HDLgear

Speedup HDL development

Programmers Manual

Authors:	Ruud Overeem
Version	0.2 dd. 18 sept 2018

Content

1 INTRODUCTION **1**

2 REPLACEMENT OF COMMONDICTFILE CLASS **1**

2.1 UTILITY MOD_CONFIGFILES 3
2.2 RELATION WITH OLD COMMONDICTFILE 3

3 CHANGES IN INITIALISATION OF THE ENVIRONMENT **4**

3.1 USAGE OF CONFIGFILES IN SHELL 4
3.2 SET_QUARTUS AND SET_MODELSIM 4

4 PROGRAMMING PRINCIPLES **5**

4.1 SHELL SCRIPTS 5
4.1.1 Shell options *5*
4.1.2 generic.sh *5*
4.1.3 Parsing arguments *5*
4.2 PYTHON 8
4.2.1 Importing packages *8*
4.2.2 Limit export *8*
4.2.3 Argument parsing *8*

1 Introduction

After the decision that the RadioHDL package would be distributed on opencores.org there was the need to clean up the code so that no Astron or Uniboard related code was in the scripts anymore. Furthermore the scripts had to be checked for robustness and clarity of error messages.
This finally resulted in changes in many scripts, hdllib.cfg files and introduced a strong movement to store all 'user tuneable' variables in configuration files.

This document describes the inside of the package where often the relation with the 'old' RadioHDL package is used to explain the changes. *(When RadioHDL is really ancient history these compare section should be removed from this document. ☺)*.

2 Replacement of CommonDictFile class

The CommonDictFile class played an important role in the python environment of oneclick. After many years of maintenance and modification by several developers the class unfortunately grew into a 'god class'. One class that can do everything. To mention a few capabilities of this class:

- It can represent the content of one configfile
- It can represent the content of a whole hierarchical tree of configfiles
- It can be used as bulk modification tool to modify keys and/or values.

So whenever you see an instance of a CommonDictFile in a program you can only guess with which identity in mind the class was constructed by the programmer.

Typically classes are used to 'embed/guard' some data(structure) and provide a few functions for manipulating this data in a controlled way. This normally means that a class is very good in only one or two things. Applying this principle to the functionality of the CommonDictFile class this resulted in eight new classes and one interactive commandline tool. Although this are far more classes than the one we used to have the maintenance is much easier since it is very clear what each class does.

Figure 1 below shows the class diagram.

The main functionality of CommonDictFile is relocated in two base classes:

ConfigFile — This base class can read in one configuration file and stored that content to an OrderedDict inside the class. This is done during the construction of the class. If the read in fails a ConfigFileException is thrown.

- Arguments
  - filename : full filename including the absolute path of the file
  - sections : optional argument to limit the sections that are read in.
  - required_keys : optional list of keynames that must exist in the configfile.
- Variables:
  - filename : name of the file read in without the path.
  - location : path to the file
  - sections : user argument which sections should be stored.
  - content : property that gives you the stored OrderedDict
  - ID : unique identification of this file (defaults to location+filename)
- Functions:
  - resolve_key_references()
  - get_value(key, must_exist=False)

ConfigTree — This base class implements the 'tree'-aspect of CommonDictFile. On construction it reads in a collection of ConfigFiles.

- Arguments
  - rootdirs : list of top directories where to search for files
  - filename : name of the files to search for. Use '*' as wild-char, e.g. '`hdl_buildset_*.cfg`' matches all buildset files.
  - sections : optional argument to limit the sections that are read in.
- Variables:
  - The three arguments are stored as class variables.
  - configfiles : returns a dict containing all read in ConfigFile objects.
- Functions:
  - remove_files_from_tree(files_to_remove)

limit_tree_to(files_to_keep)
get_key_values(key, configfiles=None, must_exist=False)
get_configfiles(key, values=None, user_configfiles=None)

Note: A function `_factory_constructor` is used in the main loop to read in each file that matches the filename(mask) argument. The default implementation calls the ConfigFile constructor. Inhereted classes should implement their own `_factory_constructor`.

