



**Documentation**

**GenericScons**

**ITK Engineering AG**   
www.itk-engineering.de  
Im Speyerer Tal 6  
76761 Rülzheim  
Tel: +49-7272-7703-0  
Fax: +49-7272-7703-100

1. General

|  |  |
| --- | --- |
| Document state: | draft |
| Date: | 24 Sep 2015 |
| Author: | André Fischer |
| Projekttitel: | GenericScons |

|  |  |  |  |
| --- | --- | --- | --- |
| **Rev.** | **Date** | **Description** | **Executor** |
| 0.1 | 01.03.2013 | Document created | Sasmito |
| 0.2 | 03.04.2013 | Update chapter “Building Target”.  Add new chapters, “Development Tools”, “SCons common files”, “Sconscript File”, “Configuration File” and “Build result” | Sasmito |
| 0.3 | 09.04.2013 | Update example.  Add IDE version. | Sasmito |
| 0.4 | 16.04.2013 | Add chapter “Modifying CIPP Scons build files for other component” | Sasmito |
| 0.5 | 04.11.2013 | Adapt content to shared Scons for MFC400 algo | Sasmito |
| 0.6 | 30.07.2014 | Revise GenericScons documentation | Fischer |
| 0.7 | 26.09.2014 | Minor modifications | Fischer |
| 0.8 | 07.10.2014 | Update + new layout | Fischer |
| 0.9 | 16.10.2014 | Include review comments by Roland Stark | Fischer |
| 0.10 | 24.10.2014 | Include Chapter 5 on GenericScons options | Fischer |
| 0.11 | 30.10.2014 | Include Sections 5.3 and 5.4 on new GenericScons options | Fischer |
| 0.12 | 20.11.2014 | Include Section 5.9 | Singh |
| 0.13 | 21.11.2014 | Minor clarification | Fischer |
| 0.14 | 03.12.2014 | Add Section 5.2  Amend FAQ | Fischer |
| 0.15 | 10.12.2014 | Update Section 4.1.7.2 | Fischer |
| 0.16 | 22.01.2015 | Add Section 4.5 | Fischer |
| 0.17 | 05.02.2015 | Minor corrections in Section 4.5 | Fischer |
| 0.18 | 09.02.2015 | Update Section 4.5 | Fischer |
| 0.19 | 20.02.2015 | Include Section 4.4.3 | Singh |
| 0.20 | 20.02.2015 | Update Section 4.5 | Fischer |
| 0.21 | 23.04.2015 | Update Section 3 | Singh |
| 0.22 | 23.04.2015 | Update Section 5.1.8.1 | Singh |
| 0.23 | 23.04.2015 | Update Section 5.1.8.2 | Singh |
| 0.24 | 03.06.2015 | Update Section 3 | Singh |
| 0.25 | 24.06.2015 | Include section 4.6 | Bhagawati |
| 0.26 | 06.07.2015 | Add Remark at end of Section 4.5 | Fischer |
| 0.27 | 14.07.2015 | Move path info from Chapter 3 to Section 2.2.1  Update Chapter 3 for Scons version in use | Fischer |
| 0.28 | 31.07.2015 | Update section 4.1.2 | Bhagawati |
| 0.29 | 28.08.2015 | Added section 4.1.5.1 | Bhagawati |
| 0.30 | 14.09.2015 | Update Section 5.1.8.2. | Fischer |
| 0.31 | 23.09.2015 | Added section 4.1.5.2 | Bhagawati |
| 0.32 | 24.09.2015 | Update Section 3.1  Added Section 4.1.1 | Fischer |
| 0.33 | 05.11.2015 | Updated Section 4.1.7 | Bhagawati |
| 0.34 | 07.03.2016 | Added section 5.1.10 | Bhagawati |
| 0.35 | 25.03.2016 | Updated sections 1.3 and 3.1 | Lenin Palanisamy |
| 0.36 | 25.11.2016 | Added section 4.1.8.4 | Tridip Bhagawati |
| 0.37 | 12.12.2016 | Updated section 2.2.3 | Tridip Bhagawati |
| 0.38 | 12.12.2016 | Updated section 3 and 3.4 | Tridip Bhagawati |

Table : Change history

1. Terms, abbreviations and definitions
   1. Terms
   2. Abbreviations
   3. Definitions
2. Contents

[I General 2](#_Toc447187211)

[II Terms, abbreviations and definitions 3](#_Toc447187212)

[II.1 Terms 3](#_Toc447187213)

[II.2 Abbreviations 3](#_Toc447187214)

[II.3 Definitions 3](#_Toc447187215)

[III Contents 4](#_Toc447187216)

[IV Figures 5](#_Toc447187217)

[V Tables 6](#_Toc447187218)

[1 Introduction 7](#_Toc447187219)

[1.1 Purpose / task 7](#_Toc447187220)

[1.2 Validity of this document 7](#_Toc447187221)

[1.3 Used documents and references 7](#_Toc447187222)

[2 Design 8](#_Toc447187223)

[2.1 Structure – An overview 8](#_Toc447187224)

[2.2 Requirements for GenericScons 9](#_Toc447187225)

[2.2.1 Structural requirements according to CM 9](#_Toc447187226)

[2.2.2 Development Tools 11](#_Toc447187227)

[2.2.3 IDE Version 11](#_Toc447187228)

[2.3 GenericScons structure embedded into the CM structure 11](#_Toc447187229)

[2.3.1 Structure related to ECU Targets 12](#_Toc447187230)

[2.3.2 Structure related to PC simulation targets 12](#_Toc447187231)

[2.3.3 Structure related to HiL targets 13](#_Toc447187232)

[3 GenericScons constituents and their roles 15](#_Toc447187233)

[3.1 SCons extensions 16](#_Toc447187234)

[3.2 SConscript.py’s 17](#_Toc447187235)

[3.2.1 Shared SConscript.py’s 17](#_Toc447187236)

[3.2.2 Organization of shared SConscript.py’s 20](#_Toc447187237)

[3.2.3 Non-shared SConscript.py’s 22](#_Toc447187238)

[3.3 Configuration Files 23](#_Toc447187239)

[3.3.1 Common configuration files 23](#_Toc447187240)

[3.3.2 Private configuration files 24](#_Toc447187241)

[3.4 Implementing GenericScons for a new algo component 26](#_Toc447187242)

[4 Building with GenericScons 28](#_Toc447187243)

[4.1 Build targets 28](#_Toc447187244)

[4.1.1 Ti-Tools extraction 28](#_Toc447187245)

[4.1.2 Algo Libraries 28](#_Toc447187246)

[4.1.3 SDL files 28](#_Toc447187247)

[4.1.4 Algo Fingerprint 29](#_Toc447187248)

[4.1.5 PC Simulation Targets 30](#_Toc447187249)

[4.1.5.1 Generate RTA XML 30](#_Toc447187250)

[4.1.5.2 Component specific copy operations 31](#_Toc447187251)

[4.1.6 Generate doxygen documentation 31](#_Toc447187252)

[4.1.7 Evalboard out-files 32](#_Toc447187253)

[4.1.8 Generate Microsoft Visual C++ project 32](#_Toc447187254)

[4.1.8.1 GenericScons-Release ≤ 1.3.0 32](#_Toc447187255)

[4.1.8.2 GenericScons Release ≥ 1.3.1 34](#_Toc447187256)

[4.1.8.3 Build from Command Line and from Visual Studio – Differences 36](#_Toc447187257)

4.1.8.4 Generate Microsoft Visual Studio 2010 solution………………………...37

[4.1.9 Generate CCS 5 Eclipse project 37](#_Toc447187258)

[4.2 How to build 38](#_Toc447187259)

[4.2.1 Via command line 38](#_Toc447187260)

[4.2.2 Via IDE 41](#_Toc447187261)

[4.3 Sysbios library 41](#_Toc447187262)

[4.4 Visualization of SCons workflow 41](#_Toc447187263)

[4.4.1 Building ecu libraries and CCS 5 projects 41](#_Toc447187264)

[4.4.2 Building HIL simulation out file and CCS 5 project 44](#_Toc447187265)

[4.4.3 MACRO defined in CCS 5 project for ecu libraries and HIL simulation 46](#_Toc447187266)

[4.4.4 Building PC simulation wrapper and Microsoft Visual C++ project 47](#_Toc447187267)

[4.5 Global Build Setting for individual targets 49](#_Toc447187268)

[4.6 Built in macro of different compilers for Visual C++ project 52](#_Toc447187269)

[5 GenericScons Option 52](#_Toc447187270)

[5.1 GenericScons Flags 52](#_Toc447187271)

[5.1.1 Flag “verbose” 52](#_Toc447187272)

[5.1.2 Flag “skip\_ide\_generation” 53](#_Toc447187273)

[5.1.3 Flag “build\_from\_ide” 53](#_Toc447187274)

[5.1.4 Flag “build\_from\_vs” 54](#_Toc447187275)

[5.1.5 Flag “build\_from\_ccs” 54](#_Toc447187276)

[5.1.6 Flag “custom\_sim\_copy” 54](#_Toc447187277)

[5.1.7 Flag “profile” 55](#_Toc447187278)

[5.1.8 Flag “-h” 55](#_Toc447187279)

[5.1.8.1 Case of component-specific version of *scons.bat* 55](#_Toc447187280)

[5.1.8.2 Case of shared version of *scons.bat* 55](#_Toc447187281)

[5.1.9 Flag “install\_to\_deliverables\_dir” 56](#_Toc447187282)

[5.1.10 Flag “no\_err\_warning” 56](#_Toc447187283)

[5.2 GenericScons SubAlgo-Feature 56](#_Toc447187284)

[5.2.1 Initiation of SubAlgo-Feature 56](#_Toc447187285)

[5.2.2 Setting the define sets for SubAlgos 57](#_Toc447187286)

[5.2.3 Handling PC build targets not being shared to the project – “utilities” 58](#_Toc447187287)

[6 Build results 59](#_Toc447187288)

[7 Errors 62](#_Toc447187289)

[7.1 Finding Error 62](#_Toc447187290)

[7.2 Typical Errors 62](#_Toc447187291)

[7.2.1 Build results are stored to the Source directory 63](#_Toc447187292)

[7.3 Frequently asked questions 63](#_Toc447187293)

[8 Build Farm 66](#_Toc447187294)

1. Figures

[Figure 1: How GenericScons builds targets 8](#_Toc424642965)

[Figure 2: Required folder structure 10](#_Toc424642966)

[Figure 3: SConscripts and scfg files for algo library and fingerprint 12](#_Toc424642967)

[Figure 4: SConscripts and scfg files for DLLs 13](#_Toc424642968)

[Figure 5: SConscripts and scfg files for EVM HIL Wrapper 14](#_Toc424642969)

[Figure 6: sconscript\_setup\_config.scfg 27](#_Toc424642970)

[Figure 7: Visual Studio Solution of GenericScons ≤ 01.03.01 33](#_Toc424642971)

[Figure 8: Visual Studio Solution of GenericScons ≥ 01.03.01 3](#_Toc424642972)6

[Figure 9: List of build targets 3](#_Toc424642973)9

[Figure 10: Build of ecu libraries - Step 1 4](#_Toc424642974)2

[Figure 11: Build of ecu libraries - Step 2 4](#_Toc424642975)3

[Figure 12: Build of HiL simulation out – Step 1 4](#_Toc424642976)4

[Figure 13: Build of HiL Simulation out - Step 2 4](#_Toc424642977)5

[Figure 14: Build of PC simulation wrapper - Step 1 4](#_Toc424642978)7

[Figure 15: Build of PC simulation wrapper - Step 2 4](#_Toc424642979)8

[Figure 16: Build of PC simulation wrapper - Step 3 4](#_Toc424642980)9

[Figure 17: Non-verbose build output 5](#_Toc424642981)2

[Figure 18: Verbose build output 5](#_Toc424642982)3

[Figure 19: Build products of SCons in 04\_Build folder 5](#_Toc424642983)9

[Figure 20: Build products of SCons in 05\_Deliverables 60](#_Toc424642984)

[Figure 21: Build products of SCons under 05\_Testing 6](#_Toc424642985)1

[Figure 22: Finding Errors 6](#_Toc424642986)2

[Figure 23: Example of xml file with SCons target 6](#_Toc424642987)6

1. Tables

[Table 1: Change history 3](#_Toc424642988)

[Table 2: References and supplementary documents 7](#_Toc424642989)

[Table 3: Executing GenericScons 40](#_Toc424642990)

[Table 4: MACRO defined for CCS 5 project 4](#_Toc424642991)6

[Table 5: Default Flags for ECU Libraries 50](#_Toc424642992)

[Table 6: Default Flags for HiL Targets 5](#_Toc424642993)1

[Table 7: Default Flags for PC Simulation Targets 5](#_Toc424642994)1

[Table 8: Abbreviations 66](#_Toc424642995)

# Introduction

## Purpose / task

GenericScons represents a common build environment for the algo components of ADC GmbH.

This document describes structure and application of the GenericSCons build environment.

## Validity of this document

## Used documents and references

|  |  |
| --- | --- |
| Pos. | Title |
| 1 | MFC400 Scons Build Environment Requirements, Revision 1 |
| 2 | AD CM Sharing Concept, Version 05.00.00 |
| 3 | PDO SW Manual, Version 2.10 |
| 4 | Quick Start Guide : Shared SCons Build Environment |
| 5 | Fingerprint\_manual.docx |
| 6 | QAC\_manual.docx |
| 7 | GenericScons\_Polyspace\_Integration.pptx |

Table 2: References and supplementary documents

# Design

## Structure – An overview

GenericSCons build environment consists of windows batch file which starts SCons (scons.bat), python scripts (SConstruct, SConscript.py’s and further .py-files) and SCons configuration files (.scfg files).

SConstruct and SConscript.py’s contain build commands being read and executed by SCons. All other .py-scripts represent extensions to the existing SCons and .scfg-files provide certain configuration for the build.

SConstruct, most of SConscript.py-files, and all other .py-scripts are shared, i.e. they are the same for all components and reside under 02\_Development\_Tools\scons\_tools as a share in MKS. Only scons.bat, certain SConscript-files, and most of the .scfg files are non-shared, i.e. they need to be created and maintained by the algo components themselves.

Before SCons or, more precisely, the actual build is executed, shared files (SConstruct and SConscript.py’s) are copied from 02\_Development\_Tools\scons\_tools to the required locations within 04\_Engineering and 05\_Testing. And then those copied files are executed by SCons in order to perform the build.

