'linkerscript_path',
        linkflags = ['linkerflag',...],
        name = 'taskgenerator_name',
        source = [ 'file',...],
        source_arm = [ 'file_arm',...],
        source_thumb = [ 'file_thumb',...],
        platform = 'platform_name',
        target = 'target_path/target_name',
        toolchain    = 'toolchain',
        use = [ 'name',...],
        version_include = 'version_headerfile_name',
    )
```

### cmd

Is the build command. Valid values for cmd are defined in next sections.

### includes = ['include_path',...]

Paths to include folders required to compile the source files are defined using the includes argument. Its value shall be set to a comma separated list of paths to include folders. The paths are interpreted relative to the wscript containing the command.

### cflags = ['cflag',...]

Additional C-Compiler flags required to compile the component can be specified using the cflags argument as comma separated list. Preprocessor-Defines should not be set using this argument. For that purpose the defines argument described below shall be used. Using this argument is not portable and requires knowledge about the used toolchain and compiler as these flags are compiler specific.

### cxxflags = ['cxxflag',...]

This argument is similar to cflags argument described above but for C++ Compiler flags.

### defines = [ 'define=value',...]

Preprocessor-Defines are specified using a comma separated list with the defines argument. The Waf build system will set up proper arguments for assembler, C compiler and C++ compiler depending on used toolchain.

### description = 'description_string'

A human readable string describing the task generators artefact. Used when printing available targets / task

generators when Waf is run using the list command.

### displaygroup = 'displaygroup_name'

A human readable string naming the group this task generator belongs too. Used when printing available targets / task generators when Waf is run using the list command.

### export_includes = ['include_path',...]

List of comma separated paths to include directories which shall be exported to other task generators. If this task generator is referenced by another task generator using the use argument all directories declared with this argument are added to the include path list of referencing task generator. The paths are interpreted relative to the *wscript* containing the definition.

### export_defines = ['define=value',...]

Similar to exporting include directories to other *task generators* it is also possible to propagate pre-processor defines to other *task generators*. All pre-processor defines which shall be set for *task generators* referencing this task generator shall be specified as comma separated list with this argument.

### features= 'feature ...'

The features argument is a space separated string or comma separated list of features that request additional actions when the *task generator* is processed. This argument is extended by the build command wrapper functions with task generator specific entries. (e.g. 'stlib' for static libraries, 'program' for executables, 'c' for c source code) Such builtin values should be used with care since they may break the build. The following values can be used for the features argument:

#### buildstamp

Will add pre-processor defines BUILDSTAMP_BUILD_DATE_YEAR, BUILDSTAMP_BUILD_DATE_MONTH and BUILDSTAMP_BUILD_DATE_DAY which are set to current date

#### group_lib

Will add grouped library linking options to linker command if supported. By default linker uses non grouped linking which requires that input libraries must be in correct order. Otherwise the linker might not be able to resolve symbols. If this feature is specified, the linker command will be modified to use library grouping: the linker will look for a symbol in all library files regardless of the order they appear.

**NOTE** Using this feature can cause unexpected issues with weak symbols and will prevent detection of multiply defined symbols. It is recommended to fix the link library order instead of using this features when unresolved symbols occur in linking step.

#### libsused

Will generate a text file '<artefact_filename>_usedlibs.txt' which contains meta-information about the referenced task generators used when generating the artefact.

#### warninglevel1

Will enable additional compiler code checking flags. Details about this can be found in the toolchain section.

### hidden_from_list = True|False

When this argument is set to true the *task generator* will not be printed when Waf is invoked with the *list* command. The *task generator* will be also omitted in meta-information files. The argument is useful to hide e.g. debug libraries or sdk components.

### install_path = 'distribution_subdirectory'

If the artefact build by the *task generator* should be part of the projects distribution the install_path argument shall be set to the desired destination path. The path is interpreted as directory relative to to distribution directory. (by default 'dist', so artefact will be placed in '<project_dir>/dist/<distribution_subdirectory><target_file_name>').

**WARNING** It is strongly recommended to use the [section-commands-distribute] to generate a distribution instead of using this argument.

### linkerscript = 'linkerscript_path'

For embedded applications the linker process is controlled by a target and project specific linker script. The linkerscript argument is used to specify the path to this file. It is interpreted relative to current wscript file.

### linkflags = ['linkerflag',...]

Specifies additional flags to be passed directly to the linker command. The argument is not portable and correct linkerflag encoding depends on the used toolchain.

**NOTE** For GCC/Clang based toolchains the Hilscher Waf based build system will use the compiler for linking. Therefore flags to be passed to the linker must be prefixed with `-Wl,`.

### name = 'taskgenerator_name'

The name of a task generator is either equal to target_name or explicitly set using this argument. This name is used in other task generator's use argument to refer to the task generator, e.g. include the library when linking or to add include paths.

**NOTE** The name of a task generator is a global symbol in Waf. It is not possible to define multiple task generators with the same name. The last defined task generator with equal name will replace previous ones. This must be considered especially when the name argument is omitted and the target name is used implictly. (E.g. when building a `netx.rom`)

### platform = 'platform_name'

The platform argument is used to choose the correct compiler flags for the desired target platform. Additionaly the pre-processor define `__<platform_name>` will be set in order to detect the target type in source code files. Valid values for platform depend on the toolchain. Currently the followinf platforms are supported:

#### netx

generic ARM926/966 compatible with netX10, netX50, netX51 netX52, netX100 and netX500. Cn be used in conjunction with hitex, codesourcery or gccarmemb toolchain.

#### netx50, netx100, netx500

Can be used in conjunction with hitex, codesourcery or gccarmemb toolchain

#### netx10, netx51, netx52

Can be used in conjunction with codesourcery or gccarmemb toolchain

#### netx90, netx90_app

Can be used in conjunction with gccarmemb toolchain or ecoscentric toolchain. These platform are using software floating point

#### netx90_app_softfp

Can be used in conjunction with gccarmemb toolchain. This platform uses hardware floating point operations but software floating point calling convention. (Compatible with existing precompiled libs using software floating point

#### netx90_app_hardfp

Can be used in conjunction with gccarmemb toolchain. This platform uses hardware floating point operations and floating point register calling convention. This is incomptible with software floating point calling convention.

#### xpic, xpic2

Can be used in conjunction with llvm-xpic toolchain

For detailed information about the compiler flags used for each toolchain and platform please refer to section Supported toolchains

**NOTE** Binary code generated by codesourcery and gccarmemb toolchain using the same platform can be linked together as both toolchains generate binary code compatible with ARM EABI. Hitex toolchain binary code cannot be linked with binary code generated by other toolchains due to different ABIs.

**NOTE** All code using floating point must be compiled using the same calling convention. (ABI) Either with hardware floating point calling convention (*_hardfp) or the other ones. Otherwise linking will fail.

### source, source_thumb, source_arm = [ 'file_path',...]

The source arguments are used to define the source files needed to build the target. The value is a comma separated list of paths to the source files. Paths are interpreted relative to the wscript file containing the command. The source parameters have been adapted to the requirements of ARM processors: Source files specified using parameters

source or source_thumb will be compiled using thumb mode while source files specified using parameter source_arm are compiled in ARM mode. When the target architecture does not support thumb or arm mode all parameters are equivalent. It is recommended to use the suffixed arguments only in case of special requirements. (e.g. arm interrupt handling) All source arguments can be used at the same time to compile different source files of same target in different modes.

### target = 'target_path/target_name'

The target argument sets the name of the artefact defined by the task generator. The target_path if exists is a slash separated subdirectory path to place the target file within. This is e.g. required to generate multiple targets from same wscript file when the name would lead to a filename clash

Depending on the task generator type and the toolchain the target_file_name of the artefact is derived from the target_name value as follows: (inconclusive list)

- For static libraries 'lib<target_name>.a'
- For dynamic libraries: 'lib<target_name>.so'
- For linked firmware executables: '<target_name>.elf'
- For linked applications on windows: '<target_name>.exe'
- For linked applications on linux: '<target_name>'
- For bootable (bootblocker, nxf) netx firmwares: '<target_name>'

Details are described in reference of each build command.

### toolchain = 'toolchain'

The toolchain argument is used to specify the compiler toolchain to use. The argument is mandatory. See table Toolchains provided by Hilscher Waf based build system for a list of supported toolchains.

### use = ['name',...]

References or dependencies to other task generators are specified using the *use* argument. The argument is a list of names of the task generators required by the build command. Dependencies and inputs to tasks will be added according to relationships defined by this argument:

- Include paths exported by listed task generators using the *export_includes* argument are added to include search paths
- Static or dynamic libraries build by referenced task generators are recursively added to linker library list recursively. That means if referenced library references other libraries itself these are also added. Order of linking libraries is prepared according to interdependencies between all referenced libraries. If no direct library interdependencies are known to Waf (typically if SDK task generators are used) the order of libraries when invoking the linker command is determined by the order of the items in the *use* argument. Thus reordering the values can be used to setup correct library order for linker.

The use arguments are processed recursively. Thus variables or artefacts exported from task generators that are referenced from a referenced task generator are also added.

**NOTE** By default Hilscher task generator names for libraries follow the scheme 'toolchain_prefix/toolchain_version/library_name'. In order to support linking independent from knowledge of the version of the toolchain, the use argument accepts a wildcard reference 'toolchain_prefix/*/library_name' if only a single '<toolchain_prefix>/<toolchain_version>' variant of the generator is available within the project.

### version_include = 'version_headerfile_name'

When generating the project distribution folder the Hilscher Waf based build system requires knowledge about the (software) version of the generated artefact. This information is used to prepare various meta-information files. The version can be either specified using the `version` argument or or defined in a header file using a specific format being parsed by waf. The build system will parse the header file to extract the version information using the regular expression:

... ^#define\s+\w+*VERSION*(MAJOR|MINOR|BUILD|REVISION)\s+(\S.*)$ ...

This expression will match the following version header file content: (Everything before *VERSION** is ignored)