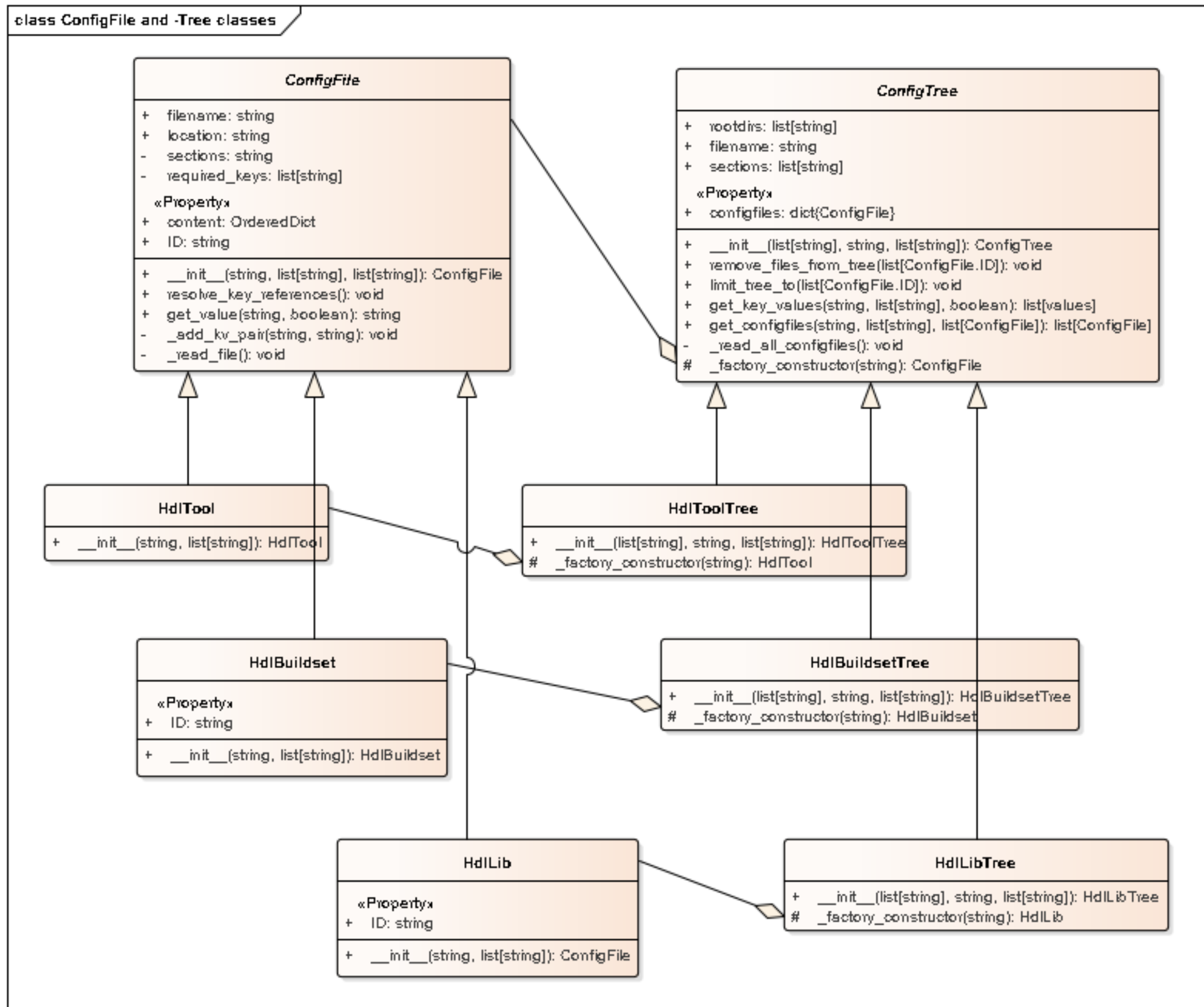

Figure 1: Class diagram of the configurationfile classes.

Three kinds of configuration files are currently used in HDLgear. The keys *inside* these files decide for which flavour a file qualifies. To implement this we created three derived classes that only implement the ID property and they call the base class constructor with their own set of required keys.
In a similar way three flavour of configuration trees are implemented with three derived classes from ConfigTree that only implement their own _factory_constructor function.