SConstruct is the main script that is executed by **SCons**. This script triggers the build of one or more targets by calling SConscript files. All .scfg-files are text files holding python variables that will be read by SConstruct and SConscript.py before building targets. The content of those .scfg-files is component-specific in the majority of cases, so generally they are non-shared. Lists of source files and include-paths are only two examples for the possible content of a configuration file. The separation of build information into SConstruct, SConscript.py’s, further .py-files, and .scfg files is chosen to make the scripts of GenericScons **generic**.

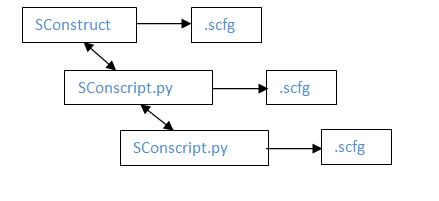


Figure 1: How GenericScons builds targets

Let us remark on the phrase “GenericScons”. In fact, this name is misleading, since the term “Generic” tends to indicate that the build environment being discussed here is a tool that has been implemented and provided by the SCons-developpers themselves and that has been published. However, the open source product “SCons” is utilized to implement “GenericScons” as required by ADC GmbH. So, as initially employed, something like “shared scons environment” would be more appropriate. But since the term “GenericScons” seems to have been established in the algo components, we stick to this phrase

Let us give an overview over the document at hand:

* Chapter 2 gives a rough overview over the functionalities of GenericScons.
* Chapter 3 deals with the individual constituents of GenericScons, i.e. all the files that add up the GenericScons. Moreover their role for the build is explained in detail.
* Chapter 4 explains how to build and also comments on the build targets that are integrated.
* Chapter 5 deals with certain GenericScons options and special features.
* In Chapter 6 the location of the individual build results is shown.
* Chapter 7 should be referred to when errors occur. There a list of FAQ is included which will be updated continuously
* Chapter 8 an introduction to the build farm is given.

## Requirements for GenericScons

### Structural requirements according to CM

The following folder structure is required in order to use the GenericScons build. Note that this structure follows the CM plan (xxx is the component name).

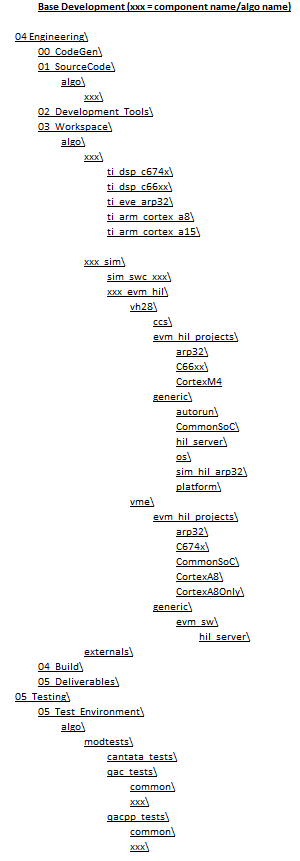


Figure : Required folder structure

* 04\_Engineering\01\_SourceCode\ is the folder which contains the algo source code.
* 04\_Engineering\02\_Development\_Tools\ contains compilers and all tools used by the algo component, including the SCons tool and SCons extensions. This folder is shared in MKS.
* 04\_Engineering\03\_Workspace\ is the folder which contains the SConstruct file and the build process is executed in 03\_Workspace\algo\xxx[[1]](#footnote-1).
* 04\_Engineering\04\_Build\ is the folder which contains the build results of GenericScons.
* 04\_Engineering\05\_Deliverables\ contains the results of SCons build process which will be shared in MKS.
* 05\_Testing\ This folder contains SConscripts for unit tests and qac/qacpp tests and some build results (DLLs) are copied to this folder to be tested with MTS.

### Development Tools

All development tools required by this component are stored in 02\_Development\_Tools\. The following tools need to be provided:

* doxygen : generates doxygen documentation from a doxyfile configuration file.
* movpy : executable python for running SCons.
* pdo\_tool : tools for extracting measurement meta-data from source code.
* scons\_tools : SCons common files.
* sdlcompiler : tool to convert SDL file to CDL file.
* ti\_tools : Toolchain from Texas Instrument.
* visual\_studio\_scripts : additional scripts used by visual studio

Before implementing GenericScons for an algo component, please make sure to provide all required tools in order to avoid errors caused by missing tools. Since shared SConscripts will be copied from scons\_tools, it is important to have the correct scons\_tools checkpoint.

When there is a new feature added, a new **scons\_tools** checkpoint is created. As soon as the algo component moves onto this new checkpoint the updated python scripts are present and all the new SConscript files are copied automatically to the project folders when SCons is executed.

### IDE Version

In GenericScons, three IDEs are used. These IDEs are

1. Visual Studio 2005
2. Visual Studio 2010
3. Code Composer Studio 5.3.0

## GenericScons structure embedded into the CM structure

Now let us have a closer look at which files are used by GenericScons. As already said before, there are shared files, where most of them are copied to the correct location under 04\_Engineering and 05\_Testing, and there are component-specific files which need to be created in algo components also to be maintained by the algo component teams. In the following pictures shared files are depicted in orange and component-specific ones in green. The respective function of those constituents is explained in detail in Chapters 3 and 4.

### Structure related to ECU Targets

For every algo component one or more ecu libraries are being built and, if activated, a fingerprint of algo and arp is generated.

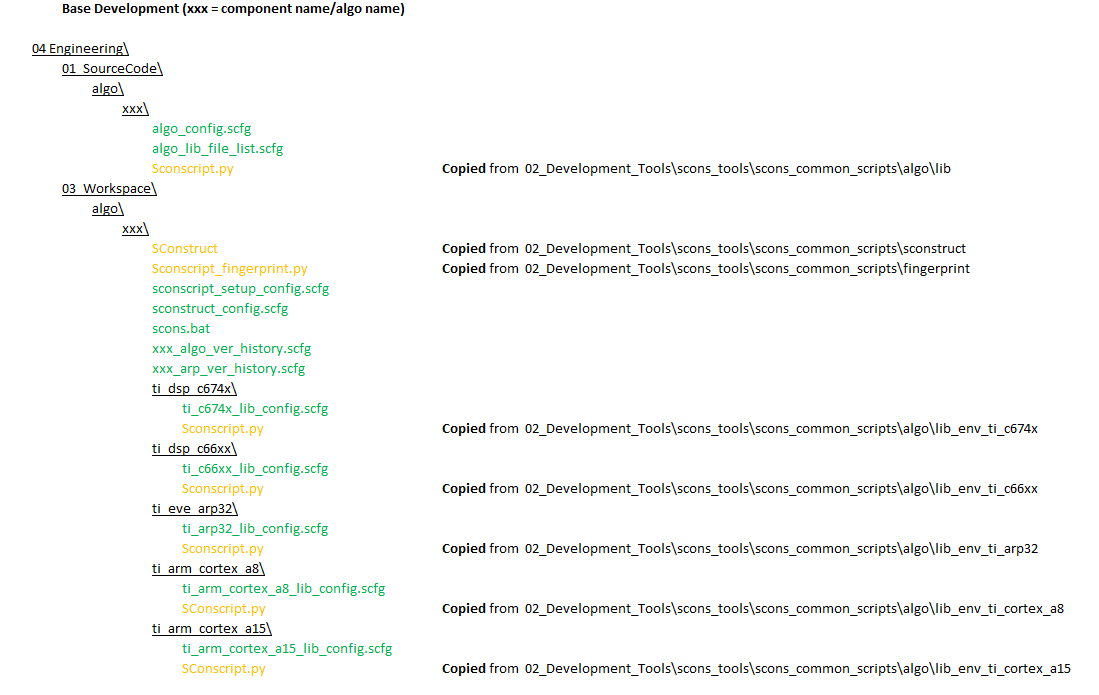


Figure 3: SConscripts and scfg files for algo library and fingerprint

Depending on which architectures are used, SConstruct then calls the SConscript.py-files in the corresponding subfolders of 03\_Workspace\algo\xxx. Those SConscript-files specify certain architecture-specific settings as well as component-specific settings through the .scfg-file in the same folder. Finally all those SConscript.py-files call 01\_Source\_Code\algo\xxx\SConscript.py respectively in order to build the static library.

### Structure related to PC simulation targets

There are shared libraries, static libraries as well as executables being built for pc simulation.

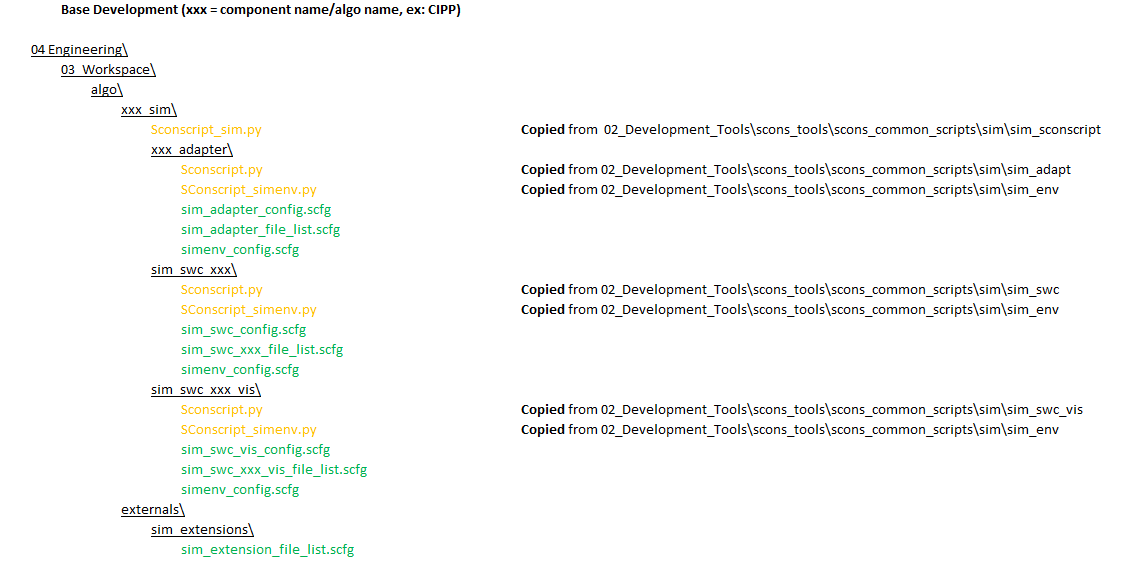
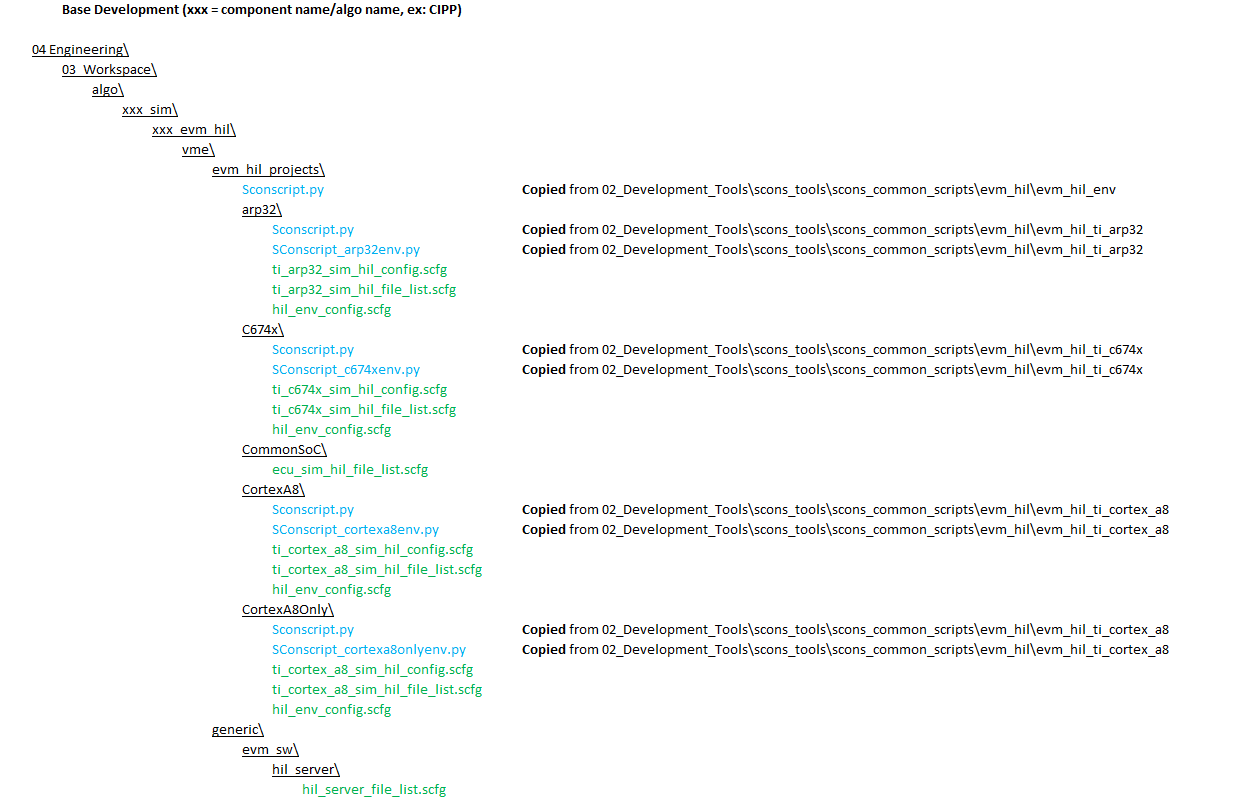
****

Figure 4: SConscripts and scfg files for DLLs

To this end SConstruct calls the shared script 03\_Workspace\algo\xxx\_sim\SConscript\_sim.py (GenericScons Release ≥ 1.3.1) resp. the non-shared script 03\_Workspace\algo\xxx\_sim\SConscript.py (GenericScons Release ≤ 1.3.0). For each pc simulation target the corresponding SConscript\_simenv.py is called which then calls the SConscript.py in the very same folder. While the SConscript\_simenv.py-files are all identical, the SConscript.py-files differ according to the type of simulation target, for more detailed information on those scripts, we refer to paragraph 3.2.1.

### Structure related to HiL targets

The structure for the build of .out-files for the eval board can be seen in the following diagram. GenericScons support “**vision mid-eve**” and “**vision high 28**”. The correspoing sub-structure is essentially identical.