```
#ifndef MY_VERSION_H_
#define MY_VERSION_H_
```

```
#define MY_VERSION_MAJOR    4
#define MY_VERSION_MINOR    3
#define MY_VERSION_BUILD    1
#define MY_VERSION_REVISION 0

#endif /* MY_VERSION_H_ */
```

A version number according to Hilscher protocol software version scheme '<major>.<minor>.<build>.<revision>' is constructed. (In this example the result would be '4.3.1.0')

The version header file will be looked up in all include paths including include paths from task generators referenced via use argument. Thus a firmware task generator can use the application library version header file to get its version information from.

## 5.2 General commands

### 5.2.1 <conf|bld>.path.ant_glob - make list of files using pattern search

#### Synopsis

```
def configure(conf):
    nodes = conf.path.ant_glob(['pattern',...])

def build(bld):
    nodes = bld.path.ant_glob(['pattern',...])
```

#### Description

The ant_glob function is used to recursively match all files and subdirectories within a directory with a pattern and return a list of matching nodes. The function is invoked using on an existing (directory) node object. The *path* attribute of the *conf* or *bld* object is the directory node containing the current wscript. Combination of this can be used to generate a list of files and/or directories matching a pattern .

**nodes =**

The function returns a list of Waf node objects representing all files and subdirectories matching any of the specified patterns.

**['pattern', ...]**

This mandatory argument is a list of ant patterns. Patterns are to be specified using the ant file set syntax:

- Path components like directory names and file names are separated by a slash (*/*).
- A single star (*) matches everything except multiple path components. E.g. the pattern * does not match on 'directory_a/file_b' while the pattern */* does
- A double star (**) matches an arbitrary number of directories. E.g. **/* will match everything within the directory

### 5.2.2 <opt|conf|bld>.autorecurse - recursively process subdirectories

#### Synopsis

```
def options(opt):
    opt.autorecurse(
        mandatory = True|False
        excl      = [ 'exclude_path',...]
    )

def configure(conf):
    conf.autorecurse(
        mandatory = True|False
        excl      = [ 'exclude_path',...]
    )
```

```
def build(bld):
    bld.autorecurse(
        mandatory = True|False
        excl      = [ 'exclude_path',...]
    )
```

### Description

The Waf build system supports modular project structures where each module/part of the project is described using an own wscript associated with that component. The build system will not automatically scan the project directory for additional wscript files. The `autorecurse` command is used to recursively search and load all found wscript files in all subdirectories.

> **NOTE** Automatic recursing will load and process all wscript files found. If a found wscript file contains the autorecurse or recurse command by itself, other wscript files might be processed multiple times with unexpected results.

#### excl = [ 'exclude_path',...]

Directories specified by the excl argument are not included and not recursively processed when searching for wscript files. Typically this argument is set to value `['build', 'dist']` to avoid inclusion of *wscript* files built during the build or install commands.

#### mandatory = True|False

If this argument is set to `True` or not specified the command will fail if the wscript files does not contain the `options()`, `configure()` or `build()` function. (depending on the function the recurse function was invoked from). If set to `False` `missing` functions will be silently ignored. This argument is typically set to `True` when recursing from within the `options()` or `configure()` function as most wscripts don't define these functions.

### 5.2.3 <opt|conf|bld>.load - load waf extension modules

### Synopsis

```
def options(opt):
    opt.load('extension_module ...', tooldir = 'extension_directory')

def configure(conf):
    conf.load('extension_module ...', tooldir = 'extension_directory')

def build(bld):
    bld.load('extension_module ...', tooldir = 'extension_directory')
```

### Description

Waf build system} can be modified or extended with 'waf tools'. E.g. the Hilscher Waf based build system is such an extension to Waf build system. An extension can add, modify or remove functionality of Waf build system in a very flexible and transparent way. The `load()` command is used to load an extension for use with the Waf build system. An extension must be loaded in a wscript from `options()` and `configure()` function. Loading from `build()` function is not necessary when already loaded in `configure()` function as Waf automatically loads all extensions in build stage which have been previously loaded in configure stage.

#### 'extension_module ...'

This mandatory argument is a space separated string or string list of extension modules to load from the directory specified by the tooldir argument.

#### tooldir = 'extension_directory'

Path to directory where the extension modules are located.

### 5.2.4 <opt|conf|bld>.recurse - load wscript file from subdirectory

### Synopsis

```
def options(opt):
    opt.recurse('subdirectory_path', mandatory = True|False)

def configure(conf):
    conf.recurse('subdirectory_path', mandatory = True|False)

def build(bld):
    bld.recurse('subdirectory_path', mandatory = True|False)
```

### Description

The recurse command is used to load a wscript file from a specific directory. Despite its name, it does not recursively load all wscript files found within the specified directory. See <opt|conf|bld>.autorecurse - recursively process subdirectories command for this purpose.

#### 'subdirectory_path'

This mandatory argument is the path to the directory containing the wscript to load.

#### mandatory=True|False

If this argument is set to `True` or not specified the command will fail if no wscript file is found in the specified directory or if the wscript does not contain the `options()`, `configure()` or `build()`` function. (Depending on the function the recurse function was invoked from). If set to `False` a missing wscript file or function will be silently ignored.

## 5.2.5 bld.get_name_prefix - compute task generator name prefix

### Synopsis

```
def build(bld):
    name_prefix = bld.get_name_prefix(
        toolchain = 'toolchain',
        platform  = 'netx'
        suffix    = 'suffix',
    )

    bld.<cmd>(
        name   = name_prefix + 'name',
        ...
    )
```

### Description

According to internal Hilscher definitions, a task generator name shall be formatted as slash separated string as follows:

```
<target_triple>/<toolchain_version>/<os_identifier>/<name>
```

The os_identifier field is optional. The prefix should reflect the proper toolchain prefix and toolchain version. To assist in this requirement the function `get_name_prefix()` is provided. It returns the proper prefix according to selected toolchain and optional suffix.

#### toolchain = 'toolchain'

The toolchain argument is used to specify the compiler toolchain the prefix shall be computed for. See above for valid values.

#### platform = 'platform'

The platform argument specifies the platform the prefix shall be computed for depending on the toolchain, a different platforms may result in a different target triple.

#### suffix = 'suffix'

The optional suffix argument is used to specify the operating system dependend suffix. If the target does not depend on any operating system, no suffix shall be used.

## 5.3 Configuration commands

### 5.3.1 <opt|conf>.load - Load extension modules

#### Synopsis

```
def opt(opt):
    opt.load(['module_name',...], tooldir = 'path/to/package')

def configure(conf):
    conf.load(['module_name',...], tooldir = 'path/to/package')
```

#### Description

Additional modules are loaded into Waf build system using the `opt.load()` and `conf.load()` command. It is usually used in the project's top-level wscript file to load Hilscher Waf based build system modules and to initialize the toolchains. If a module provides additional command line options, it must be also loaded in `opt()` in order to make the optins available.

**['module_name',...]**

List of strings or whitespace separated string which modules should be load

**tooldir = 'path/to/package'**

Optional argument used to provide the path to the directory where the modules to load are located. If this argument is not provided Waf build system will lookup the module in its internal search paths. When loading Hilscher Waf based build system, the internal seach paths will be extended such that toolchains modules can be loaded without *tooldir* argument.

### 5.3.2 conf.set_toolchain_optimize_flags - set toolchain optimization settings

#### Synopsis

```
def configure(conf):
    conf.set_toolchain_optimize_flags(
        toolchain  = 'toolchain name',
        conditions = 'build_conditions',
        cflags     = ['flag',...],
        cxxflags   = ['flag',...],
        defines    = ['define=value',...],
        linkflags  = ['flag',...],
    )
```

#### Description

**toolchain = 'toolchain name'**

The name of toolchain to set the optimization flag value for. See table Toolchains provided by Hilscher Waf based build system for valid toolchain names.

**conditions = 'build_conditions'**

The build condition to apply the settings to. Must be one of 'debug', 'debugrel', 'release'.

**cflags = ['flagg',...], cxxflags = ['flagg',...], linkflags = ['flagg',...]**

The flags to pass to c-compiler, c++ compiler or linker when building for the specified condition

**defines = ['define=value',...]**

Defines to set in c/c++ compiler when building fpr specified condition

## 5.4 Declaration of headers & external libraries

### 5.4.1 bld.externalcomponent - declare pre-compiled library

#### Synopsis

```
def build(bld):
    bld.externalcomponent(
        description     = 'description_string',
        displaygroup    = 'displaygroup_name',
        export_includes = ['include_path',...],
        name            = 'taskgenerator_name',
        target          = "library_name",
        path            = "path_to_directory",
        use             = [ 'name',...],
        version         = "version_string",
    )
```

#### Description

**name = 'taskgenerator_name'**

The name of the external component task generator is used by other components to include the external component in linking step. The name should conform to standard Hilscher Waf based build system naming scheme for task generators

```
<toolchain_prefix>/<toolchain_version>/<component_name>
```

If the toolchain version is not known, it might be replaced with a star (*).

**path = 'path_to_directory'**

Path to directory where to find the library file associated with this external component. The path is relative to the declaring wscript file.

**target = 'library_name'**

Basename of the library file associated with this external component. The library filename will be derived from this argument as follows:

- For static libraries 'lib<library_name>.a'
- For dynamic libraries: 'lib<library_name>.so' Waf will look for the library file in directory pointed to by argument path.

**version = 'version_string'**

The version argument can be used to specify the version of the external component. The version is expected to be a string made of four numbers separated by a dot (.):

```
<major>.<minor>.<build>.<revision>
```

### 5.4.2 bld.sdkcomponent - declare a SDK

Snippet 3. Synopsis

```
def build(bld):
    bld.sdkcomponent(
        description     = 'description_string',
        displaygroup    = 'displaygroup_name',
        export_includes = ['include_path',...],
        name            = 'taskgenerator_name',
        use             = [ 'name',...],
        version         = "version_string",
        version_include = 'version_headerfile_name',
    )
```

### Description

#### name = 'taskgenerator_name'

The name of the SDK task generator is used by other components to declare a dependency on this SDK. The name should conform to standard Hilscher Waf build system naming scheme for SDK task generators

```
<component_name>_sdk or (1)
<toolchain_prefix>/<toolchain_version>/<component_name>_sdk (2)
```

(1) If possible, sdks should not contain a toolchain/platform dependency,

(2) otherwise the SDK name should follow the common rules.

#### version = 'version_string'

The version argument can be used to specify the version of the SDK task generator. The version is expected to be a string made of four numbers separated by a dot (.):

```
<major>.<minor>.<build>.<revision>
```

## 5.5 Compilation and linking

The functions described in this section are used to compile and linke firmware files and applications from input files and libraries.

### 5.5.1 bld.program - link an application / executable

### Synopsis

```
def build(bld):
    bld.program(
        name           = 'taskgenerator_name',
        target         = 'target_path/target_name',
        description     = 'description_string',
        displaygroup    = 'displaygroup_name',

        toolchain      = 'toolchain',
        platform       = 'platform_name',

        features       = 'feature ...',

        includes       = ['include_path',...],
        defines        = [ 'define=value',...],
        source         = [ 'file',...],
        source_arm      = [ 'file_arm',...],
        source_thumb    = [ 'file_thumb',...],
        use            = [ 'name',...],

        linkerscript    = 'linkerscript_path',

        cflags         = ['cflag',...],
        cxxflags       = ['cxxflag',...],
        linkflags      = ['linkerflag',...],
        hidden_from_list = True | False,
        install_path    = 'path',
        version_include  = 'version_headerfile_name',
    )
```

### Description

The `program()` build function generates a linked application file which can either be loaded into an embedded target using a debugger or executed as application. The command accepts all standard arguments which are described in section Common arguments The command can compile target specific source files and add them to the linking step. In case of embedded targets it is mandatory to specify a linkerscript. This linkerscript is passed to the toolchains linker tool and thus

is toolchain dependent.

### target = 'target_path/target_name'

The target argument defines the base name of the linked object file generated in final linking step. The file name and path will be derived from the argument depending on the toolchain. The following naming rules are defined:

- embedded target toolchains the file name will be `<target_path>/<target_name>.elf`.
- for host target the filename will be according the host scheme, e.g. `<target_path>/<target_name>.exe`

## 5.5.2 bld.stlib - build a static library

### Synopsis

```
def build(bld):
    bld.stlib(
        name           = 'taskgenerator_name',
        target         = 'target_path/target_name',
        description     = 'description_string',
        displaygroup    = 'displaygroup_name',

        toolchain       = 'toolchain',
        platform        = 'platform_name',

        features        = 'feature ...',

        includes        = ['include_path',...],
        export_includes  = ['include_path',...],
        export_defines   = ['define=value',...],
        defines         = [ 'define=value',...],
        source          = [ 'file',...],
        source_arm       = [ 'file_arm',...],
        source_thumb     = [ 'file_thumb',...],

        use            = [ 'name',...],

        cflags          = ['cflag',...],
        cxxflags        = ['cxxflag',...],
        hidden_from_list = True | False,
        install_path     = 'path',
        version_include  = 'version_headerfile_name',
    )
```

### Description

The `stlib()` build functins defines the build of a static library. The functionc accepts all standard arguments which are described in section Common arguments. The library file name is derived from the target name using the following naming rule `lib<target_name>.a`

## 5.5.3 bld.set_target_optimize_flags - set target optimization settings

### Synopsis