Another functionality of the CommonDictFile class is that it can be used to modify collections of files. When modifying files we like the preserve as much of the original file as possible, this includes comments and spatial layout of the file. Since ConfigFile (the only class that reads in the files) discards this kind of information this class cannot be used. Like CommonDictFile that implemented a separate read function for the modification functionality there is now a separate class that does this: RawConfigFile. Together with RawConfigTree (derived from ConfigTree) it forms the base for (bulk) modifying configuration files. See Figure 2.

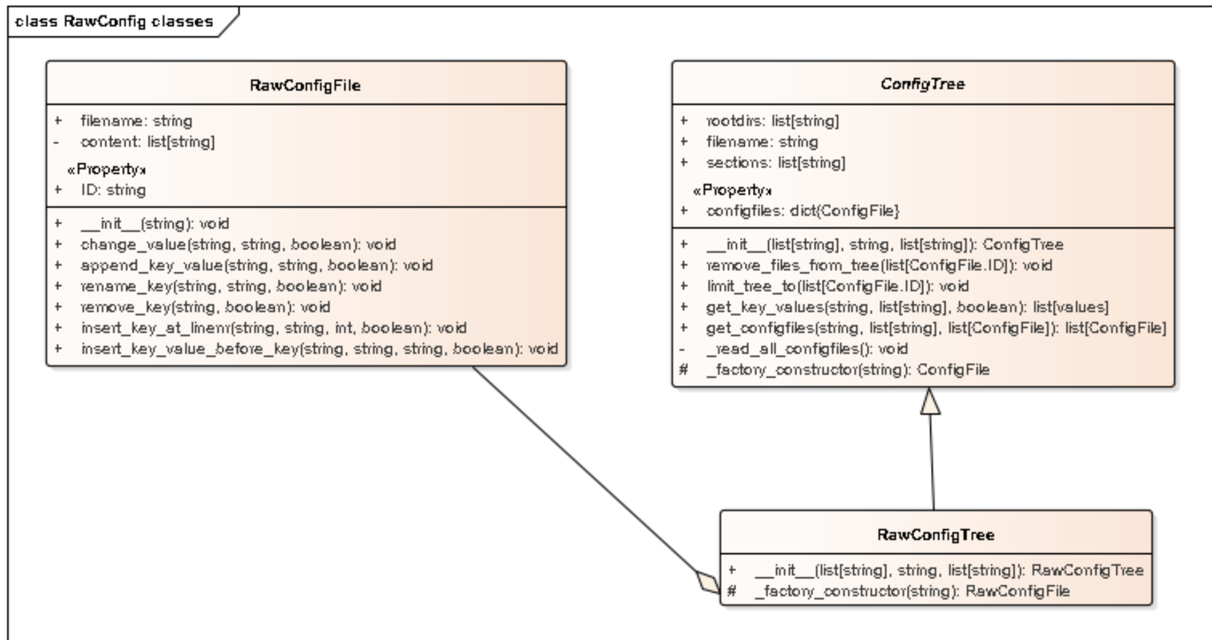

Figure 2: Classes that provide access to the raw content of the configuration file to support modifications.

2.1 Utility mod_configfiles

As small interactive python program `mod_configfiles` implements a tiny menu system that enables you to execute the modification functions that the RawConfigFile class provides.

2.2 Relation with old CommonDictFile

For those who were used to work with CommonDictFile: the next table shows the new function names.