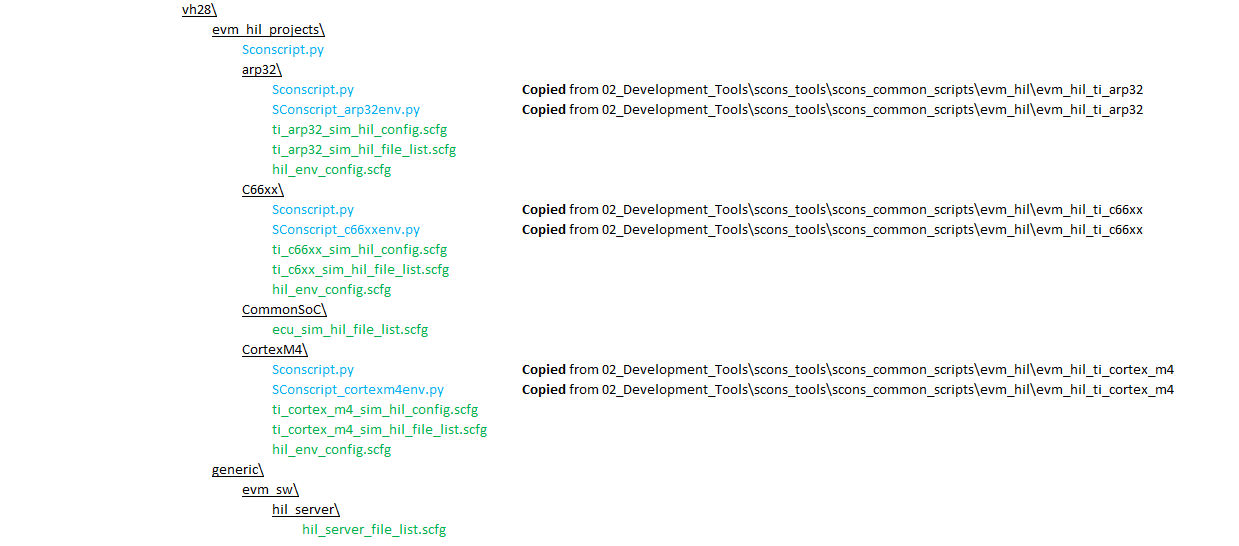


Figure 5: SConscripts and scfg files for EVM HIL Wrapper

SConstruct calls 03\_Workspace\algo\xxx\_sim\xxx\_evm\_hil\vme\evm\_hil\_projects\SConscript.py which calls SConscript\_...env.py, which in turn calls SConscript.py in the respective folder. So the build structure is similar to the one described in paragraph 2.3.2.

# GenericScons constituents and their roles

The purpose of this chapter is to explain the role of all the files being part of GenericScons, i.e. of the content of the folder 02\_Development\_Tools\scons\_tools\ as well as all the GenericScons constituents being located in the project under 04\_Engineering and 05\_Testing and their connection.

SCons common files are stored under 02\_Development\_Tools\scons\_tools\ so that they can be shared to all algo components. The folder scons\_tools contains the following subfolders:

* **docs :** contains documentations for GenericScons.
* **planning :** contains lists of components that use shared SCons build environment for MFC400 in an excel file.
* **scons\_adas\_extensions :** contains python scripts, where numerous builder are defined and which are used by SConscripts.
* **scons\_common\_config :** contains common configuration used by SConscripts.
* **scons\_common\_scripts :** contains shared SConscripts which will be copied to 04\_Engineering and 05\_Testing when scons is executed.
* **scons\_templates :** contains templates for all files which are to be maintained by the algo teams such as scons.bat and .scfg files used by SCons build environment. The template files are arranged according to the folder structure in 04\_Engineering and 05\_Testing. When rolling out GenericScons for a new component, copy the files in this folder to 04\_Engineering and 05\_Testing and modify them according to the algo component except scons.bat.
* **scons :** Local scons-package, which GenericScons is based on plus documentation. Official release, which this scons-source is based on, is 2.2.0. There are only few minor differences between the scons package in use and the official release, e.g. certain patches discussed in scons forum.
* **setup :** contains a batch file, a scfg config file and a python script to automate the initial copying of scfg config files to the required locations.

The GenericScons constituents in use by a component can be grouped into three large sets:

* **Common python scripts:**

These refer to scons extensions, i.e. functionalities which have not been provided by the SCons software itself, but have been implemented by ADC GmbH. They are stored under **scons\_tools\scons\_adas\_extensions** and theyrepresent a set of python scripts which are executed while performing the build. The purpose of the individual files is explained in Section .

* **Common build scripts:**

SConstruct and SConscript.py’s which are stored under **scons\_tools\ scons\_common\_scripts;** prior to every execution of SCons they are copied automatically to a predetermined location under 04\_Engineering or 05\_Testing and called from this new location. We focus on these scripts in section 3.2.

* **Configuration files:**

There are common configuration files under **scons\_tools\scons\_common\_config** as well as poject-specific ones under 04\_Engineering and 05\_Testing. They contain certain information on the build which is required by the common scripts of GenericScons – see Section 3.3.

A special role of GenericScons plays scons.bat for each component. This is a component-specific batch-file which triggers the scons-build. It is stored under 03\_Workspace\algo\xxx and essentially executes SCons. A template for this file is stored under 02\_Development\_Tools\scons\_tools\scons\_templates\03\_Workspace\algo\xxx. We will not go into more detail on this file in this treatise.

## SCons extensions

All the extensions used can be found under **scons\_tools\scons\_adas\_extensions.** Here is a complete list of the corresponding python scripts.

* **tms470r1x.py**

This extension is used to setup ARM build environment using TI compiler. It specifies the ARM TI compiler used, include directories and library directories.

* **arm\_gcc.py**

This extension is used to setup ARM build environment using gcc.

* **doxygen.py**

This extension is used to generate HTML documentation from doxygen doxyfile.

* **eclipse\_cdt.py**

This extension is used to generate Eclipse project for unit test.

* **eclipse\_cdt\_ccs5.py**

This extension is used to generate CCS 5 Eclipse project.

* **fingerprint.py**

This extension is used to calculate checksum from the content of source files. It is used to create a C header file containing the value of the checksum.

* **generate\_DebugServerScripting.py**

This extension is used to create **\_CopyDSScriptLoadCmdToClipboard.bat** and **DebugServerScripting.js** which are used to run the debugger and load **out files** to one or more processor cores.

* **help\_menu.py**

Generates GenericScons help menu including a list of available targets, see Section 5.1.8.

* **msvc-addon.py, msvs-patched.py**

These scripts contain extensions and modifications to the Microsoft Visual Studio builder which is already supplied by SCons.

* **pdo.py**

This extension is used to generate SDL and CDL files containing measurement meta-data extracted from the source code.

* **qac.py**

This extension is used to generate QAC project files as well as to perform code analysis and generate the compliance report.

* **profile.py**

Allows using a profiling mode for GenericScons, see Section 5.1.7.

* **ram\_rom\_algo\_libsize.py, areas.py, libs\_obj.py, maps.py**

These extensions are used to show the size of the library.

* **sconscript\_setup.py**

This extension is used to copy shared SConscripts from 02\_Development\_Tools to 04\_Engineering and 05\_Testing.

* **sconstruct\_helpers.py**

This extension contains python functions used by SCons build environment.

* **ti\_eve\_arp32.py**

This extension is used to setup ARP32 build environment. It specifies the TI compiler used, include directories and library directories.

* **tms320dm6000**

This extension is used to setup DSP build environment. It specifies the TI DSP compiler used, include directories and library directories.

* **unittest.py**

SCons extension for unit test

* **unzip.py**

SCons extension for unzipping archives

* **vsscript.py**

Additional builder to patch Visual Studio rc file.

* **polyspace.py**

This extension is used to generate polyspace project files.

When needed those scripts are executed in order to set up the consctruction environment variables for GenericScons.

## SConscript.py’s

### Shared SConscript.py’s

All shared GenericScons scripts are located under **scons\_tools\scons\_common\_scripts** and when executing scons build all required files are copied from their original location into the project. This behavior is controlled by 03\_Workspace\algo\sr\sconscript\_setup\_config.scfg, see paragraph 3.3. In this paragraph will indicate both paths respectively and shortly explain what the SConscript.py in question serves for.

**SConstruct**

* 03\_Workspace\algo\xxx\SConsctruct (from \sconstruct)

Gathers global and component specific settings such as “ca8\_used”, “include\_unittests” and triggers the corresponding SConscripts.

**Fingerprint**

* 03\_Workspace\algo\xxx\SConscript\_fingerprint.py (from \fingerprint\SConscript\_fingerprint.py)

If activated this script generates the md5 checksum of algo and arp sources and compares those checksums with existing revisions in 03\_Workspace\algo\xxx\xxx\_algo\_ver\_history.scfg and 03\_Workspace\algo\xxx\xxx\_arp\_ver\_history.scfg. Checksums as well as revision numbers of algo and arp are written in corresponding C-Header-files. See Fingerprint\_manual.docx for more information.

**Algo libraries**

* 03\_Workspace\algo\xxx\ti\_dsp\_c674x\SConscript.py (from \algo\lib\_env\_ti\_c674x):
* 03\_Workspace\algo\xxx\ti\_dsp\_c66xx\SConscript.py (from \algo\lib\_env\_ti\_c66xx):
* 03\_Workspace\algo\xxx\ti\_eve\_arp32\SConscript.py (from \algo\lib\_env\_ti\_eve\_arp32):
* 03\_Workspace\algo\xxx\ti\_arm\_cortex\_a8\SConscript.py (from \algo\lib\_env\_ti\_cortex\_a8):
* 03\_Workspace\algo\xxx\ti\_arm\_cortex\_a15\SConscript.py (from \algo\lib\_env\_ti \_cortex\_a15):

In all those files the construction environment according to global and component-specific settings such as compiler paths, variants, compiler flags, etc, is set up and the following SConscript is called.

* 01\_Source\_Code\algo\xxx\SConscript.py (from \algo\lib):

Imports existing construction environments and reads source and include lists in order to build a static algo library and sdl-files for each architecture and variant respectively.

**Hil – vme**

* 03\_Workspace\algo\xxx\_sim\xxx\_evm\_hil\vme\evm\_hil\_projects\SConscript.py (from evm\_hil\evm\_hil\_env)

Gathers global and component specific settings such as “ca8\_used”, “eve\_used”, etc. and triggers the corresponding hil SConscripts

* 03\_Workspace\algo\xxx\_sim\xxx\_evm\_hil\vme\evm\_hil\_projects\arp32\SConscript\_arp32env.py (from evm\_hil\evm\_hil\_ti\_arp32)

Reads global and component-specific settings in order to setup the construction environment for arp32-out-file and calls the following SConscript.

* 03\_Workspace\algo\xxx\_sim\xxx\_evm\_hil\vme\evm\_hil\_projects\arp32\SConscript.py (from evm\_hil\evm\_hil\_ti\_arp32)

Imports existing construction environment and builds arp32-out-file.

* 03\_Workspace\algo\xxx\_sim\xxx\_evm\_hil\vme\evm\_hil\_projects\C674x\SConscript\_c674xenv.py (from evm\_hil\evm\_hil\_ti\_c674x)

Reads global and component-specific settings in order to setup the construction environment for c674x-out-file and calls the following SConscript.

* 03\_Workspace\algo\xxx\_sim\xxx\_evm\_hil\vme\evm\_hil\_projects\C674x\SConscript.py (from evm\_hil\evm\_hil\_ti\_c674x)

Imports existing construction environment and builds c674x-out-file.

* 03\_Workspace\algo\xxx\_sim\xxx\_evm\_hil\vme\evm\_hil\_projects\CortexA8\SConscript\_cortexa8env.py (from evm\_hil\evm\_hil\_ti\_cortex\_a8)

Reads global and component-specific settings in order to setup the construction environment for cortex-a8-out-file and calls the following SConscript.

* 03\_Workspace\algo\xxx\_sim\xxx\_evm\_hil\vme\evm\_hil\_projects\CortexA8\SConscript.py (from evm\_hil\evm\_hil\_ti\_cortex\_a8)

Imports existing construction environment and builds cortex-a8-out-file.

* 03\_Workspace\algo\xxx\_sim\xxx\_evm\_hil\vme\evm\_hil\_projects\CortexA8Only\SConscript\_cortexa8onlyenv.py (from evm\_hil\evm\_hil\_ti\_cortex\_a8)

Reads global and component-specific settings in order to setup the construction environment for cortex-a8-out-file and calls the following SConscript.

* 03\_Workspace\algo\xxx\_sim\xxx\_evm\_hil\vme\evm\_hil\_projects\CortexA8Only\SConscript.py (from evm\_hil\evm\_hil\_ti\_cortex\_a8)

Imports existing construction environment and builds cortex-a8-out-file.

*Remark:* “CortexA8-folder is used when algo runs on DSP and ARM. If DSP is neglected for the algo, “CortexA8Only” shall be used.

**Hil – vh28**

All the following scripts have the analogue purpose compared to “Hil – vme”.

* 03\_Workspace\algo\xxx\_sim\xxx\_evm\_hil\vh28\evm\_hil\_projects\SConscript.py (from evm\_hil\evm\_hil\_env)
* 03\_Workspace\algo\xxx\_sim\xxx\_evm\_hil\vh28\evm\_hil\_projects\arp32\SConscript\_arp32env.py (from evm\_hil\evm\_hil\_ti\_arp32)
* 03\_Workspace\algo\xxx\_sim\xxx\_evm\_hil\ vh28\evm\_hil\_projects\arp32\SConscript.py (from evm\_hil\evm\_hil\_ti\_arp32)
* 03\_Workspace\algo\xxx\_sim\xxx\_evm\_hil\vh28\evm\_hil\_projects\C66xx\SConscript\_c66xxenv.py (from evm\_hil\evm\_hil\_ti\_c66xx)
* 03\_Workspace\algo\xxx\_sim\xxx\_evm\_hil\ vh28\evm\_hil\_projects\C66xx\SConscript.py (from evm\_hil\evm\_hil\_ti\_c66xx)
* 03\_Workspace\algo\xxx\_sim\xxx\_evm\_hil\vh28\evm\_hil\_projects\CortexM4\SConscript\_cortexm4env.py (from evm\_hil\evm\_hil\_ti\_cortex\_m4)
* 03\_Workspace\algo\xxx\_sim\xxx\_evm\_hil\ vh28\evm\_hil\_projects\CortexM4\SConscript.py (from evm\_hil\evm\_hil\_ti\_cortex\_m4)

**PC simulation**

* 03\_Workspace\algo\xxx\_sim\SConscript\_sim.py (from \sim\sim\_sconscript)

Setups up construction environment, reads pc component lists and calls corresponding SConscript\_simenv.py’s.