```
def build(bld):
    ...
    bld.set_target_optimize_flags(
        name       = 'taskgenerator_name',
        conditions = 'build_conditions',
        cflags     = ['flag',...],
        cxxflags    = ['flag',...],
        defines    = ['define=value',...],
        linkflags  = ['flag',...],
    )
```

#### Description

### name = 'taskgenerator_name'

The name of the task generator to set the flags for. The task generator must be declared in advance using one of the build functions. If the task generator has no name, the value of its *target* attribute can be used instead.

### conditions = 'build_conditions'

The build condition to apply the settings to. Must be one of *debug*, *debugrel* or *release*.

### cflags = ['flagg',...], cxxflags = ['flagg',...], linkflags = ['flagg',...]

The optimization flags to pass to C-compiler, C++ compiler or linker when building for the specified condition

### defines = ['define=value',...]

Defines to set in C/C++ compiler when building for specified condition

## 5.6 Firmware creation commands

### 5.6.1 bld.bootimage - generate a netX bootimage

#### Synopsis

```
def build(bld):
    bld.bootimage(
        name           = 'taskgenerator_name',
        target         = 'target_path/target_name',
        description     = 'description_string',

        use            = [ 'application_name'],
        BOOTBLOCKERFLAGS = [ 'bootblock_arg',...],

        hidden_from_list = True | False,
        install_path = 'path',
    )
```

#### Description

The `bootimage()` build function generates a bootable firmware image for netX10/netX50/netX51/netX52/netX100/netX500 processors. The command was derived from the bootblocker tool. In order to generate a valid bootimage, chip type, source device and destination device parameters must be specified as bootblocker args. The input to this command must be single reference to a program *task generator* specified via the use parameter.

> **NOTE** | In order to generate a valid boot image the input elf file and bootblocker arguments must suit the target.

### BOOTBLOCKERFLAGS

This argument specifies the command line options as would be passed to the Hilscher Bootblocker tool. The options have to be provided as comma separated list. The Hilscher Waf based build system parses this argument and will generate a boot image accordingly. For that purpose the part of Bootblocker which generates the boot image has been integrated into Hilscher Waf based build system. External Bootblocker tool is not required.

### 5.6.2 bld.firmware - generate a loadable firmware

#### Synopsis

```
def build(bld):
    bld.firmware(
        name        = 'taskgenerator_name',
        target      = 'firmware_file_name',
        description  = 'description_string',
        displaygroup = 'displaygroup_name',
```

```
    toolchain    = 'toolchain',
    platform     = 'platform_name',

    features     = 'feature ...',

    includes     = ['include_path',...],
    defines      = ['define=value',...],
    source       = ['file',...],
    source_arm   = ['file_arm',...],
    source_thumb = ['file_thumb',...],

    use         = [ 'name',...],

    linkerscript = 'linkerscript_path',

    netx_type  = 'netx_type_name',

    BOOTBLOCKERFLAGS = [ 'bootblock_arg',...],

    hboot_xml            = 'hboot_xml_file',
    headeraddress_extflash = absolute_address,
    sdram_split_offset     = sdram_application_start,
    segments_intflash      = ['intflash_section1', 'intflash_section2' ...],
    segments_extflash      = ['extflash_section1', 'extflash_section2' ...],
    sniplib              =  'path_to_sniplib',

    cflags      = ['cflag',...],
    cxxflags    = ['cxxflag',...],
    linkflags   = ['linkerflag',...],

    hidden_from_list = True | False,
    install_path     = 'path',
    version_include  = 'version_headerfile_name',
)
```

### Description

The `bld.firmware` build function defines the generation of a loadable firmware for use with second stage loader based netX boot sequence, hboot based boot sequence or application side firmware. The command is an extension of function `bld.program`. It combines this build functions with te actions performed by build functions `bld.bootimage`, `bld.hboot` or `bld.generate_netx90_app_image.` depending on the firmware file extension. I Thus the functions command accepts arguments from all of these firmware creation commands. Only the relevant one have to specified.

In the following only differences in certain arguments are explained. For full reference of all arguments see description of bld.program - link an application / executable, bld.bootimage - generate a netX bootimage, bld.generate_hboot_image - generate a hboot image and bld.generate_netx90_app_image - generate netx90 application image.

Depending on kind of firmware either:

- argument *BOOTBLOCKERFLAGS* (for .nxf firmware) or
- arguments *hboot_xml* and *netx_type* (for .nxi firmware) or
- argument *netx_type* (.nai firmware) must be present.

**NOTE** In order to generate a valid loadable firmware image the input elf file must conform to loadable firmware file elf rules. This includes a proper structure definition of the elf file within the linkerscript and proper hboot file structure definition.

### `target = 'firmware_file_name'`

The target argument sets the name of the loadable firmware file to be generated. The argument defines the full name and must be specified with extension '.nxf', '.nxi', '.mxf' or '.nai'. The command will derive the file name of the linker output from the target argument by replacing its extension with '.elf'. The filename extension will be used to determine the firmware generation command to use.

### 5.6.3 bld.module - generate a loadable module

#### Synopsis

```
def build(bld)
    bld.module(
        name        = 'taskgenerator_name',
        target      = 'nxo_file_name',
        description  = 'description_string',
        displaygroup = 'displaygroup_name',

        toolchain    = 'toolchain',
        platform     = 'platform_name',

        features     = 'feature ...',

        includes     = ['include_path',...],
        defines      = [ 'define=value',...],
        source       = [ 'file',...],
        source_arm   = [ 'file_arm',...],
        source_thumb = [ 'file_thumb',...],

        use         = [ 'name',...],

        linkerscript = 'linkerscript_path',

        fileheader_source = 'fileheader_source_file',
        taglist_source    = 'taglist_source_file',

        cflags    = ['cflag',...],
        cxxflags  = ['cxxflag',...],
        linkflags = ['linkerflag',...],

        install_path     = 'path',
        version_include  = 'version_headerfile_name',
        hidden_from_list = True | False,
    )
```

#### Description

The `bld.module()` functions requests building a loadable module. A loadable module is used with a base firmware containing the operating system. The base firmware will load, dynamically link and execute the loadable module at runtime. Multiple loadable modules can be executed at the same time on a single netX chip and thus allow combination of modules to a custom communication firmware. The build function is dervied from `bld.program()` but performs additional code generation and processing steps. The command accepts the same arguments as `bld.program()` but requires some additional parameters.

**NOTE** In order to generate a valid loadable module linked elf file must conform to loadable module file elf rules. This includes a proper structure definition of the elf file within the linkerscript.

#### `fileheader_source = 'fileheader_source_file'`

A loadable firmware as well as a loadable module contain a file header structure. This structure contains a basic description of the module. For loadable modules the file header source file must be provided using the separate argument fileheader_source.

#### `taglist_source = 'taglist_source_file`

A loadable module contain a taglist. This taglist define the used resources by the loadable module and can be modified within the module file using the taglist editor tool. The source of the taglist file must specified using this argument.

#### `target = 'nxo_file_name'`

The target argument sets the name of the loadable module file to be generated. The argument defines the full name and must be specified with extension *.nxo*. The function will derive the file name of the linker output from the target argument by replacing its extension with '.elf'.

### 5.6.4 bld.generate_hboot_image - generate a hboot image

#### Synopsis