CommonDictFile	**ConfigFile, ConfigTree, mod_configfiles**
dicts()	ConfigTree.configfiles
nof_dicts()	len(ConfigTree.configfiles)
filePathNames()	ConfigTree.configfiles.keys()
filePaths()	iterate over ConfigTree.configfiles, use ConfigFile.location
remove_dict_from_list(dict_to_remove)	ConfigTree.remove_files_from_tree (files_to_remove)
remove_all_but_the_dict_from_list(dict_to_keep)	ConfigTree.limit_tree_to(files_to_keep)
find_all_dict_file_paths(rootDir=None)	obsolete
read_all_dict_files(filePathNames=None)	obsolete
read_dict_file(filePathName=None)	ConfigFile(fullFileName)
write_dict_file(…)	interactive mod_configfiles program
append_key_to_dict_file(…)	interactive mod_configfiles program
insert_key_in_dict_file_at_line_number(…)	interactive mod_configfiles program
insert_key_in_dict_file_before_another_key(…)	interactive mod_configfiles program
remove_key_from_dict_file(…)	interactive mod_configfiles program
rename_key_in_dict_file(…)	interactive mod_configfiles program
change_key_value_in_dict_file(…)	interactive mod_configfiles program
resolve_key_references()	ConfigFile.resolve_key_references()
get_filePath(the_dict)	ConfigFile.location
get_filePathName(the_dict)	ConfigFile.location + '/' + ConfigFile.filename
get_key_values(key, dicts=None, must_exist=False)	ConfigTree.get_key_values(key, configfiles=None, must_exist=False)
get_key_value(key, the_dict, must_exist=False)	ConfigFile.get_value(key, must_exist=False)
get_dicts(key, values=None, dicts=None)	ConfigTree.get_configfiles(key, values=None, user_configfiles=None)

3 Changes in initialisation of the environment

`setup_RadioHDL.sh` is replaced with `init_radiohdl.sh`. The main difference is that the shell environment is kept as clean as possible. Where `setup_RadioHDL.sh` cluttered your environment with many functions (actually everything in generic.sh) `init_radiohdl.sh` defines only three environment variables and extends your path with the necessary paths.

3.1 Usage of configfiles in shell

Using the configuration files is easy since they can be accessed through ConfigFile and ConfigTree. But the content of the configuration files should also be available for shell. The cleanest way to do this is to reuse/wrap the python code so that we don't have to reimplement the file interpretation.
So we made two small python programs that read in a configuration file and print the requested information. Two other small shell scripts invoke those python scripts and execute the information that was printed by the python script.

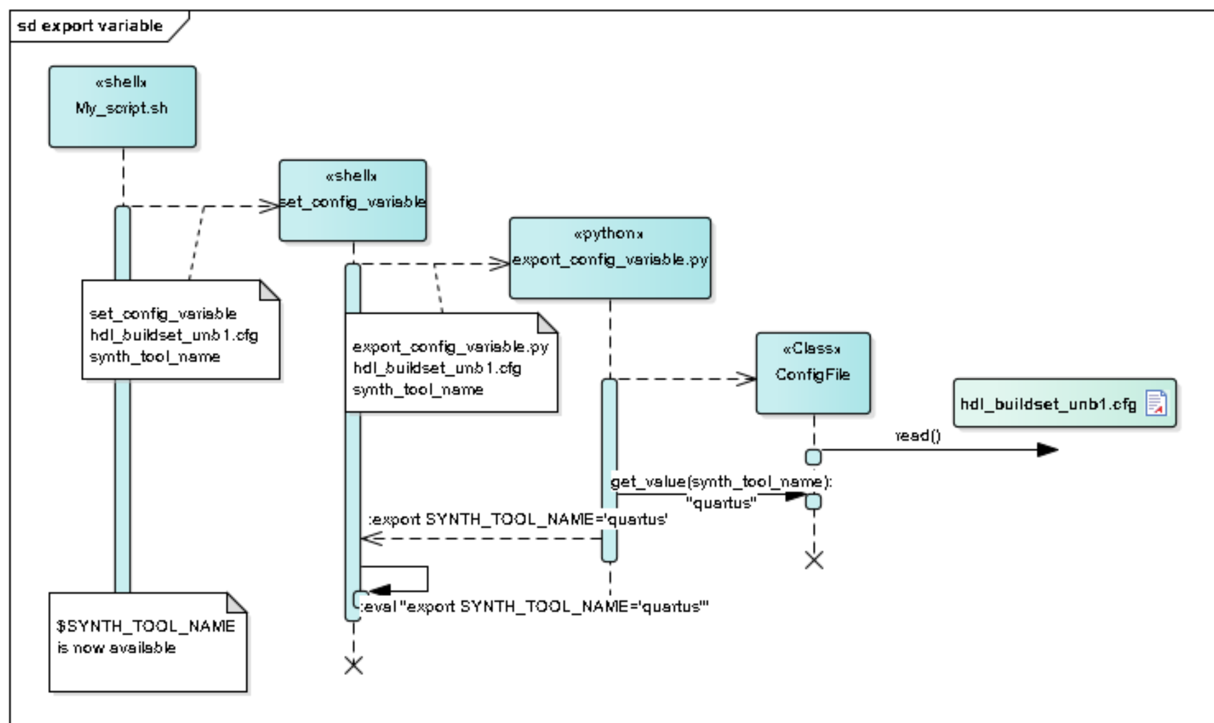

Figure 3: Example how configfile information is made available in shell.