* 03\_Workspace\algo\xxx\_sim\<simulation target folder>\SConscript\_simenv.py (from \sim\sim\_env)

Imports and adapts existing construction environment according to component-specific and target-specific settings and calls the following SConscript.

* 03\_Workspace\algo\xxx\_sim\<simulation target folder>\SConscript.py (from either \sim\sim\_swc or \sim\sim\_swc\_vis or \sim\sim\_adapt or \sim\sim\_lib or \sim\sim\_exe)

Imports existing construction environment and builds target. Depending on the type of target the appropriate SConscript.py has to be chosen when implementing GenericScons for this component. We have the following SConscript.py-files at our disposal (the correspondingly required .scfg-files are mentioned in brackets, see paragraph 3.3.2 for more details):

* \sim\sim\_swc: sdl-file, .res-file generated, shared library (requires sim\_swc\_config.scfg, sim\_swc\_xxx\_file\_list.scfg, simenv\_config.scfg)
* \sim\sim\_swc\_vis: .res-file generated, shared library (typically a visualization; it requires sim\_swc\_vis\_config.scfg, sim\_swc\_xxx\_vis\_file\_list.scfg, simenv\_config.scfg)
* \sim\sim\_adapt: shared library (requires sim\_adapter\_config.scfg, sim\_adapter\_file\_list.scfg, simenv\_config.scfg)
* \sim\sim\_lib: static library (requires sim\_adapter\_config.scfg, sim\_adapter\_file\_list.scfg, simenv\_config.scfg)
* \sim\sim\_exe: executable (requires sim\_adapter\_config.scfg, sim\_adapter\_file\_list.scfg, simenv\_config.scfg)

In exceptional cases, it is possible to use a component-specific SConscript.py-file instead of a provided shared SConscript.py, when it is needed. This is done by deactivating the corresponding copy action and checking in a component-specific version to the required location in the project, see paragraph 3.3.2. However, this should generally be avoided, since all updates on the shared script would have to be applied to this customized SConscript.py-file as well and this would usually add to the duties of the algo component team. If this task would be neglected, future updates might cause trouble. So, if needed, it is much more feasible to add a new shared script to the GenericScons environment, which can be maintained by the GenericScons responsible.

### Organization of shared SConscript.py’s

A SConscript.py in GenericScons is generally organized in several parts. As a rule of thumb these parts comprise:

* Importing construction environment and/or global variables when needed.
* Reading SCons configuration files containing tool paths, source files and include paths.
* Reading SConstruct helper or other python scripts to get access to required python functions.
* Setup/adapt construction environment.
* Building target (library, executable, etc) - quite often by calling some further SConscript.py-file.
* Return build results or other files when needed.

***Note:*** Not all parts are available in all SConscript.py-files.

An example of a SConscript file can be seen below. This SConscript (from 01\_Source\_Code\algo\xxx\) builds algo library using environment imported from other SConscript.

# SConscript to build algo library

#-------------------------

# import

#-------------------------

Import("env", "target", "component\_name", "algo\_name")

import os

#-------------------------

# read configuration file

#-------------------------

sconscript\_dir = os.path.dirname((lambda x:x).func\_code.co\_filename)

execfile(sconscript\_dir + "/algo\_config.scfg")

# get common scons config containing tool paths

execfile(File("#../../../02\_Development\_Tools/scons\_tools/scons\_common\_config/common\_config.scfg").abspath)

#-------------------------

# read helper file

#-------------------------

# read sconstruct\_helpers.py

execfile(File("#../../../02\_Development\_Tools/scons\_tools/scons\_adas\_extensions/sconstruct\_helpers.py").abspath)

#-------------------------

# setup environment

#-------------------------

local\_env = env.Clone()

# add source files and include file paths

if "TI\_C674X" in local\_env["\_BUILD\_TARGET"]:

src = c674x\_src

local\_env.Append(CPPPATH = c674x\_cpp\_path)

local\_env.Prepend(CPPPATH = c674x\_prepend\_cpp\_path)

cpp\_path = c674x\_cpp\_path + c674x\_prepend\_cpp\_path

arch = "ti\_c674x"

elif "TI\_CORTEX\_A8" in local\_env["\_BUILD\_TARGET"]:

src = ca8\_src

local\_env.Append(CPPPATH = ca8\_cpp\_path)

local\_env.Prepend(CPPPATH = ca8\_prepend\_cpp\_path)

cpp\_path = ca8\_cpp\_path + ca8\_prepend\_cpp\_path

arch = "ti\_cortex\_a8"

elif "SIM" in local\_env["\_BUILD\_TARGET"]:

src = pc\_sim\_src

local\_env.Append(CPPPATH = pc\_sim\_cpp\_path)

local\_env.Prepend(CPPPATH = pc\_prepend\_cpp\_path)

cpp\_path = pc\_sim\_cpp\_path + pc\_prepend\_cpp\_path

arch = "pc"

elif "TI\_ARP32" in local\_env["\_BUILD\_TARGET"]:

src = arp32\_src

local\_env.Append(CPPPATH = arp32\_cpp\_path)

local\_env.Prepend(CPPPATH = arp32\_prepend\_cpp\_path)

cpp\_path = arp32\_cpp\_path + arp32\_prepend\_cpp\_path

arch = "ti\_arp32"

#-------------------------

# build static library

#-------------------------

lib = local\_env.StaticLibrary(target, src)

#-------------------------

# return results

#-------------------------

# get absolute include path

cpp\_path = map(lambda x: str(Dir(x).srcnode().abspath), cpp\_path)

# get absolute source file

cpp\_src = abspath\_filenames(src)

# get preprocessor define

defines = local\_env["CPPDEFINES"]

# return library, source files and include paths

Return(["lib", "cpp\_src", "cpp\_path", "defines"])

Target Variant

Most targets in algo component are generated in at least two variants, namely “debug” and “release” (and possible some further ones). Usually those variants use the same source files but different compilation options and flags. When building more than one variant, SConstruct calls a first SConscript.py-file, where the latter itself calls a second SConscript.py-file. The first SConscript.py is used for setting up the construction environment and the second one for building the target. It is important that the build results (object files, executable files, library files, etc) are not placed in the same folder for different variants. Otherwise SCons complains and does not perform the build process because the second build results would overwrite the first build results. SCons provides the **VariantDir()** function to specify where the build results are to be placed. This function has three parameters, the target folder, source folder and whether to duplicate the source files in the target folder or not.

**VariantDir('target', 'src', duplicate=0)**

The following example shows how to use VariantDir() function.

for e in dsp\_env\_list: # We have generally two variants, release and debug

env = e.Clone()

# setup output folder

**dsp\_algo\_build\_dir** = (build\_dir + "/algo/" + algo\_name +

"/ti\_c674x/" + env["variant"])

dsp\_algo\_deliverables\_dir = (deliverables\_dir + "/lib/ti\_c674x/algo/" +

algo\_name + "/")

# Specify target folder as dsp\_algo\_build\_dir. The 2nd parameter of VariantDir

# is the location of the source folder relative to the location of this SConscript file.

# The location of the source files is defined in 01\_Source\_Code/algo/ and all source files

# must be under the location of the source folder (inside

# 01\_Source\_Code/algo)

env.VariantDir(**dsp\_algo\_build\_dir,**

**"../../../../",**

**duplicate = 0**)

# Call SConscript file (located in 01\_Source\_Code/algo/xxx/) to build the library.

# Since we have specified the source folder before,

# we just need to give the location of the SConscript file from the source folder

# and append it with target folder.

result = SConscript(**dsp\_algo\_build\_dir + "/01\_Source\_Code/algo/" + component\_name + "/SConscript.py"**,

exports = {

"env" : env,

"target" : (target\_name + "\_" + env["variant"] + ".lib"),

"component\_name" : component\_name,

"algo\_name" : algo\_name

})

if "release" in env["variant"]:

algo\_ti\_c674x\_lib = env.Install([dsp\_algo\_build\_dir, dsp\_algo\_deliverables\_dir],

result[0][0])

algo\_ti\_c674x\_qac = env.GenerateQAC(result[1], result[2], result[3], component\_name, algo\_name, "01\_Source\_Code/algo", ".\..\common\CompilerPers\TMS320C6x\_TI\_DSP\CodeComposer\_DSP6x.p\_c")

else:

algo\_ti\_c674x\_lib = env.Install([dsp\_algo\_build\_dir],

result[0][0])

Alias(component\_name + "\_algo\_ti\_c674x\_" + env["variant"], algo\_ti\_c674x\_lib)

Alias(component\_name + "\_algo\_ti\_c674x", algo\_ti\_c674x\_lib)

Alias(component\_name + "\_algo\_ecu", algo\_ti\_c674x\_lib)

Alias(component\_name + "\_algo", algo\_ti\_c674x\_lib)

if algo\_name != component\_name:

Alias(algo\_name + "\_algo\_ti\_c674x\_" + env["variant"], algo\_ti\_c674x\_lib)

Alias(algo\_name + "\_algo\_ti\_c674x", algo\_ti\_c674x\_lib)

Alias(algo\_name + "\_algo\_ecu", algo\_ti\_c674x\_lib)

Alias(algo\_name + "\_algo", algo\_ti\_c674x\_lib)

Alias("all", algo\_ti\_c674x\_lib)

Clean(algo\_ti\_c674x\_lib, dsp\_algo\_build\_dir)

Alias(algo\_name + "\_qac\_ti\_c674x", algo\_ti\_c674x\_qac)

### Non-shared SConscript.py’s

Certain components require targets to be built in a more specific way than it is provided by GenericScons. If needed, this can be accounted for via component specific SConscript.py-files. The following possibilities are implemented:

* Components SR and TSA maintain so-called utility targets for pc simulation, which are not shared into projects like MFC400 or SRLCam, etc. So it is desired to build them with GenericScons whenever they are present in the sandbox at hand. To this end a corresponding “03\_Workspace\xxx\_utils\SConscript\_utils.py” is called per default, whenever it exists. This is done in 03\_Workspace\algo\xxx\_sim\SConscript\_sim.py
* Sometimes the default implementation of pdo support does not suffice the needs of a component. One of the reasons might be the need for several sdl-files with different sources or the inability of pdo\_scan to scan the source code correctly which might be related with gcc-specific commands. Whenever this happens, it is possible to generate sdl-files in a component-specific way. Here is what needs to be done:
  1. Add the line

generate\_component\_specific\_sdl = True

to 01\_Source\_Code\algo\xxx\algo\_config.scfg

* 1. Adapt scons\_templates\01\_Source\_Code\algo\xxx\SConscript\_sdl.py accordingly and store it under 01\_Source\_Code\algo\xxx in the project. This script will then be called by 01\_Source\_Code\algo\xxx\SConscript.py.

## Configuration Files

Configuration files are required throughout the GenericScons environment. These are text files containing certain python variables providing information on certain paths, compiler options or libraries to be linked in. We distinguish between common and private configuration files.

### Common configuration files

The common SCons configuration files in 02\_Development\_Tools\scons\_tools\scons\_common\_config contain “global configurations” for the build. Here are the two configuration files in use:

02\_Development\_Tools\scons\_tools\scons\_common\_config\common\_config.scfg, 02\_Development\_Tools\scons\_tools\scons\_common\_config\externals\_include\_file\_list.scfg

Both they contain information oncommon paths, common complier flags, etc., see, e.g., common\_config.scfg:

# common configuration for SCons build environment

# development tool paths

engineering\_dir = "#../../../"

source\_code\_dir = engineering\_dir + "01\_Source\_Code/"

development\_tools\_dir = engineering\_dir + "02\_Development\_Tools/"

workspace\_dir = engineering\_dir + "03\_Workspace/"

build\_dir = engineering\_dir + "04\_Build/"

deliverables\_dir = engineering\_dir + "05\_Deliverables/"

testing\_dir = engineering\_dir + "../05\_Testing/"

ti\_tools\_dir = development\_tools\_dir + "ti\_tools/"

ti\_tools\_compiler\_dir = ti\_tools\_dir + "compiler/"

c6000\_code\_gen\_tool\_dir = ti\_tools\_compiler\_dir + "c6000/"

tms470\_code\_gen\_tool\_dir = ti\_tools\_compiler\_dir + "arm/"

arp32\_tool\_base\_dir = ti\_tools\_compiler\_dir + "arp32/"

ti\_bios\_dir = ti\_tools\_dir + "bios/"

ti\_bios\_package\_dir = ti\_bios\_dir + "packages/"

ti\_xdctools\_dir = ti\_tools\_dir + "xdctools/"

ti\_xdctools\_package\_dir = ti\_xdctools\_dir + "packages/"

edma3\_drv\_dir = ti\_tools\_dir + "edma3\_lld/"

evestarterware\_dir = ti\_tools\_dir + "evestarterware/"

eve\_edma\_csl\_dir = evestarterware\_dir + "edma\_csl/"

networking\_dir = ti\_tools\_dir + "networking/"

ndk\_dir = networking\_dir + "ndk/"

ndk\_package\_dir = ndk\_dir + "packages/"

pdk\_nsp\_dm814x\_dir = networking\_dir + "nsp/"

pdk\_nsp\_dm814x\_package\_dir = pdk\_nsp\_dm814x\_dir + "packages/"

pdk\_nsp\_vayu\_dir = pdk\_nsp\_dm814x\_dir

pdk\_nsp\_vayu\_package\_dir = pdk\_nsp\_dm814x\_package\_dir

arm\_gcc\_code\_gen\_tool\_dir = ti\_tools\_compiler\_dir + "gcc-arm-none-eabi/"

arm\_fastrts\_dir = tms470\_code\_gen\_tool\_dir + "lib/"

doxygen\_dir = development\_tools\_dir + "doxygen/bin/"

graphviz\_dir = development\_tools\_dir + "graphviz/bin/"

pdo\_dir = development\_tools\_dir + "pdo\_tool/"

pdo\_scan\_config = development\_tools\_dir + "pdo\_tool/msvc\_x86\_pdo\_cmt.cfg"

pdo\_algo\_scan\_config = development\_tools\_dir + "pdo\_tool/dm643x\_pdo\_cmt.cfg"

gnutools\_dir = development\_tools\_dir + "scons\_tools/gnutools/"

sdl\_compiler\_dir = development\_tools\_dir + "sdlcompiler/"

visual\_studio\_script\_dir = development\_tools\_dir + "visual\_studio\_scripts/"

cantata\_tool\_dir = ["c:/tools/qa\_systems/cantata","c:/LegacyApp/qa\_systems/cantata", "c:/LegacyApp/IPL/cantpp"]