```
def build(bld):
    bld.generate_hboot_image(
        name      = 'taskgenerator_name',
        target    = 'hboot_file_name',

        use       = [ 'name',...],
        hboot_xml  = 'hboot_xml_file',
        netx_type  = 'netx_type_name',
        sniplib    = 'path_to_sniplib',
    )
```

#### Description

The `bld.generate_hboot_image()` functions creates a hboot image from input elf files using the firmware structure defined in the hboot xml file.

**hboot_xml = 'hboot_xml_file'|[ hboot_nxi_xml_file', hboot_nxe_xml_file']**

Path to hboot xml file(s) which describes the structure of the firmware. The elf files passed to this build command by the use argument can be referenced from the hboot xml file using aliases 'tElf0', 'tElf1'... . If a combined internal/external flash firmware shall be build, two xml files must be passed as list

**netx_type = 'netx_type_name'**

This argument is used to specify the target netx type. This information is used by hboot tool to load the proper patch table which describes the rom code of the target chipset. The following netx types are defined:

- `NETX90` is the netX90 rev0 chipset
- `NETX90B` is the netX90 rev1 chipset
- `NETX90_MPW` is the netX90 multi project wafer sample
- `NETX4000` is the netX4000 chipset
- `NETX4000_RELAXED` is the netX4000 relaxed chipset

**target = 'hboot_file_name'**

The target argument defines the output file name to be used for the image. If an nxi+nxe firmware is requested by means of hboot_xml argument, the nxe's filename will be derived internally from target argument by replacing the extension with *.nxe*.

**sniplib = 'path_to_sniplib'**

When the hboot xml file references external snippets, this argument must be used to specify the path to the snippet library.

**use = [ 'name',...]**

The use argument is a list of *bld.program()* task generator names from which the firmware should be build. The entries in this list are enumerated using aliases tElf0, tElf1,... . These aliases must be used in hboot xml file to reference the elf files.

### 5.6.5 bld.generate_netx90_app_image - generate netx90 application image

#### Synopsis

```
def build(bld):
    bld.generate_netx90_app_image(
        name               = 'taskgenerator_name',
        target             = 'nai_firmware_name',

        use                = [ 'name',...],
```

```
    netx_type           = 'netx_type_name',
    headeraddress_extflash = absolute_address,
    sdram_split_offset    = sdram_application_start,
    segments_intflash     = ['intflash_section1', 'intflash_section2' ...],
    segments_extflash     = ['extflash_section1', 'extflash_section2' ...],
)
```

### Description

The `bld.generate_netx90_app_image()` function creates a firmware for the netX90 application side. The firmware files are build using the 'netx90_app_image' tool from hboot tools. The command support building firmwares using only internal flash and firmwares splitted in internal and external flash parts.

**NOTE** The netx90_app_image tool expects an XML file to control the build process. Hilscher Waf based build system uses an internal, fixed version of this file to simplify usage.

**NOTE** {waf-hilscher} expects a proper Hilscher V3 fileheader to be linked into the internal flash and external flash firmware images. Therefore the linkerscript must be written such that the fileheader for NAI and for NAE are placed in proper position in the output image.

## headeraddress_extflash = absolute_address

This argument defines the absolute address of the position of the '.nae' file in external flash. This argument is needed only in case the firmware shall be splitted across internal and external flashes. Absolute address means here the address as seen from netX90 application cpu.

## netx_type = 'netx_type_name'

This argument is used to specify the target netx type. This information is required when booting the netx. The following netx types are defined:

- `netx90_rev0` is the "NULL" revision of the netX90 chipset
- `netx90_rev1` is the "first" revision of the netX90 chipset

## sdram_split_offset = sdram_application_start

This argument defines the start address were the app side SDRAM memory starts within the SDRAM device. The address range 0x0 to (sdram_split_offset - 1) of the SDRAM device will be assigned to the COM firmware while the address range from sdram_split_offset to end of sdram device is assigned to the APP firmware. The following values are allowed:

| SDRAM split offset value | Description | Usage |
|---|---|---|
| 0x0 | Entire SDRAM assigned to APP Side | Use in conjunction with usecase A COM firmware only. |
| 0x00400000 | 8 MB SDRAM with 4MB/4MB Split | Use in conjunction with usecase A or usecase C COM firmware. |
| 0x00800000 | 16 MB SDRAM with 8MB/8MB Split | |
| 0x01000000 | 32 MB SDRAM with 16MB/16MB Split | |
| 0x02000000 | 64 MB SDRAM with 32MB/32MB Split | |
| 0x04000000 | 128 MB SDRAM with 64MB/64MB Split | |

Table 4. Supported values for SDRAM Split offset

## segments_intflash = [ 'intflash_section1', 'intflash_section2'... ]

This argument is needed only in case a firmware splitted across intflash and extflash is to be built. It is a list of section names and defines which sections should be placed in the internal flash ('.nai' file)

## segments_extflash = [ 'extflash_section1', 'extflash_section2'... ]

This argument is needed only in case a firmware splitted across intflash and extflash is to be built. It is a list of section names and defines which sections should be placed in the external flash ('.nae' file).

## target = 'nai_firmware_name'

The target argument defines the output file name to be used for the image. It is expected to have extension '.nai'. If a splitted internal/external flash firmware is build, waf will derive the file name of the '.nae' file from this parameter by replacing the extension with '.nai'

### use = [ 'name',...]

The use argument is a list of bld.program task generator names from which the firmware should be build. The entries in this list are enumerated using aliases tElf0, tElf1,... . These aliases must be used in hboot xml file to reference the elf files.

## 5.7 Distribution commands

### 5.7.1 bld.distribute_lib - generate a library distribution

#### Synopsis

```
def build(bld):
   bld.distribute_lib(
      install_path = 'subdirectory',
      use         = [
         'name',
         ('another_name', None, 'no_debug_info'),
         ...
      ],
      use_ltd     = [ 'name',...],
      use_doc     = [ ('name', 'destination'),...],
      dist_includes = [
         'include_name' ,
         ('include_name','destination_subdir'),
         ...
      ]
   )
```

#### Description

The `bld.distribute_lib()` function generates a library distribution folder structure as specified by internal definitions of Hilscher. The command arranges the files according to this definition. Also a wscript file will be generated.

### install_path = 'subdirectory'

The mandatory install_path argument defines the destination subdirectory path relative to distribution directory. E.g using value 'Library' would place all files to distributed in subdirectory 'dist/LibrarySVN'.

### use = [ 'name',...]

The use argument is a list of library task generators to include into this library distribution. Waf will collect all library files and place them in a defined directory structure in a subdirectory named *Lib*.

The syntax of an list item of the `use` argument can be either:

- A string `'name'` to just refer to the library
- A tuple `('name', None, 'flags')` to refer to the library and pass additional flags as whitespace separated string. Currently the following flags are known
  - `no_debug_info` means that any debug information should be removed in the distribution

### use_ltd = [ 'name',...]

The use argument is a list of library task generators to include into this library distribution. Waf will collect all library files and place them in a defined directory structure in a subdirectory named *Lib_Ltd*.

### use_doc = [ ('name', 'destination'),...]

The `use_doc` argument can be used to include documentation files into the distribution. Documentation files will be placed in *Documentation* subdirectory. The `destination` parameter can be directory or full path and defines where the documentation should be put to.

### dist_includes [ 'include_name', ('include_name','destination_subdir')...]

The dist includes argument can be used to specify header files to be added to the *Includes* subfolder. Waf will automatically search all include search paths of the referenced librarys to locate the include file. The argument

understands two different syntax forms:

- `'include_name'` will put the include folder as is and preserve subdirectories. E.g. 'PNSIF_API.h', 'lwip/ipv4.h' will become 'Includes/PNSIF_API.h', 'Includes/lwip/ipv4.h'
- `('Include_name', 'destination_subdir')` will put the include file into the given subdirectory path

### 5.7.2 bld.distribute_firmware - generate a library distribution

#### Synopsis