3.2 set_quartus and set_modelsim

Both scripts are completely rewritten and they only use information from the hdl_buildset- and hdl_tool- configurationfile.

4 Programming principles

This chapter describes some principles that were used for designing and writing the scripts.
In general we can state that:

- scripts must give a syntax help message when they are invoked the wrong way or when '-h' or '--help' is given as an argument
- invocation arguments are strictly checked. Unknown arguments result in an error (and the help message).
- the order of the invocation arguments is trivial
- everything is assumed to be fault/wrong/undefined until the opposite is proven.

4.1 Shell scripts

4.1.1 Shell options

Each shell scripts starts with the line:
```
#!/bin/bash -eu
```

-e option: exit immediate on error
-u option: treat undefined variables and parameters as an error

The -u option helps us to find uninitialized variables but also makes it harder to use the invocation arguments: `MY_VAR=$1` exits the script with an ugly error message if no arguments were used. Also trying to test the arguments like `if [ "$1" == "something" ]` will exit the script.
However you can access a probably-undefined variable, say `VARNAME` with `${VARNAME:-}` with triggering the -u option.

All scripts nowadays expect the buildset name to be the first argument so the following code snippet catches the undefined first argument is a proper way.
```
BUILDSET=${1:-}
if [ "${BUILDSET}" = "" ]; then
  hdl_error $0 "Please specify all arguments\nUsage: $0 <buildset>"
fi
```

4.1.2 generic.sh

One of the first lines in each script is:
```
# read generic functions
. ${RADIOHDL}/tools/generic.sh
```

This imports some generic functions like path_add, hdl_exec, hdl_exit, and so on.
By importing this in each script instead of in setup_radiohdl.sh the environment of the user stays clean and we import the functions only when we need them.

4.1.3 Parsing arguments

Unlike Python, there is no out of the box argument parser for shell programming that works well. There are two flavours: `getopts` and `getopt`.

getopts
`getopts` only accepts short options like `-e something` or `-v`. The major flaw of `getopts` however is that *options should always precede the arguments*.
For example if we have a script getopts_test.sh like:

```
EXT=
VERBOSE=false
while getopts e:v option
do
    case "$option" in
       e) EXT=${OPTARG} ;;
       v) VERBOSE=true ;;
      \?) echo "OOPS"; exit 1 ;;
    esac
done
shift $(($OPTIND - 1))
echo "EXT="$EXT
echo "VERBOSE="$VERBOSE
echo "POSITIONALS=$@"
```

then **`getopts_test.sh -v -e something cats and dogs`**
will give you the correct output:

```
EXT=something
VERBOSE=true
POSITIONALS=cats and dogs
```

but **`getopts_test.sh -v cats and dogs -e something`**
silently treats the -e as positional argument:

```
EXT=
VERBOSE=true
POSITIONALS=cats and dogs -e something
```

getopt
The other flavour is `getopt`. This parser is not picky about order of options and arguments and even also excepts long options like `--extension=something` or `--verbose`. Short and long options can be mixed and are recognized by the number is minus signs.
This is where it goes wrong! When the user makes a type like `-extension=something` (one minus instead of two) it sees short option `-e` with the value `xtension=something`.