# QAC directories

QACPATH = 'C:\\tools\QAC\\'

QACPPPATH = 'C:\\tools\\QACPP-2.5\\'

QAWBIN = 'C:\\tools\\QAW\\bin\\'

QARBIN = 'QAR-1.1\\bin'

qacpp\_rep\_dir = "#../../../../05\_Testing/02\_Reports/algo/modtests/qacpp\_tests/"

qac\_rep\_dir = "#../../../../05\_Testing/02\_Reports/algo/modtests/qac\_tests/"

# scons adas extensions path

scons\_adas\_extensions\_path = development\_tools\_dir + "scons\_tools/scons\_adas\_extensions/"

# common compile and linker options

# WIN DLL:

sim\_rel\_common\_ccflags = "/EHsc /c /errorReport:prompt /O2 /FD /MD /nologo /W3"

sim\_rel\_common\_linkflags = "/DLL /DEBUG"

sim\_dbg\_common\_ccflags = "/EHsc /c /errorReport:prompt /Od /RTC1 /MDd /nologo /W3"

sim\_dbg\_common\_linkflags = "/DLL /DEBUG"

# TI C674x lib:

c674x\_rel\_common\_ccflags = "--abi=eabi"

c674x\_rel\_optim\_flag = "-o3"

c674x\_dbg\_common\_ccflags = "--abi=eabi -g"

# TI ARP32 lib:

arp32\_rel\_common\_ccflags = "-O3 --symdebug:skeletal -kv -kh --silicon\_version=v210"

arp32\_dbg\_common\_ccflags = "-g -kv -kh --silicon\_version=v210 --gen\_func\_subsections"

# TI Cortex A8 lib:

cortex\_a8\_rel\_common\_ccflags = "--neon --display\_error\_number --diag\_warning=225 -mf=5 -k -s -os -on=2 --float\_support=vfpv3"

cortex\_a8\_dbg\_common\_ccflags = "-g --display\_error\_number --diag\_warning=225 -k --float\_support=vfpv3"

# TI Cortex A8 GCC lib:

cortex\_a8\_gcc\_rel\_common\_ccflags = ""

cortex\_a8\_gcc\_dbg\_common\_ccflags = "-g"

# TI Cortex A15 GCC lib:

cortex\_a15\_gcc\_rel\_common\_ccflags = ""

cortex\_a15\_gcc\_dbg\_common\_ccflags = ""

**Note:**

# symbol in a string (“#” or ‘#’) refers to the folder containing SConstruct file. When # is not in a string, it is a python comment symbol.

### Private configuration files

Those configuration files need to be present in the project and have to be maintained by the component’s team. Templates for these component-specific files can be found in **scons\_tools\** **scons\_templates.**

Information stored in .scfg-files of GenericScons can be split into two groups:

1. **Component-specific settings** 
   1. **Copy Action:**

03\_Workspace\algo\sr\sconscript\_setup\_config.scfg

Copy shared scripts to correct location in project.

* 1. **Target Organization:**

03\_Workspace\algo\sr\sconstruct\_config.scfg

Which architectures are used for algo, which simulation targets are required, activate unittests, etc.

1. **Target-specific settings :**
   1. **Algo Configurations:**

01\_Source\_Code\algo\xxx\algo\_config.scfg:

general settings on sources for different libs, pdo setting, etc.

01\_Source\_Code\algo\xxx\algo\_lib\_file\_list.scfg:

source and include lists.

* 1. **ECU Library Configs:**

03\_Workspace\algo\xxx\<arch>\<arch>\_lib\_config.scfg:

Define variants with corresponding compiler flags, defines, etc.

* 1. **PC Simulation:**

There are three .scfg-files per pc simulation target in the corresponding folder, e.g. 03\_Workspace\algo\xxx\_sim\sim\_swc\_xxx\simenv\_config.scfg:

Define variants with compiler flags, defines, libpaths, etc.

03\_Workspace\algo\xxx\_sim\sim\_swc\_xxx\ sim\_swc\_config.scfg:

General settings for build target, e.g. external libraries, target extension, etc. 03\_Workspace\algo\xxx\_sim\sim\_swc\_xxx\sim\_swc\_file\_list.scfg:

Source and Include lists.

Individual names of those .scfg-files may vary depending on the type of simulation target, paragraph 3.2.1.

* 1. **Hil Simulation:**

There are three .scfg-files per out-file in individual folders, e.g., 03\_Workspace\algo\xxx\_sim\xxx\_evm\_hil\vme\evm\_hil\_projects\arp32, etc.,

The aggregation of information and the names of the corresponding .scfg-files is similar to the concept for pc simulation.

Let us have a look at two examples of .scfg-files and give some further useful information on their content.

1. 01\_Source\_Code\algo\xxx\algo\_lib\_file\_list.scfg: Here source lists and include paths are provided

# algo source files

cipp\_algo\_sources = """

../00\_Custom/cipp\_wrp/cipp\_main.cpp

RefCore/CIPP\_RefCore.cpp

RefCore/CIPP\_Math.cpp

"""

# algo pdo source files

cipp\_algo\_pdo\_sources = """

../00\_Custom/cipp\_wrp/cipp\_pdo\_tags\_file.c

"""

# algo include paths

cipp\_algo\_include\_paths = """

#../../../01\_Source\_Code/algo/00\_Custom/cipp\_wrp

#../../../01\_Source\_Code/algo/cipp/RefCore

#../../../01\_Source\_Code/algo/cipp/ti\_eve\_arp32

"""

# algo common include paths

cipp\_algo\_common\_include\_paths = """

#../../../01\_Source\_Code/algo/cml

#../../../01\_Source\_Code/common

#../../../01\_Source\_Code/common/rte

"""

**Note:**

1. Source file shall be defined relative to location of the configuration file (.scfg file).
2. Include paths should be defined using SConstruct folder as the reference.
3. Under xxx\_algo\_common\_include\_paths exactly those paths should be stored that are needed for the code but which can be considered as “externals” to the component. The reason is the following: When QAC analyses the code it excludes certain directories from being scanned. And these are exactly those whoch are saved in this variable.
4. Another example is given by 03\_Workspace\algo\xxx\ti\_dsp\_c674x\ ti\_c674x\_lib\_config.scfg, which contains information on component-specific settings for generation of the corresponding algo library:

# create release variant

release\_variant = "release"

release\_ccflags = "--abi=eabi --define=c6747 --display\_error\_number --diag\_warning=225 --optimizer\_interlist --opt\_for\_speed=5 -k"

release\_cppdefines = ["NDEBUG", "DEFINE1", "DEFINE2"]

# create debug variant

debug\_variant = "debug"

debug\_ccflags = "--abi=eabi -g --define=c6747 --display\_error\_number --diag\_warning=225 -k"

debug\_cppdefines = ["DEFINE1", "DEFINE2"]

# variable containing compiler flags, defines for all variants

# when creating additional variant beside "release" and "debug", the variant must be added in # this variable

variant\_list = [

[release\_variant, release\_ccflags, release\_cppdefines],

[debug\_variant, debug\_ccflags, debug\_cppdefines],

]

# library name. The full name will be "xxx\_c674x\_{variant}.lib, e.g. sib\_c674x\_release.lib

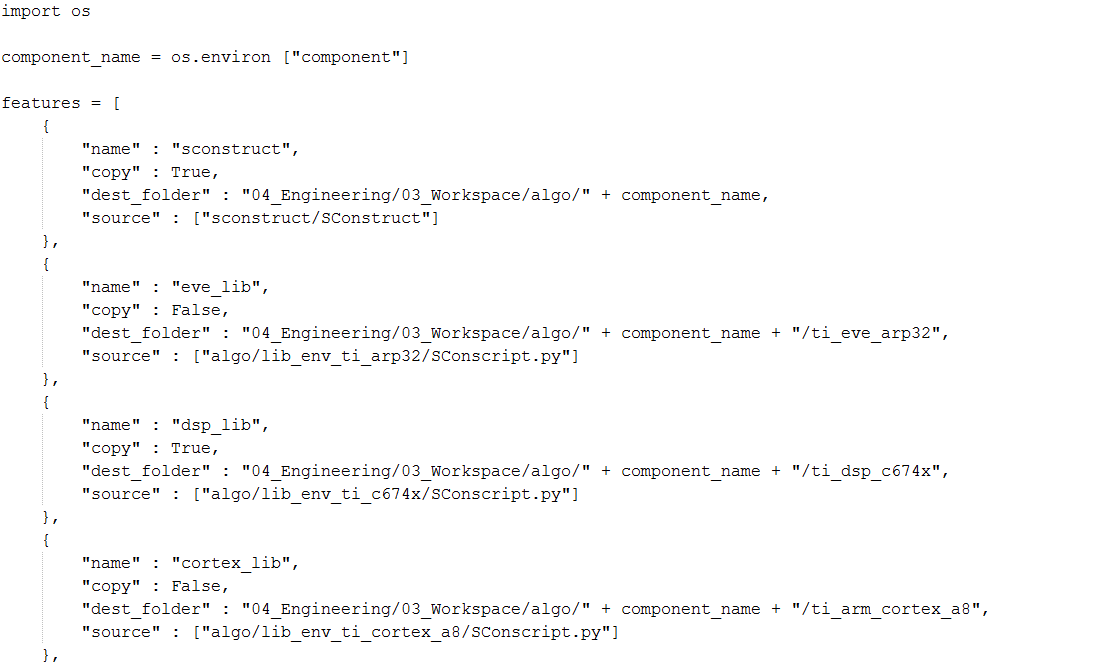
target\_name = algo\_name + "\_c674x"

## Implementing GenericScons for a new algo component

1. Add/configure shared subproject such as the shared tools in 02\_Development\_Tools.
2. The .scfg files (scons config files and file lists) are copied from 02\_Development\_Tools\scons\_tools\scons\_templates\ to 04\_Engineering and then perform modification according to the algo component. Component that already uses GenericScons can also be used as reference. This process is automated, in the path 02\_Development\_Tools\scons\_tools\setup which contains setup.scfg.Copy setup.scfg to the location 04\_Engineering\03\_Workspace\algo and modify with True/False according to the requirement of component and run setup.bat present in 02\_Development\_Tools\scons\_tools\setup.

|  |
| --- |
| component\_name = "xxx"  # For lib  dsp = True  if dsp:  dsp\_core\_dict = {  "c66xx" : True,  "c674x" : True  }  eve = True  if eve:  eve\_core\_dict = {  "arp32" : True  }  arm = True  if arm:  arm\_core\_dict = {  "cortex\_a8" : True,  "cortex\_a15" : True,  "cortex\_m3" : True,  "cortex\_m4" : True  }  # For simulation plugins  sim = True  if sim:  sim\_dict = {  "sim\_swc" : True,  "sim\_swc\_vis" : True,  "sim\_adapt" : True } |

1. This sconscript\_setup\_config.scfg file defines shared SConscripts that will be copied to project. Modify this scfg file according to the requirement of the component.



Location of file in 02\_Development\_Tools/scons\_tools/scons\_common\_scripts

Destination folder, where file is copied

When „True“, file is copied.

Figure 6: sconscript\_setup\_config.scfg

1. Execute **scons –c** to quickly check if there is an error such as missing SConscripts or scfg files. When there is no error, **Build succeeded** is printed in Windows console.

# Building with GenericScons

A first goal of this chapter is to give an overview of the targets implemented into GenericScons (Section 4.1). Here the dependency of the targets to the respective configuration files and its contained information is indicated. In Section 4.2 a brief manual on how to build is provided. The workflow of SCons in terms of which SConscript.py call which files is given in Section 4.4. For build settings being configured per default in GenericScons please refer to Section ???

## Build targets

### Ti-Tools extraction

GenericScons is configured such that relevant TI-Tools, which are present only as an archive, are unzipped prior to any build which will require them. This feature is completely provided by SCons and does not need any configuration from component’s side. The command

scons unzip\_tools

extracts all TI-Tools relevant for Scons.

### Algo Libraries

For each architecture and each variant a shared library is generated. Component-specific settings being used are defined in the following files:

* 01\_Source\_Code\algo\xxx\algo\_config.scfg:

general settings on sources for different libs, etc.

* 01\_Source\_Code\algo\xxx\algo\_lib\_file\_list.scfg:

source and include lists.

* 03\_Workspace\algo\xxx\<arch>\<arch>\_lib\_config.scfg:

Define variants with corresponding compiler flags, defines, etc.

* 03\_Workspace\algo\xxx\sconstruct\_config.scfg

Which architectures are used for algo, e.g. ca8\_used, eve\_used, etc.

### SDL files

SDL file is generated by using pdo.py extension. The SDL file contains measurement data description extracted from the source code.

Depending on the component, there are one or more SDL files generated by SCons:

1. SDL file for ECU
   * Generated for each of the following architectures, if present: C66xx, C674x, Cortex A8, Cortex A15
   * PDO source files are given in

04\_Engineering\01\_Source\_Code\algo\xxx\algo\_lib\_file\_list.scfg

* + The PDO option is specified in 04\_Engineering\01\_Source\_Code\algo\xxx\algo\_config.scfg
  + Apart from its location under 04\_Build release SDL files are copied to 04\_Engineering\05\_Deliverables\sdl\algo\xxx\

1. SDL file for simulation, included in the DLL as resource.
   * Generated for the pc library.
   * PDO source files are given in

04\_Engineering\01\_Source\_Code\algo\xxx\algo\_lib\_file\_list.scfg

* + The PDO option is specified in 04\_Engineering\01\_Source\_Code\algo\xxx\algo\_config.scfg
  + Another sdl is generated for a swc-dll, if the flag “generate\_sdl” is set to True in 04\_Engineering\03\_Workspace\algo\xxx\_sim\sim\_swc\_xxx\.
  + The PDO source list and the PDO option is specified in the .scfg file for the DLL in 04\_Engineering\03\_Workspace\algo\xxx\_sim\sim\_swc\_xxx\.
  + Because the SDL file is included in the DLL, it is not copied to 04\_Engineering\05\_Deliverables\.

The SDL file contains information of measfreeze size, this information is extracted after the file is generated and placed in the location 04\_Engineering\04\_Build\algo\algo\_name\core\release\ as a .txt file.