```
def build(bld):
    bld.distribute_firmware(
        install_path = 'subdirectory',
        use         = [ ('name', 'destination'),...],
        use_ltd     = [ ('name', 'destination'),...],
        use_doc     = [ ('name', 'destination'),...],
        dist_includes = [
            'include_name' ,
            ('include_name','destination_subdir'),...
        ]
    )
```

#### Description

The `bld.distribute_firmware()`function generates a firmware distribution folder structure as specified by internal definitions of Hilscher. The command aranges the files according to this definition.

#### `install_path = 'subdirectory'`

The mandatory install_path argument defines the destination subdirectory path relative to distribution directory. E.g using value 'Firmware' would place all files to distributed in subdirectory 'dist/Firmware'

#### `use = [ ('name', 'destination'),...]`

The use argument is a list of firmware *task generators* to be included in the firmware distribution. Each entry consists of two values:

- the `'name'` value is the name of the firmware's task generator
- the `'destination'` value specifies the location were the file should be put relative to the `install_path`. The location can be:
  - either a directory indicated by a trailing slash or
  - a path including the destination file name.

**Example 26. Destination in firmware distributions**

```
def build(bld):
    bld.distribute_firmware(
        install_path = 'MyFirmware',
        use         = [
            ('X060D000.nxf','netX/netX51/'), (1)
            ('netx_nxhx51.rom'  ,'Tests/NXHX51/netx.rom') (2)
        ],
    )
```

(1) *X060D000.nxf* is copied to *dist/MyFirmware/netX/netX51/X060D000.nxf*

(2) *netx_nxhx51.rom* is copied to *dist/MyFirmware/Tests/NXHX51/netx.rom*

#### `use_ltd = [ ('name', 'destination'),...]`

The use_ltd argument is a list of firmware task generators to be included in the firmware distribution as limited firmware. Firmwares will be placed within *Firmware_Ltd* subfolder. It has same syntax as 'use' argument above.

### use_doc = [ ('name', 'destination'),...]

The `use_doc` argument can be used to include documentation files into the distribution. Documentation files will be placed in *Documentation* subfolder. It has same syntax as 'use' argument above.

### dist_includes [ 'include_name', ('include_name','destination_subdir')...]

The dist includes argument can be used to specify header files to be added to the *Includes* subfolder. Waf will automatically search all include search paths of the referenced librarys to locate the include file. The argument understands two different syntax forms:

- `'include_name'` will put the include folder as is and preserve subdirectories. E.g. 'PNSIF_API.h', 'lwip/ipv4.h' will become 'Includes/PNSIF_API.h', 'Includes/lwip/ipv4.h'
- `('Include_name', 'destination_subdir')` will put the include file into the given subdirectory path

## 5.7.3 bld.distribute_debug - generate a debug distribution

### Synopsis

```
def build(bld):
   bld.distribute_debug(
      install_path = 'subdirectory',
      use_debug    = [ 'name',...],
      use_source   = [ 'name',...],
      use_doc      = [ ('name', 'destination'),...],
   )
```

### Description

The `bld.distribute_debug()`function generates a debug distribution folder structure as specified by internal definitions of Hilscher. Such a distribution contains linked executables, map files an source code.

### install_path = 'subdirectory'

The mandatory install_path argument defines the destination subdirectory path relative to distribution directory. E.g using value 'Debug' would place all files to distributed in subdirectory 'dist/Debug'

### use_debug = [ 'name', ,...]

The use argument is a list of firmware *task generators* to be included in the debug distribution. For each entry, waf will get the linked executable and the map file and arrange them in the same structure as in the project. All files will be place in a single zip file named *Debug.zip* to safe disk space.

### use_source = [ 'name', ,...]

The use argument is a list of *task generators* for which the source code files should be added to the debug distribution. For each entry, waf will get all source and header files and arrange them in the same structure as in the project. All files will be place in a single zip file named *Source.zip* to safe disk space.

### use_doc = [ ('name', 'destination'),...]

The `use_doc` argument can be used to include documentation files into the distribution. Documentation files will be placed in *Documentation* subfolder. It has same syntax as 'use' argument in command `bld.distribute_firmware()` above.

## 5.7.4 bld.install_as - copy file to distribution with renaming

### Synopsis

```
def build(bld):
   bld.install_as('destination_file_path', 'source_file_path')
```

### Description

The `bld.install_as()` command is a standard Waf build system command which copies a file to the distribution and optionally changes its name.

#### 'destination_file_path'

Destination file path relative to distribution directory.

#### 'source_file_path'

Path to source file relative to current wscript

## 5.7.5 bld.install_files - copy multiple files to distribution

### Synopsis

```
def build(bld):
    bld.install_files('destination_dir_path', [ 'source_file_path', ...])
```

### Description

The `bld.install_files()` command is a standard Waf build system command which copies mutliple filed to the distribution

#### 'destination_dir_path'

Path to the destination directory where to place the files. The path is relative to distribution directory.

#### ['source_file_path', ...]

List of paths of files to copy. Paths are relative to current wscript.

## 5.8 Miscellanous commands

## 5.8.1 bld.generate_doxygen_documentation - generate and distribute doxygen documentation

### Synopsis

```
def options(opt): (1)
    opt.load('hilscher_doc', path="Tools/Waf")

def configure(conf): (1)
    conf.load('hilscher_doc', path="Tools/Waf")

def build(bld):
    bld.generate_doxygen_documentation(
        doxyfile = 'doxyfile_path'
    )
```

(1) *hilscher_doc* module must be loaded in top-level wscript

### Description

**NOTE** It is strongly recommended to use asciidoc based documentation. This feature is provided by a separate Hilscher Waf based build system extension module. However, asciidoc may use doxygen parse output

Hilscher Waf based build system can run the doxygen tool to generate a doxygen based documentation. The documentation is build when Hilscher Waf based build system was invoked with the *doc* or *install* command. In order to use this command the *hilscher_doc* extension must be loaded in top-level wscript functions `options()` and `configure()`.

**IMPORTANT** The output directory configured in the Doxyfile (*OUTPUT_DIRECTORY*) must be set to a path relative to doxyfile, pointing to a subfolder within the project's 'build' folder. This is required by waf build system.

### 'doxyfile_path'

Path to the doxygen configuration file to use when invoking doxygen

# Chapter 6 Supported toolchains

This sections describes the toolchains supported by the Hilscher Waf based build system and the arguments used with them. However, since Waf build system combines arguments and options from various places prior to invoking a tool like an compiler, arguments might differ from the default settings. In order to see the final arguments passed to an external tool, the command line argument `--dump-environment=variables,...` can be used when invoking waf.

**NOTE** The Hilscher Waf based build system might accept other toolchain names not listed here. These toolchains are implemented for internal evaluation purposes only. They are officially not supported and shall not be used. Support of these toolchains is incomplete, untested and might be dropped silently.

## 6.1 GCC based toolchains

### 6.1.1 Common definitions and parameters

#### 6.1.1.1 C / C++ Source files

When the Hilscher Waf based build system compiles C source files into object files the `gcc` command is used while C++ source files are compiled using `g++` command.

**Flags passed to GCC Compiler**

```
-g
-Wall
-Wredundant-decls
-Wno-inline
-Winit-self
```

**Flags passed to GCC Compiler for condition debug**

```
-O0
```

**Flags passed to GCC Compiler for condition debugrel & release**

```
-Os
```

**Additional flags passed to GCC compiler when building for netx platforms**

```
-gdwarf-2
-mlong-calls
-mapcs
-fno-common
-mthumb-interwork (1)
-D_NETX_
-D__NETX10 / -D__NETX50 / -D__NETX51 / -D__NETX52 / -D__NETX90 / -D__NETX100 / -D__NETX500 (2)
```

(1) Only if Platform supports ARM & Thumb instruction sets

(2) Depends on actual used platform

**Additional flags passed to GCC compiler when using feature *warninglevel1***