The test script:

```
EXT=
VERBOSE=false
eval set -- `getopt -o e:v --long extension:,verbose -n $0 -- "$@"`
while true ; do
    case "$1" in
        -e|--extension)
            EXT="${2:+$2}"
            shift 2
            ;;
        -v|--verbose)
            VERBOSE=true
            shift
            ;;
        --) shift ; break ;;
        \?) echo "OOPS"; exit 1 ;;
        *) echo "Internal error!"; exit 1 ;;
    esac
done
echo "EXT="$EXT
echo "VERBOSE="$VERBOSE
echo "POSITIONALS=$@"
```

`getopt_test.sh -v -e something cats and dogs`
will give you the correct output:

```
EXT=something
VERBOSE=true
POSITIONALS=cats and dogs
```

also invocations like
`getopt_test.sh --verbose -e something cats and dogs`
`getopt_test.sh -v --extension=something cats and dogs`

getopt_test.sh -v cats and dogs -e something
getopt_test.sh --verbose cats and dogs --extension=something
will all give the correct result.

but **getopt_test.sh --verbose cats and dogs -extension=something**
silently treats the typo of -extension and give the following result:

```
EXT=xtension=something
VERBOSE=true
POSITIONALS=cats and dogs
```

Chosen solution

Since the two out of the box tools both have major flaws we have to make a DIY parser. After extensive research on the internet the following solution was made that is fully correct and is as tiny as possible.

```
missing_option_argument() {
    exit_with_error "Option $1 expects an argument"
}

exit_with_error() {
    echo "$@"
    cat <<@EndOfHelp@
Usage: $(basename $0) [options] arguments
Options
-e | --extension=   <explain>
-v | --verbose      <explain>
Arguments
<explain>
@EndOfHelp@
    exit 1
}

POSITIONAL=()
EXT=
VERBOSE=false
while [[ $# -gt 0 ]]
do
    case $1 in
        -e)
            [ $# -lt 2 ] && missing_option_argument $1
            EXT="$2" ; shift ;;
        --extension=*)
            EXT=${1#*=} ;;
        -v|--verbose)
            VERBOSE=true ;;
        -h|--help)
            exit_with_error "Information about the options and arguments" ;;
        -*|--*)
            exit_with_error "Unknown option: "$1 ;;
        *)
            POSITIONAL+=("$1") ;;
    esac
    shift
done
if [ ${#POSITIONAL[@]} -gt 0 ]; then
    set -- "${POSITIONAL[@]}"
fi
echo "EXT="$EXT
echo "VERBOSE="$VERBOSE
echo "POSITIONALS=$@"
```

To give neat responses to the user when something goes wrong we defined two small functions. `missing_option_argument()` tells the user that a value is expected for the option and then calls the `exit_with_error` function.
`exit_with_error()` shows the user the correct syntax of the command and exits the script with exitcode 1. Please provide useful information to the user.

The main loop of the parser is only slightly larger than with the out-of-the-box-with-major-flaws parsers. The main idea behind the parser loop is:

1) handle all defined options. Options without an argument can be combine in one 'case' match. For options that do need an argument we have to treat the short and the long version separate as the short version covers two arguments (no connecting '=' sign) and the long version includes the value of the option.
2) catch 'help' options
3) reject all other options
4) gather the arguments that may be anywhere in the invocation order.

Finally assign the collected positional arguments to $1, $2, and so on.

This DIY parser meets all programming principles we defined in the beginning of this chapter.

4.2 Python

4.2.1 Importing packages

When you need only a few functions from a package you can better limit the import to these few functions. This keeps the 'lookup tables' of python smaller, makes the code cleaner and give insight in what you use from the packages.
So instead of writing:

```
import os.path
dir_name=os.path.expandvars('RADIOHDL')
```

write:

```
from os.path import expandvars
dir_name=expandvars('RADIOHDL')
```

4.2.2 Limiting export

If a source file contains both public functions/classes as well as private ones you can limit what a user will see if it imports your file by defining the `__all__` variable. E.g. by adding the line

```
__all__ = [ 'public_function_1', 'public_class_1', 'public_constant' ]
```

to your source file limits the exposure the these three entities when someone imports your file.

4.2.3 Parsing arguments

Fortunately python has an excellent parser for arguments: `ArgumentParser` from the `argparse` package. Look on internet for the manual or look e.g. in `export_config_variables.py` how to use this parser. In short:

1) create an `ArgumentParser` instance.
2) for each argument and for each option call `add_argument`
3) finally call `parse_args()`