The PDO tool only supports C files.

More information about PDO can be found in “PDO SW Manual”.

### Algo Fingerprint

For checking the version of the library, md5 checksum is generated from source files of the library. Python script fingerprint.py is used to generate this checksum value and the value of the checksum is stored in an array variable in a C header file. This action is performed by the shared 03\_Workspace\algo\xxx\SConscript\_fingerprint.py, whose core code is depicted here:

# SConscript to build fingerprint

#-------------------------

# import

#-------------------------

Import("algo\_name", "component\_name", "eve\_used")

import os

#-------------------------

# read configuration file

#-------------------------

execfile(File("#../../../01\_Source\_Code/algo/" + component\_name + "/algo\_config.scfg").abspath) # contains all necessary information on algo sources

#-------------------------

# setup environment

#-------------------------

fp\_env = Environment(# list of scons ADAS extensions used

tools = ["fingerprint"],

# path to scons ADAS extensions

toolpath = [scons\_adas\_extensions\_path])

#-----------------------------

# generate algo fingerprint

#-----------------------------

fingerprint\_file = []

fingerprint\_src = []

# import version history

execfile(File("#/" + algo\_name + "\_algo\_ver\_history.scfg").abspath)

# get absolute include paths

fingerprint\_src += add\_dirpath(c674x\_src, "#../../../01\_Source\_Code/algo/" + algo\_name)

# get all header files in include directories

fingerprint\_src += find\_all\_h\_files(c674x\_fp\_path)

# remove version file and the fingerprint file from source list

fingerprint\_src\_list = filter(lambda x:

algo\_name + "\_ver.h" not in x and

algo\_name.upper() + "AlgoFingerprint.h" not in x, fingerprint\_src)

# generate XXXAlgoFingerprint.h

fingerprint\_file += fp\_env.Fingerprint(

# name of the header file

build\_dir + "/algo/" + algo\_name + "/fingerprint/" + algo\_name.upper() + "AlgoFingerprint.h",

# source file containing PDO tags

fingerprint\_src\_list + [Value({"ul\_FP\_" + algo\_name.upper() + "Algo" : Versions})],

FINGERPRINT\_VARS={"ul\_FP\_PdLaneInfo":[0], "ul\_FP\_" + algo\_name.upper() + "Algo":None})

If EVE is used a corresponding version of the fingerprint for arp sources is generated as well.

Let us only mention that the component-specific configuration files here are represented by 03\_Workspace\algo\xxx\xxx\_algo\_ver\_history.scfg and 03\_Workspace\algo\xxx\xxx\_arp\_ver\_history.scfg and refer to scons\_tools\docs\Fingerprint\_manual.docx for more information on the fingerprint mechanism.

### PC Simulation Targets

Each pc simulation component is located in a separate folder under 03\_Workspace\algo\xxx\_sim. Every pc simulation target requires a SConscript\_simenv.py and a SConscript.py in the respective folder. While the former script evaluates several configuration files and scons extensions in order to prepare the construction environment for the build, the latter essentially only executes the build command. The separation of the build in these two scripts allows using the same SConscript\_simenv.py for all pc simulation targets, i.e. there is only one shared SConscript\_simenv.py in GenericScons, but five SConscript.py’s to choose from depending on the target.

Every pc simulation target then requires also three .scfg-files, see end of paragraph 3.2.1.

The list of pc simulation components being built is defined in the variable “sim\_proj\_list” in 03\_Workspace\algo\xxx\sconstruct\_config.scfg.

*Remarks*:

* Certain components maintain simulation targets which are not shared in the project, but still need to be built within the component. Those targets are usually referred to as “utilities” by the component team. Those utilities targets are specified in the variable “utils\_proj\_list” and “utils64\_proj\_list” depending whether it is a 32bit- or a 64bit-target.

#### Generate RTA XML

Out of all pc simulation plugins, swc simulation plugin does an extra function. This function extract the enum definitions from the header files. The enum definitions in the header will be in a standard format. It generates xml file(RTA-xml) in the location 05\_Deliverables\cfg\algo . RTA-xml contains all the enum definitions in xml format extracted from header files. This function requires information to act and they are defined in the configuration file sim\_swc\_config.scfg particular to swc plugin.

Template is present in the location 02\_Development\_Tools\scons\_tools\scons\_templates\03\_Workspace\algo\xxx\_sim\sim\_swc\_xxx\sim\_swc\_config.scfg. Example shows the variables defined in configuration file

Example:

|  |
| --- |
| #-----------------------------  # variables - generate RTA xml file  #-----------------------------  # generate RTA xml file (default: False)  analyse\_rta = False  # Path to the files which contain the enum definition  sourceList = []  # List of enum variables  enumList = []  # List of patterns  patternList = []  # Choose the path of the generated xml file (default: "#../../../05\_Deliverables/cfg/algo/" + component\_name)  #xmlDestinationPath = ""  # if component has more then one algo. Select the algo name for the target (default: component\_name)  #rta\_target\_algo\_name = "sr"  #-----------------------------  # upto here  #----------------------------- |

#### Component specific copy operations

Some components want extra copy operation in different destination paths from the generic destination paths defined in GenericScons. This is found in PC simulation plugins and utilities for which they need to define some extra fields in simenv\_config.scfg file as shown below.

Example:

|  |
| --- |
| # For special copy operation (default: False)  special\_copy\_operation = False  # If only copy operation is required use the first dictionary set the flag(default: False) to true and give the destination path  # If copy and rename operation is required use the second dictionary set the flag(default: False) to true, set the destination path, set the new name and set the variant.  special\_copy\_dict = [  {'copy\_only' : False, 'dest' : ""},  {'copy\_rename' : False, 'dest' : "", 'replace\_name' : '', 'variant' : ''}  ] |

### Generate doxygen documentation

To generate doxygen HTML documentation, doxygen extension is used. This extension needs the doxyfile configuration as the input.

Example:

# setup environment and specify doxygen as one of the tool

pc\_env = Environment(# list of scons extensions used

tools = ["msvc", "msvs-patched", "mslib", "mslink",

"msvc-addon", "**doxygen**", "fingerprint"],

# path to scons extensions

toolpath = [scons\_adas\_extensions\_path],

MSVS\_USE\_MFC\_DIRS = 1,

TARGET\_ARCH = "X86",

# path to doxygen tool

DOXYGEN\_TOOLS = doxygen\_dir,

# manifest files need to be included in the dlls/apps

WINDOWS\_EMBED\_MANIFEST = True,

# use Visual Studio 2005

MSVC\_VERSION = "8.0",

# specify processor for scons build

BUILD\_TARGET = "SIM")

if generate\_doxygen:

vars()[component\_name + "\_" + scons\_doxygen\_target\_name] = pc\_env.Doxygen(**doxy\_file\_name**)

# restore image

pc\_env.AddPostAction(vars()[component\_name + "\_" + scons\_doxygen\_target\_name], Copy(Dir(os.path.join("doxygen", "html")).abspath, File(os.path.join("doxygen", "style", "conti.png")).abspath))

Alias(component\_name + "\_" + scons\_doxygen\_target\_name, vars()[component\_name + "\_" + scons\_doxygen\_target\_name])

Alias("all", vars()[component\_name + "\_" + scons\_doxygen\_target\_name])

### Evalboard out-files

Apart from the targets to build for cores ARP32, ARM Cortex A8, ARM Cortex A15, C66XX, C674X, M3 and M4 a new target is introduced which is called as ARP32 miniapp,This miniapp

Will build for core ARP32 and it will be use for simulation of ARP32 in both VME and VH28. This miniapp is a standalone app used only for simulation of ARP32. To enable this feature, make eve\_hil\_miniapp\_used = True(default: False) in sconscript\_config.scfg file and also make eve\_hil\_miniapp = True (default: False) in hil\_env\_config.scfg in the location “04\_Engineering\03\_Workspace\algo\cipp\_sim\cipp\_evm\_hil\vme\evm\_hil\_projects\arp32\_miniapp”and same for VH28 if enabled.

### Generate Microsoft Visual C++ project

Two versions of visual studio solutions are supported by GenericScons. There is one major difference between those two. The one that is currently build by default, allows in addition to the build of pc simulation targets also the build of ecu targets, while the former one integrates only the pc simulation targets.

The difference of the two in terms of usage of SConscipt.py-files is the following:

* VS solution of **GenericScons-Release ≤ 1.3.0** requires a component-specific **non-shared** 03\_Workspace\algo\xxx\_sim\SConscript.py, where a template can be found in scons\_tools\scons\_templates\03\_Workspace\algo\xxx\_sim.
* VS solution of **GenericScons Release ≥ 1.3.1** utilizes the shared script 03\_Workspace\algo\xxx\_sim\SConscript\_sim.py, whose copy action needs to be activated in 03\_Workspace\algo\xxx\sconstruct\_config.scfg.

#### GenericScons-Release ≤ 1.3.0

GenericScons generates Visual C++ project for each DLL project and one or more Visual Studio Solutions. The following example shows how to build Visual C++ project.

Example:

#-------------------------

# build msvc project

#-------------------------

if not skip\_ide\_generation:

ide\_scons\_cmd = (r"cd $SCONS\_DIR && scons.bat -Q -j " +

str(thread\_number) + r" $SCONS\_ARGUMENTS")

# get source codes, build result, cpppath from all variants

tsrcs = []

tbld = []

tcpppath = []

for variant in variant\_list:

if generate\_algo\_lib:

tsrcs.extend(lib\_srcs[variant[0]])

tbld.extend(lib\_build[variant[0]])

tcpppath.extend(lib\_cppp[variant[0]])

tsrcs.extend(sim\_srcs[variant[0]])

tbld.extend(sim\_build[variant[0]])

tcpppath.extend(sim\_cppp[variant[0]])

# get all source files

tsrcs = unique(map(lambda x: "" + x, tsrcs))

# get all include files

tincs = find\_all\_incs(env, tbld)

tincs = unique(tincs)

# get all variant names

variant\_names = [variant[0] for variant in variant\_list]

project\_name = (build\_dir + "/algo/" + algo\_name\_for\_build\_folder + "\_sim/pc/" +

build\_folder\_name + "/" + vc\_project\_name + env.subst("$MSVSPROJECTSUFFIX"))

sim\_ide = env.MSVSProject(

# specify project name

target = project\_name,

# add source files

srcs = tsrcs,

# add include paths

incs = tincs,

# add build targets

buildtarget = [algo\_name + "\_" + scons\_target\_name + "\_" + variant[0] for variant in variant\_list],

# add target variant

variant = variant\_names,

# don't generate .sln

auto\_build\_solution = 0,

# scons build command

MSVSSCONS = ide\_scons\_cmd,

# folder containing SConstruct

SCONS\_DIR = Dir("#").abspath,

# scons command argument

SCONS\_ARGUMENTS = "skip\_ide\_generation=1" + " " + "build\_from\_ide=1" + " " + scons\_arguments,

# add CPPPATH

CPPPATH = unique(tcpppath))

if algo\_name != component\_name:

Alias(algo\_name + "\_ide\_msvc\_" + scons\_target\_name, sim\_ide)

Alias(algo\_name + "\_ide\_msvc\_sim\_swc", sim\_ide)

Alias(algo\_name + "\_ide\_msvc", sim\_ide)

Alias(algo\_name + "\_ide", sim\_ide)

Alias(component\_name + "\_ide\_msvc\_" + scons\_target\_name, sim\_ide)

Alias(component\_name + "\_ide\_msvc\_sim\_swc", sim\_ide)

Alias(component\_name + "\_ide\_msvc", sim\_ide)

Alias(component\_name + "\_ide", sim\_ide)

Alias("all", sim\_ide)

The vs solution then looks as follows as in the case of PFC:

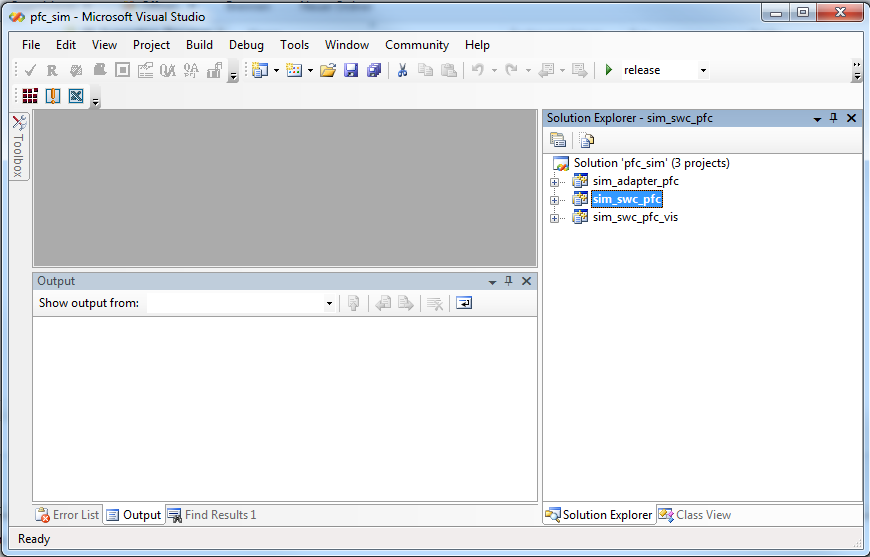


Figure 7: Visual Studio Solution of GenericScons ≤ 01.03.01

The solution explorer contains all simulation plugins and the variant of interest can be chosen in the configuration manager.