```
-Wsystem-headers
-Wbad-function-cast
-Wsign-compare
-Wswitch-default
-Wstrict-prototypes
-Wpointer-arith
```

#### 6.1.1.2 Assembler source files

Assembler source files are processed with the `gcc` or `as` command depending on their extension. In order to run the C preprocessor on assembler source files before running the assembler, the file extension must be an uppercase `.S`. A lower case `.s` means not to run the preprocessor.

**Flags passed to assembler**

```
-Wa,-g
-Wall
-Wredundant-decls
-Wno-inline
```

**Additional flags passed to assembler when building for netx platforms**

```
-mapcs
```

#### 6.1.1.3 Linker

The linker will be invoked via the `gcc` tool.

**Flags passes to linker command**

```
-mthumb-interwork (1)
-nostdlib
```

(1) Only if Platform supports ARM & Thumb instruction sets

### 6.1.2 Hitex toolchain

The Hitex toolchain is based on GCC V4.0.3 and supports code generation for ARMv5 architecture compatible CPUs. It is identified by toolchain name “hitex”. The code generated by this toolchain uses a pre EABI binary format and thus its binary output can only be linked with other code generated by Hitex toolchain.

**Additional flags passed to GCC Compiler when using Hitx Toolchain**

```
-fshort-enums

-mfloat-abi=soft
-msoft-float

-march=armv5te   -mfpu=vfp (1)
-mcpu=arm9e      -mfpu=vfp (2)
-mcpu=arm926ej-s -mfpu=vfp (3)
```

(1) When building for platform "netx"

(2) When building for platform "netx50"

(3) When building for platform "netx100" or "netx500"

### 6.1.3 CodeSourcery toolchain

The CodeSourcery toolchain is based on GCC V4.5.2 and supports code generation for ARMv5 and ARMv6 architecture compatible CPUs. It is identified by toolchain name "codesourcery". The code generated by this toolchain is ARM EABI compatible and can be linked with any other binary conforming to ARM EABI.

**Additional flags passed to GCC Compiler when using CodeSourcery Toolchain**

```
-fshort-enums
-ffunction-sections (1)
-fdata-sections (1)

-mfloat-abi=soft
-msoft-float

-march=armv5te   -mfpu=vfp (2)
-mcpu=arm966e-s  -mfpu=vfp (3)
-mcpu=arm926ej-s -mfpu=vfp (4)
```

(1) Can be suppressed with "disable_gc_sections" feature.

(2) When building for platform "netx"

3 When building for platform "netx10", "netx50", "netx51" or "netx52"

4 When building for platform "netx100" or "netx500"

**Additional flags passed to GCC Linker when using CodeSourcery Toolchain**

```
-Wl,-gc-sections
```

### 6.1.4 GNU ARM Embedded toolchain

The GNU ARM Embedded toolchain is based on GCC V4.9.3 and supports code generation for ARMv5, ARMv6 and ARMv7 architecture compatible CPUs. It is identified by toolchain name "gccarmemb". The code generated by this toolchain is ARM EABI compatible and can be linked with any other binary conforming to ARM EABI.

**Additional flags passed to GCC Compiler when using GNU ARM Embedded Toolchain**

```
-fshort-enums
-ffunction-sections (1)
-fdata-sections (1)

-march=armv5te   -mfpu=vfp -mfloat-abi=soft -msoft-float (2)
-mcpu=arm966e-s  -mfpu=vfp -mfloat-abi=soft -msoft-float (3)
-mcpu=arm926ej-s -mfpu=vfp -mfloat-abi=soft -msoft-float (4)
-march=armv7e-m  -mfloat-abi=soft                        (5)
-march=armv7e-m  -mfpu=fpv4-sp-d16 -mfloat-abi=softfp    (6)
-march=armv7e-m  -mfpu=fpv4-sp-d16 -mfloat-abi=hardfp    (7)
```

1 Can be suppressed with "disable_gc_sections" feature.

2 When building for platform "netx"

3 When building for platform "netx10", "netx50", "netx51" or "netx52"

4 When building for platform "netx100" or "netx500"

5 When building for platform "netx90" or "netx90_app"

6 When building for platform "netx90_app_softfp"

7 When building for platform "netx90_app_hardfp"

**Additional flags passed to GCC Linker when using GNU ARM Embedded Toolchain**

```
-Wl,-gc-sections
```

### 6.1.5 eCosCentric GCC toolchain

The eCosCentric GCC toolchain is provided by eCosCentric to be used in conjunction with the eCos operating system it supports code generation for ARMv5, ARMv6 and ARMv7 architecture compatible CPUs. It is identified by toolchain name "ecoscentric". The code generated by this toolchain is ARM EABI compatible and can be linked with any other binary conforming to ARM EABI.

**Additional flags passed to GCC Compiler when using eCosCentric GCC toolchain**

```
-fshort-enums

-march=armv7e-m  -mfloat-abi=soft (1)
-march=armv8a    -mfloat-abi=soft (2)
```

1 When building for platform "netx90"

2 When building for platform "armv8a" or "netxxxlmpw"

**Additional flags passed to GCC Linker when using eCosCentric GCC toolchain**

```
-Wl,-gc-sections
```

## 6.2 LLVM/CLang based toolchains

### 6.2.1 Hilscher xPIC toolchain

The Hilscher xPIC toolchain is a LLVM/Clang based toolchain used for the xPIC units within the netX10/netX51/netX52/netX90 devices.

**Flags passed to Clang compiler when using Hilscher xPIC toolchain**

```
-g
-gdwarf-4

-O0 (1)
-Os (2)

-mcpu=xpic  (3)
-mcpu=xpic2 (4)
```

(1) For build condition *debug*

(2) For build conditions *debugrel* and *release*

(3) When building for platform "xpic" (netX10)

(4) When building for platform "xpic2" (netX51/netX52,netX90)

**Flags passed to Clang assembler when using Hilscher xPIC toolchain**