#### GenericScons Release ≥ 1.3.1

GenericScons creates only one Visual Studio project with all sources and includes for all targets except of hil related matter. To this end the xxx\_sim/SConscript.py, which used to be component-specific has been made shared. More precisely, it is replaced by the shared script SConscript\_sim.py. In order to gather all information for all targets for the visual studio project a certain

update\_vs\_data()

Is called after most SConscript-call in SConstruct, e.g.:

if c674x\_used:

result = SConscript("ti\_dsp\_c674x/SConscript.py",

exports = {"component\_name" : component\_name, "algo\_name" : algo\_name})

if result and not skip\_ide\_generation:

[tblds,tsrcs,tcppp,talias,tcppdefs] = update\_vs\_data(tblds, tsrcs, tcppp, talias, tcppdefs,

result[0],result[1],result[2],result[3],result[4])

In the shared xxx\_sim\SConscript.py this is used to generate only one Visual Studio Project:

**# build ms visual studio solution**

**if not skip\_ide\_generation:**

**# scons command for visual studio**

**ide\_scons\_cmd = (r"cd $SCONS\_DIR && scons.bat -Q -j " + str(thread\_number) + r" $SCONS\_ARGUMENTS")**

**# sed filter for converting dsp/gcc diagnostic messages to VS format for easy double clicking and jumping to error**

**diagnostics\_sed\_filter\_pipe\_cmd = r"2>&1 | ..\..\..\02\_Development\_Tools\scons\_tools\gnutools\sed.exe -e " + '"' + r"s/:\([1-9]\+\):[1-9]\+:/(\1) : /" + '"' + " -e " + '"' + r"s/, line \([0-9]\+\):/(\1) : /" + '"'**

**# build visual studio project**

**sln = pc\_env.MSVSProject(target = (build\_dir + "/algo/" + component\_name + "\_sim/pc/" +**

**component\_name + pc\_env.subst("$MSVSPROJECTSUFFIX")),**

**# define sources**

**srcs = tsrcs,**

**incs = tincs,**

**# define buildtargets, variants, and corr. define sets**

**buildtarget = talias,**

**variant = talias,**

**cppdefs = tcppdefs,**

**# define additional cmd arguments**

**cmdargs = diagnostics\_sed\_filter\_pipe\_cmd,**

**# define scons command**

**MSVSSCONS = ide\_scons\_cmd,**

**# folder containing SConstruct**

**SCONS\_DIR = Dir("#").abspath,**

**# scons command argument**

**SCONS\_ARGUMENTS="skip\_ide\_generation=1"+" "+"build\_from\_ide=1"+ " "+scons\_arguments,**

**# add CPPPATH**

**CPPPATH = unique(tcppp)**

**)**

Those targets which are forwarded to the IDE can be extended via 03\_Workspace\alg\xxx\sconstruct\_config.scfg.

The related visual studio solution then looks as follows:

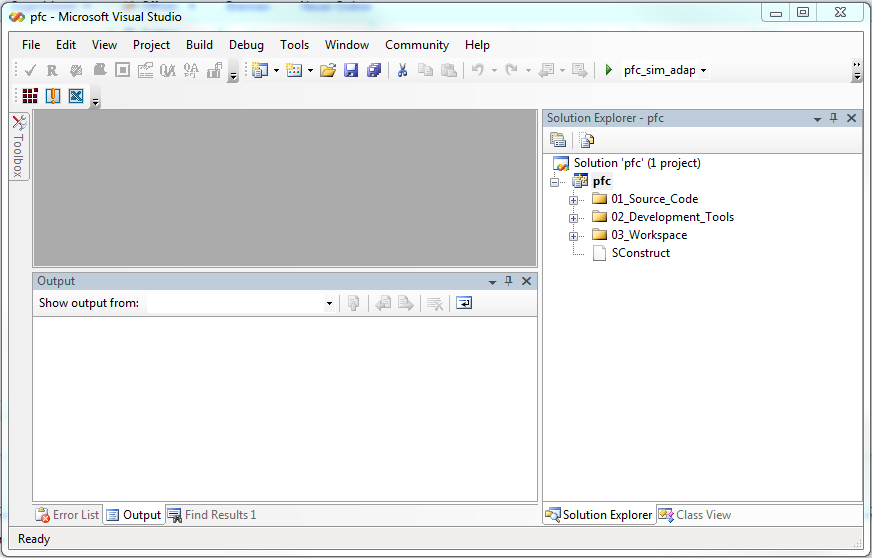


Figure 8: Visual Studio Solution of GenericScons ≥ 01.03.01

Here only one project appears in the solution explorer and the targets of interest are set in the configuration manager.

**Macro Definitions ( ≥ AL\_ETK\_SCT\_01.07.00 )**

According to the activated build target certain code locations are greyed out. These are exactly those locations which have been switched off by macro definitions in the build settings. This feature works correctly for all single build targets, if the component uses exclusively shared SConscript-files provided by the GenericScons package. In this case the macros being used for target build can be viewed in the project properties --> NMake for each single build target. However, if a define set requires a modification, this has to be done in the **corresponding .scfg-file**. A change in the visual studio solution will not have any effect on the build!!

*Remark:* The use of the former xxx\_sim\SConscript.py is still supported in case the component prefers the visual studio solution described in 4.1.7.1. In this case it needs to be made sure that this component-specific xxx\_sim\SConscript.py still exists and copying SConscript\_sim.py to xxx\_sim is deactivated. Then SConstruct will execute the former script in order to build simulation targets and the visual studio solution.

#### Build from Command Line and from Visual Studio – Differences

There are minor differences for the build from command line and from visual studio. The reason lies in the fact that the Visual Studio Solution adds certain GenericScons option (see Chapter 5) to the scons command. There are two kinds of different behaviors:

1. **Parsing Sconscript.py-files**

Since the build targets integrated into Visual Studio are not related to the generation of out-files for the evalboard, no SConscript.py being connected with those targets is read from Visual Studio. This is done for optimization, more precisely, parsing time of SConscripts is – depending on the project – reduced by up to 10-20%.

Neglecting those SConscripts is accomplished by the Flag “build\_from\_vs”, see Section .

1. **Installation of Shared Libraries for PC Simulation to 05\_Deliverables and 05\_Testing**

When building shared libraries from command line, per default release versions of those libraries are installed to 05\_Deliverables and to 05\_Testing\06\_Test\_Tools\mts\_measurement\dll\algo\xxx\_sim. The behavior from Visual Studio is a little trickier:

* If a release version of a shared library is built using the corresponding Alias, e.g. xxx\_sim\_swc\_release”, then per default the corresponding release version is installed to 05\_Deliverables and to 05\_Testing\06\_Test\_Tools\mts\_measurement\dll\algo\xxx\_sim, so here the same behavior applies as for the build per command line.
* If a non-release version is built is the corresponding alias, e.g. xxx\_sim\_swc\_debug, then the debug version of this Llibrary is installed to 05\_Testing\06\_Test\_Tools\mts\_measurement\dll\algo\xxx\_sim, but not to 05\_Deliverables. Note that in case of the build from command line, per default the shared library is not installed to any of those folders, if it is not a release version.
* If all variants are built for some simulation target with the corresponding alias, e.g. xxx\_sim\_swc, then **nothing** is installed to 05\_Deliverables or 05\_Testing.

The just described behavior is controlled by the flag “build\_from\_ide”, see Section 5.1.3.

Let us stress at this stage though, that this differing behavior does not affect the actual build targets, more precisely, if a build target can be generated from command line and from visual studio, the outcome is the same in both cases!

**4.1.8.4 Generate Microsoft Visual Studio 2010 solution**

GenericScons also generates visual studio 2010 solution same as visual studio 2005 solution on building target xxx\_ide\_msvc with one project and all the necessary targets in the visual studio 2010 solution explorer. To generate visual studio 2010 project mention generate\_vs2010 = True in the config file sconstruct\_config.scfg. To generate visual studio 2005 project mention generate\_vs2010 = False in the config file sconstruct\_config.scfg.

A new script file msvs-patched2010.py has been introduced in scons\_adas\_extensions to generate three files .sln, .vcxproj, vcxproj.filters files required for visual studio 2010 project.

### Generate CCS 5 Eclipse project

GenericScons generates CCS 5 Eclipse project for each library and also for each out file. The following example shows how to build CCS 5 project.

Example:

#-------------------------

# generate ccs 5.3 project

#-------------------------

# get all variant names

variant\_names = [variant[0] for variant in variant\_list]

if not skip\_ide\_generation:

# add include paths

dsp\_env.Append(CPPPATH=result[2])

ide\_ccs\_algo\_ti\_c674x = dsp\_env.EclipseProject(

# specify project name

target\_name = algo\_name.upper() + "\_TI\_C674x\_Lib",

# specify folder for storing the project

target\_dir = build\_dir + "/algo/" + algo\_name + "/ti\_c674x/",

# specify build variants

variants = variant\_names,

# specify preprocessor defines

defines = [define\_list + variant[2] for variant in variant\_list],

# target\_type : "lib" or "out"

target\_type = "lib",

# core : "C6000" or "TMS470"

core = "C6000",

# specify compiler version

code\_gen\_version = get\_c6000\_codegen\_version(Dir(c6000\_code\_gen\_tool\_dir).abspath),

# list of target variants (use the same order as in variants)

scons\_target = [algo\_name + "\_algo\_ti\_c674x\_" + variant[0] for variant in variant\_list],

# specify additional argument when running scons

scons\_argument = "-j" + str(thread\_number) + " " + "skip\_ide\_generation=1" + " " + scons\_arguments,

# add source files

linked\_resources = result[1],

additional\_natures = "")

Alias(component\_name + "\_ide\_ccs\_algo\_ti\_c674x", ide\_ccs\_algo\_ti\_c674x)

Alias(component\_name + "\_ide\_ccs", ide\_ccs\_algo\_ti\_c674x)

Alias(component\_name + "\_ide", ide\_ccs\_algo\_ti\_c674x)

if algo\_name != component\_name:

Alias(algo\_name + "\_ide\_ccs\_algo\_ti\_c674x", ide\_ccs\_algo\_ti\_c674x)

Alias(algo\_name + "\_ide\_ccs", ide\_ccs\_algo\_ti\_c674x)

Alias(algo\_name + "\_ide", ide\_ccs\_algo\_ti\_c674x)

Alias("all", ide\_ccs\_algo\_ti\_c674x)

Similarly to the case of the Visual Studio solution, there is a minor difference between the build of ecu libraries or HiL-out-Files when building from command line and from CCS. Since all of those targets are not related to any pc simulation targets, no SConscript.py being related to any of those targets is read when scons is triggered from CCS. Note that, build results are not affected by this! This is done for optimization and depending on the project time for parsing SConscript.py-files is reduced by up to 30%.

This behavior is controlled by the GenericScons option “build\_from\_ccs”, see also Section 5.1.5.

## How to build

### Via command line

The following figure shows build targets of GenericScons and its hierarchy. The number of build targets depends on the algo component. Different components may have different build targets. The components maintain their individual targets list in 03\_Workspace\algo\xxx\scons.bat.

Recall that xxx, as usual, stands for the component/algo name

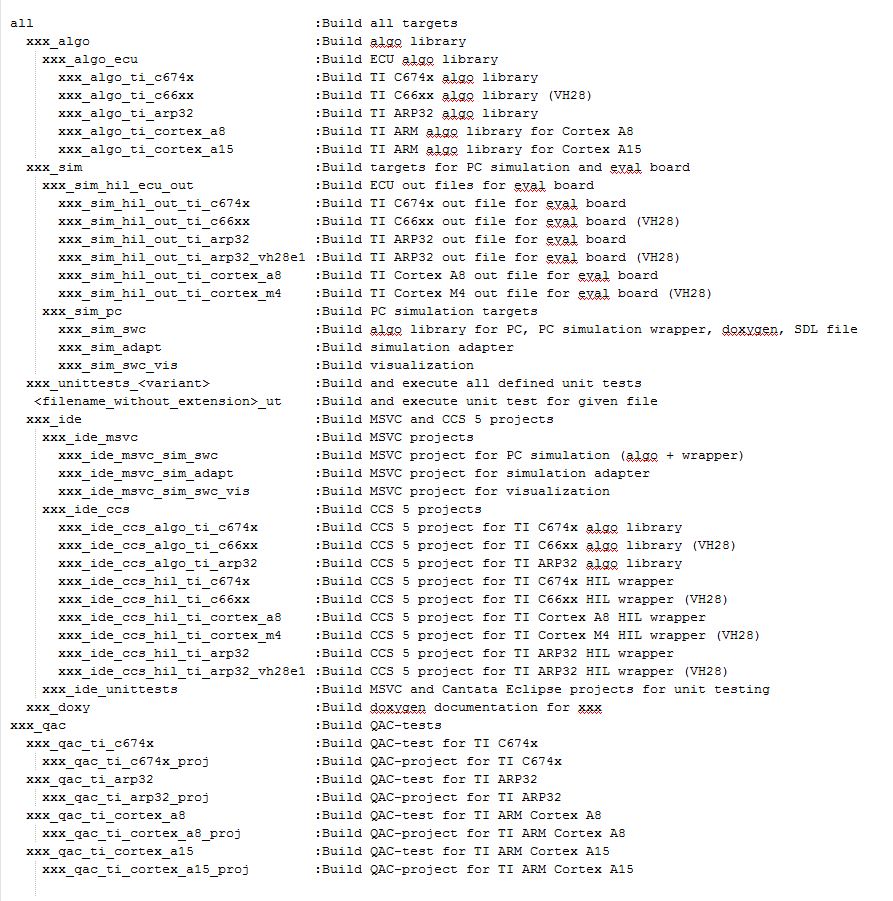


Figure 9: List of build targets

Calling “**scons all**” or “**scons**” from 04\_Engineering\03\_Workspace\algo\xxx\ folder will build all targets below **all** (**xxx\_algo, xxx\_sim, xxx\_ide and xxx\_doxy**). To clean all targets, execute “**scons –c**” command. SCons command must be invoked from 04\_Engineering\03\_Workspace\algo\xxx\ because this folder contains **scons.bat** file, which specifies where to look for movable python, and also contains SConstruct file which is the main SConscript file that are read by SCons.

It is worth mentioning that all targets can be called for every variant which is available, e.g. xxx\_algo\_release, xxx\_algo\_ecu\_release, etc., are usable as well.

The following table shows options available and how to use them:

|  |  |
| --- | --- |
| Show help information (available targets, option). | **scons -h** |
| Build all. | **scons** |
| Build all and show full command line | **scons verbose=1** |
| Clean all. | **scons -c** |
| Build all with multi-thread (2 threads). | **scons -j2** |
| Build all with certain C preprocessor define. The C preprocessor define will be used when building libs, DLLs, out files. | **scons define=DEFINE1, DEFINE2** |
| Build all with certain C preprocessor define using 2 threads. | **scons –j2 define=DEFINE1, DEFINE2** |
| Build a target. | **scons (target\_name)**, ex: **scons ped\_algo\_ti\_c674x** |
| Build a target and show full command line. | **scons verbose=1 (target\_name)** |
| Clean a target. | **scons -c (target\_name)**, ex: **scons -c ped\_algo\_ti\_c674x** |
| Build a target with certain variant. | **scons (target\_name)\_(variant)**, ex: **scons ped\_algo\_ti\_c674x\_release** |
| Build all IDEs. | **scons (component/algo)\_ide**, ex: **scons ped\_ide** |

Table : Executing GenericScons

Remarks

|  |  |
| --- | --- |
| Build doxygen. | **scons (target\_name)**, ex: **scons ped\_doxy** |

1. When building with GenericScons the following three phases are passed always:
   1. **Copy** all required shared scripts to the correct locations (see 03\_Workspace\algo\xxx\sconscript\_setup\_config.scfg.).
   2. **Scan** all related Scons-scripts and configs, more precisely:
      1. SCons looks for SConstruct which is the main script.
      2. SConstruct reads required .scfg file and calls SConscript.py to set up the environment for a target.
      3. SConscript.py reads required .scfg files (e.g. file and include lists) and builds the target.
      4. Scons sets up dependency tree and prepares build according to those dependencies.
   3. **Build** required targets/all.