```
-Wa,-mmcu=xpic (1)
-Wa,-mmcu=xpic2 (2)
```

(1) When building for platform "xpic" (netX10)

(2) When building for platform "xpic2" (netX51/netX52,netX90)

# Appendix A: Appendix

## A.1 List of figures

## A.2 List of tables

## A.3 List of snippets

## A.4 Legal Notes

### Copyright



### Important notes

Utmost care was/is given in the preparation of the documentation at hand consisting of a user's manual, operating manual and any other document type and accompanying texts. However, errors cannot be ruled out. Therefore, we cannot assume any guarantee or legal responsibility for erroneous information or liability of any kind. You are hereby made aware that descriptions found in the user's manual, the accompanying texts and the documentation neither represent a guarantee nor any indication on proper use as stipulated in the agreement or a promised attribute. It cannot be ruled out that the user's manual, the accompanying texts and the documentation do not completely match the described attributes, standards or any other data for the delivered product. A warranty or guarantee with respect to the correctness or accuracy of the information is not assumed.

We reserve the right to modify our products and the specifications for such as well as the corresponding documentation in the form of a user's manual, operating manual and/or any other document types and accompanying texts at any time and without notice without being required to notify of said modification. Changes shall be taken into account in future manuals and do not represent an obligation of any kind, in particular there shall be no right to have delivered documents revised. The manual delivered with the product shall apply.

Under no circumstances shall Hilscher Gesellschaft für Systemautomation mbH be liable for direct, indirect, ancillary or subsequent damage, or for any loss of income, which may arise after use of the information contained herein.

**Liability disclaimer**

The hardware and/or software was created and tested by Hilscher Gesellschaft für Systemautomation mbH with utmost care and is made available as is. No warranty can be assumed for the performance or flawlessness of the hardware and/or software under all application conditions and scenarios and the work results achieved by the user when using the hardware and/or software. Liability for any damage that may have occurred as a result of using the hardware and/or software or the corresponding documents shall be limited to an event involving willful intent or a grossly negligent violation of a fundamental contractual obligation. However, the right to assert damages due to a violation of a fundamental contractual obligation shall be limited to contract-typical foreseeable damage.

It is hereby expressly agreed upon in particular that any use or utilization of the hardware and/or software in connection with

- Flight control systems in aviation and aerospace;
- Nuclear fission processes in nuclear power plants;
- Medical devices used for life support and
- Vehicle control systems used in passenger transport

shall be excluded. Use of the hardware and/or software in any of the following areas is strictly prohibited:

- For military purposes or in weaponry;
- For designing, engineering, maintaining or operating nuclear systems;
- In flight safety systems, aviation and flight telecommunications systems;
- In life-support systems;
- In systems in which any malfunction in the hardware and/or software may result in physical injuries or fatalities.

You are hereby made aware that the hardware and/or software was not created for use in hazardous environments, which require fail-safe control mechanisms. Use of the hardware and/or software in this kind of environment shall be at your own risk; any liability for damage or loss due to impermissible use shall be excluded.

**Warranty**

Hilscher Gesellschaft für Systemautomation mbH hereby guarantees that the software shall run without errors in accordance with the requirements listed in the specifications and that there were no defects on the date of acceptance. The warranty period shall be 12 months commencing as of the date of acceptance or purchase (with express declaration or implied, by customer's conclusive behavior, e.g. putting into operation permanently).

The warranty obligation for equipment (hardware) we produce is 36 months, calculated as of the date of delivery ex works. The aforementioned provisions shall not apply if longer warranty periods are mandatory by law pursuant to Section 438 (1.2) BGB, Section 479 (1) BGB and Section 634a (1) BGB [Bürgerliches Gesetzbuch; German Civil Code] If, despite of all due care taken, the delivered product should have a defect, which already existed at the time of the transfer of risk, it shall be at our discretion to either repair the product or to deliver a replacement product, subject to timely notification of defect.

The warranty obligation shall not apply if the notification of defect is not asserted promptly, if the purchaser or third party has tampered with the products, if the defect is the result of natural wear, was caused by unfavorable operating conditions or is due to violations against our operating regulations or against rules of good electrical engineering practice, or if our request to return the defective object is not promptly complied with.

**Costs of support, maintenance, customization and product care**

Please be advised that any subsequent improvement shall only be free of charge if a defect is found. Any form of technical support, maintenance and customization is not a warranty service, but instead shall be charged extra.

**Additional guarantees**

Although the hardware and software was developed and tested in-depth with greatest care, Hilscher Gesellschaft für Systemautomation mbH shall not assume any guarantee for the suitability thereof for any purpose that was not confirmed in writing. No guarantee can be granted whereby the hardware and software satisfies your requirements, or the use of the hardware and/or software is uninterruptable or the hardware and/or software is fault-free.

It cannot be guaranteed that patents and/or ownership privileges have not been infringed upon or violated or that the products are free from third-party influence. No additional guarantees or promises shall be made as to whether the product is market current, free from deficiency in title, or can be integrated or is usable for specific purposes, unless such guarantees or promises are required under existing law and cannot be restricted.

**Confidentiality**

The customer hereby expressly acknowledges that this document contains trade secrets, information protected by copyright and other patent and ownership privileges as well as any related rights of Hilscher Gesellschaft für Systemautomation mbH. The customer agrees to treat as confidential all of the information made available to customer by Hilscher Gesellschaft für Systemautomation mbH and rights, which were disclosed by Hilscher Gesellschaft für Systemautomation mbH and that were made accessible as well as the terms and conditions of this agreement itself.

The parties hereby agree to one another that the information that each party receives from the other party respectively is and shall remain the intellectual property of said other party, unless provided for otherwise in a contractual agreement.

The customer must not allow any third party to become knowledgeable of this expertise and shall only provide knowledge thereof to authorized users as appropriate and necessary. Companies associated with the customer shall not be deemed third parties. The customer must obligate authorized users to confidentiality. The customer should only use the confidential information in connection with the performances specified in this agreement.

The customer must not use this confidential information to his own advantage or for his own purposes or rather to the advantage or for the purpose of a third party, nor must it be used for commercial purposes and this confidential information must only be used to the extent provided for in this agreement or otherwise to the extent as expressly authorized by the disclosing party in written form. The customer has the right, subject to the obligation to confidentiality, to disclose the terms and conditions of this agreement directly to his legal and financial consultants as would be required for the customer's normal business operation.

**Export provisions**

The delivered product (including technical data) is subject to the legal export and/or import laws as well as any associated regulations of various countries, especially such laws applicable in Germany and in the United States. The products / hardware / software must not be exported into such countries for which export is prohibited under US American export control laws and its supplementary provisions. You hereby agree to strictly follow the regulations and to yourself be responsible for observing them. You are hereby made aware that you may be required to obtain governmental approval to export, reexport or import the product.

## A.5 Contacts

### Headquarters

**Germany**
Hilscher Gesellschaft für Systemautomation mbH
Rheinstrasse 15
65795 Hattersheim
Phone: +49 (0) 6190 9907-0
Fax: +49 (0) 6190 9907-50
E-Mail: info@hilscher.com

**Support**
Phone: +49 (0) 6190 9907-99
E-Mail: de.support@hilscher.com

### Subsidiaries

**China**
Hilscher Systemautomation (Shanghai) Co. Ltd.
200010 Shanghai
Phone: +86 (0) 21-6355-5161
E-Mail: info@hilscher.cn
**Support**
Phone: +86 (0) 21-6355-5161
E-Mail: cn.support@hilscher.com

**France**
Hilscher France S.a.r.l.
69800 Saint Priest
Phone: +33 (0) 4 72 37 98 40
E-Mail: info@hilscher.fr
**Support**
Phone: +33 (0) 4 72 37 98 40
E-Mail: fr.support@hilscher.com

**India**
Hilscher India Pvt. Ltd.
Pune, Delhi, Mumbai, Bangalore
Phone: +91 8888 750 777
E-Mail: info@hilscher.in
**Support**
Phone: +91 8108884011
E-Mail: info@hilscher.in

**Austria**
Hilscher Austria GmbH
4020 Linz
Phone: +43 732 931 675-0
E-Mail: sales.at@hilscher.com
**Support**
Phone: +43 732 931 675-0
E-Mail: at.support@hilscher.com

**USA**
Hilscher North America, Inc.
Lisle, IL 60532
Phone: +1 630-505-5301
E-Mail: info@hilscher.us
**Support**
Phone: +1 630-505-5301
E-Mail: us.support@hilscher.com

**Japan**
Hilscher Japan KK
Tokyo, 160-0022
Phone: +81 (0) 3-5362-0521
E-Mail: info@hilscher.jp
**Support**
Phone: +81 (0) 3-5362-0521
E-Mail: jp.support@hilscher.com

**Republic of Korea**
Hilscher Korea Inc.
13494, Seongnam, Gyeonggi
Phone: +82 (0) 31-739-8361
E-Mail: info@hilscher.kr
**Support**
Phone: +82 (0) 31-739-8363
E-Mail: kr.support@hilscher.com

**Switzerland**
Hilscher Swiss GmbH
4500 Solothurn
Phone: +41 (0) 32 623 6633
E-Mail: info@hilscher.ch
**Support**
Phone: +41 (0) 32 623 6633
E-Mail: ch.support@hilscher.com

**Italy**
Hilscher Italia S.r.l.
20090 Vimodrone (MI)
Phone: +39 02 25007068
E-Mail: info@hilscher.it
**Support**
Phone: +39 02 25007068
E-Mail: it.support@hilscher.com