Note that b) does not build anything! Stage iv. generally consumes most of the time: 20-40 seconds are not uncommon depending on the size of the algo project

1. The preceding remark implies that SCons reads all SConscript files before starts to build. Therefore when “print” is used in the SConscript, the “print” is executed before the build process actually starts.
2. SCons creates the same folder structure in build folder as in the source folder, e.g.: 04\_Build\algo\xxx\ti\_arp32\release\01\_Source\_Code\algo\xxx.
3. When variant is used (release, debug, etc), build result of each variant must be in different folder. Otherwise SCons gives error message because the build result of one variant overwrites the other variant.
4. When multi-thread build is used, the build log file might look out of order because of multi thread.
5. The SCons build log is stored in 04\_Engineering\03\_Workspace\algo\xxx\sconsbuild.log.
6. SCons supports more than one algos in a component. When there are multiple algos, the target\_name starts with algo name instead of component name. For example: in GB component, there are cb and scb algos. Execute **scons cb\_algo** to build cb algo or **scons scb\_algo** to build scb algo. To build both algos, execute **scons gb\_algo**.
7. When 04\_Engineering\03\_Workspace\algo\xxx\.sconsign.dblite file or 04\_Engineering\04\_Build folder is deleted, SCons will do rebuild when it is executed.
8. SCons looks for 03\_Workspace\algo\xxx\sconscript\_setup\_config.scfg to know which SConscripts should be copied from 02\_Development\_Tools\scons\_tools\scons\_common\_scripts to the project before it starts the build process. If the project has non-shared SConscript , this non-shared SConscript should not be listed in this config file to prevent it being overwritten or the "copy" attribute should be set to False.
9. Cantata unit test is integrated in SCons. For further information, please refer to “ETK SCT SCons Cantata Integration.docx”.

### Via IDE

For more information on how to build from Visual Studio or from Code Composer Studio, please refer to Sections 4.1.7and 4.1.8.

## Sysbios library

The HIL wrapper out files of C674x, Cortex A8 (VISION MID EVE), and Cortex M4, Cortex A15 (VISION HIGH 28) use custom sysbios (= TI operating system) library which must be created before the out file is created. The sysbios library is configured using .cfg file in C674x\ folder or CortexA8\ folder. To shorten the build time, Shared-SCons will check if the sysbios library is already created or not. If Sysbios library is already created, GenericSCons will directly build the out file. To force rebuilding Sysbios library, execute **scons –c “hil\_wrapper\_target”** to delete the Sysbios library and then build the out file using **scons “hil\_wrapper\_target” (**replace hil\_wrapper\_target with the target name listed in scons help).

## Visualization of SCons workflow

### Building ecu libraries and CCS 5 projects

The visualisazion here is restricted to the C674x library and its ccs project. The cases of ARP32, ARM Cortex A8, ARM Cortex A15 work in analogy.

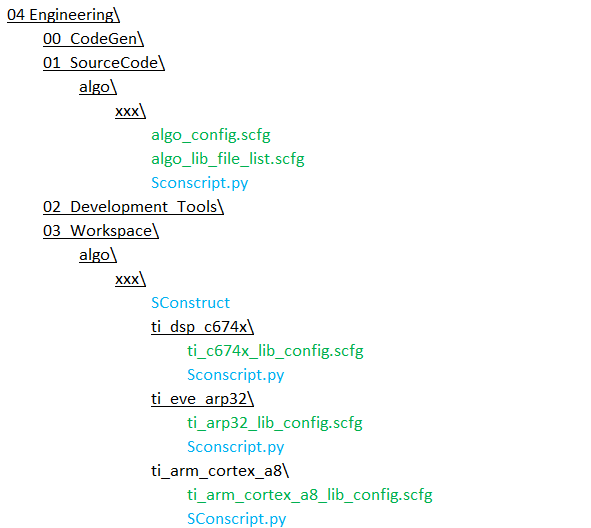


Figure 10: Build of ecu libraries - Step 1

SConstruct calls SConscript.py in 03\_Workspace\algo\xxx\ti\_dsp\_c674x\ which sets up the build environment for building c674x algo library. Compiler options and defines are defined in ti\_c67x\_lib\_config.scfg.

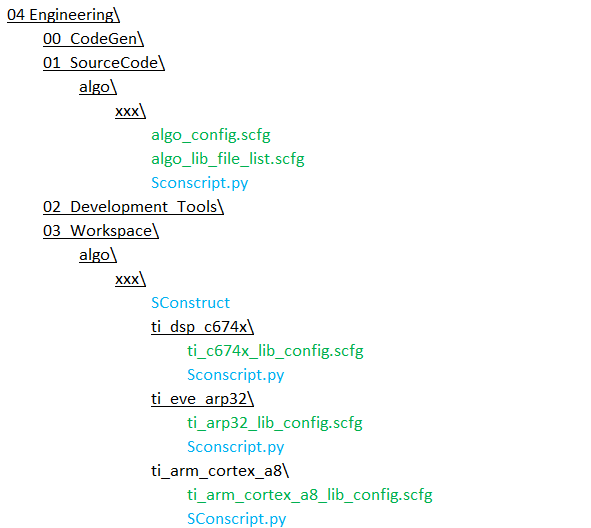


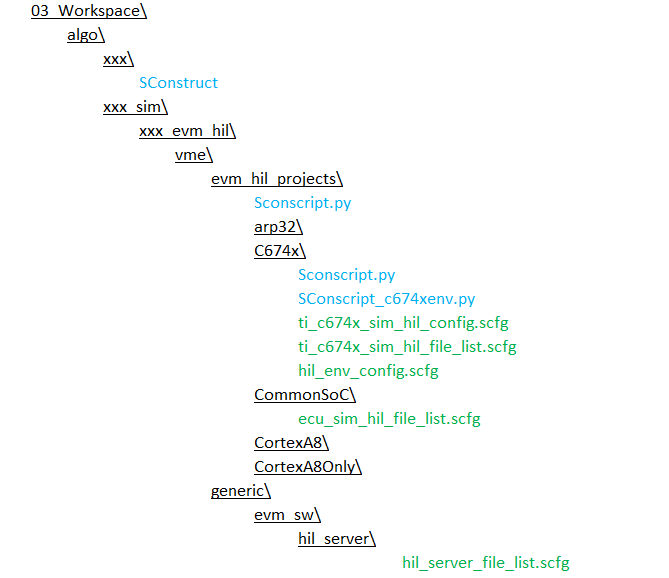
Figure 11: Build of ecu libraries - Step 2

This SConscript.py then calls SConscript.py in 01\_SourceCode\algo\xxx\ to actually build the algo library in the required variants. SConscript.py in 01\_SourceCode\algo\xxx\ gets the required source files and include paths by reading the content of algo\_lib\_file\_list.scfg and algo\_config.scfg.

After algo library is built, they are passed together with the list of source files, include paths and defines back to the caller SConscript.py (in 03\_Workspace\algo\xxx\ti\_dsp\_c674x\). CCS 5 project is then built using this information.

### Building HIL simulation out file and CCS 5 project

As in paragraph 394.4.1, we restrict ourselves to the case of C674x out files for VISION MID EVE (folder “vme”). The situation for the remaining out files as well as the case of VISION HIGH 28 (folder “vh28”) are very similar



2

1

Figure 12: Build of HiL simulation out – Step 1

SConstruct calls SConscript.py in 03\_Workspace\algo\xxx\_sim\xxx\_evm\_hil\vme\evm\_hil\_projects\ which then calls SConscript\_c674xenv.py in 03\_Workspace\algo\xxx\_sim\xxx\_evm\_hil\vme\evm\_hil\_projects\C674x\ to create build environment for building c674x simulation out file.

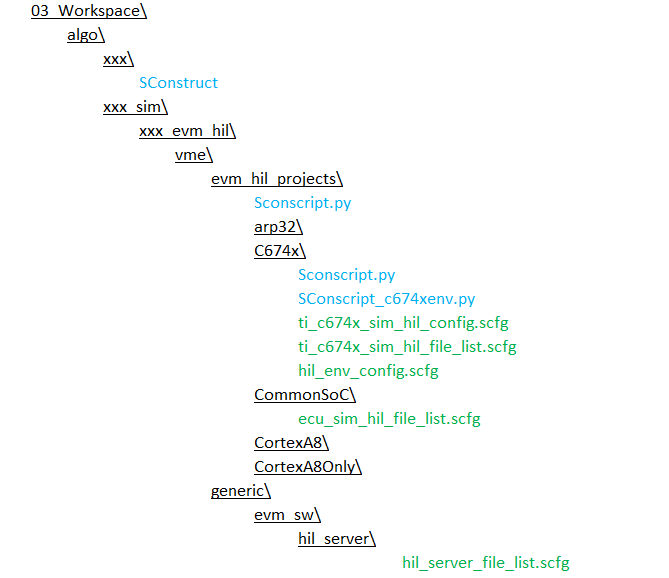


Figure 13: Build of HiL Simulation out - Step 2

SConscript\_c674xenv.py calls SConscript.py in the same folder (03\_Workspace\algo\xxx\_sim\xxx\_evm\_hil\vme\evm\_hil\_projects\C674x\) to build simulation output file release and debug. The required compiler options, defines, source files and include paths are read from hil\_env\_config.scfg, ti\_c674x\_sim\_hil\_file\_list.scfg, ti\_c674x\_sim\_hil\_config.scfg, ecu\_sim\_hil\_file\_list.scfg, algo\_lib\_file\_list.scfg, sim\_swc\_cipp\_file\_list.cfg and hil\_server\_file\_list.scfg.

Return out files, sources, include paths

After simulations out files are built, they are passed together with the list of source files, include paths and defines back to caller SConscript.py. CCS 5 project is then built using these files.

### MACRO defined in CCS 5 project for ecu libraries and HIL simulation

The following tables contain complete lists of MACRO define in CCS 5 project for ecu libraries and HIL simulation.

|  |  |  |
| --- | --- | --- |
| **ARM** | **ARP32** | **C6600** |
| \_\_TI\_ ARM\_V7A8\_\_ | \_\_ARP32\_\_ | \_\_TI\_EABI\_\_ |
| \_\_TI\_EABI\_SUPPORT\_\_ | \_\_TI\_EABI\_\_ | \_TMS320C6X |
| \_\_TI\_ ARM\_V7\_\_ |  | \_TMS320C6400 |
| \_\_TI\_ ARM\_\_ |  | \_TMS320C6400\_PLUS |
|  |  | \_TMS320C6600 |
|  |  | \_TMS320C6700 |
|  |  | \_TMS320C6700\_PLUS |
|  |  | \_TMS320C6740 |

Table 4: MACRO defined for CCS 5 project

### Building PC simulation wrapper and Microsoft Visual C++ project

A co mponent may build arbitrarily many pc simulation targets, mostly shared libraries. We only sketch the build of one such target. Note, however, that depending on the type of target, the naming of the required .scfg-files may vary, see paragraph 3.3.2.

Figure 14: Build of PC simulation wrapper - Step 1



2

1

SConstruct calls SConscript.py (resp. SConscript\_sim.py) in 03\_Workspace\algo\xxx\_sim\ which then calls SConscript.py in 03\_Workspace\algo\xxx\_sim\sim\_swc\_xxx\ to create build environment for building PC algo library and simulation wrapper DLL file.

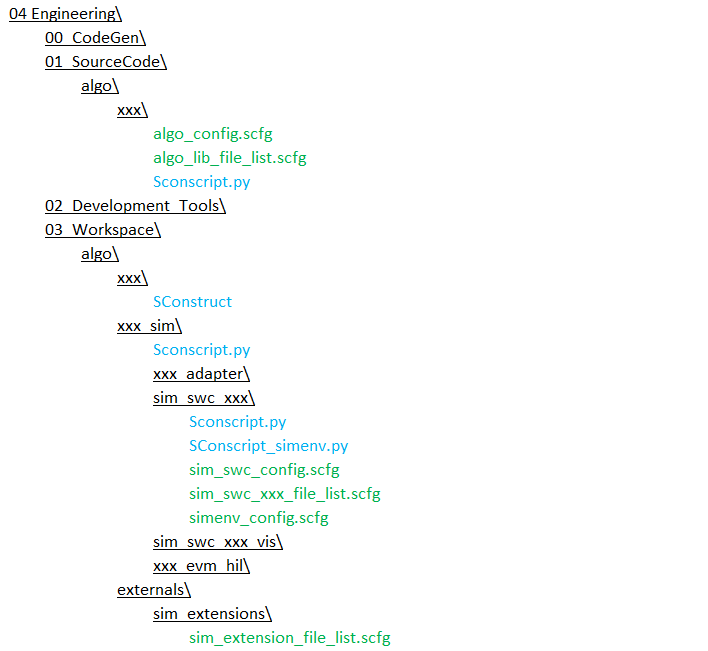


Figure 15: Build of PC simulation wrapper - Step 2

SConscript.py in 01\_SourceCode\algo\cipp\ is then called to build PC algo library. This library is used later to build simulation wrapper DLL file.



Figure 16: Build of PC simulation wrapper - Step 3

SConscript.py in 03\_Workspace\algo\xxx\_sim\sim\_swc\_xxx\ is then called to build simulation wrapper DLL file. The required compiler options, defines, source files and include paths are read from simenv\_config.scfg, sim\_swc\_config.scfg, sim\_swc\_xxx\_file\_list.scfg, externals\_include\_file\_list.scfg and sim\_extension\_file\_list.scfg.

After DLL files are built, they are passed together with the list of source files and include paths back to the caller SConscript.py. Microsoft Visual C++ project is then built using these files.

## Global Build Setting for individual targets

The following tables contain complete lists of compiler flags being set per default for all the individual targets.

|  |  |  |
| --- | --- | --- |
| ECU Libraries | Variants | Compiler Flags |
| C674x | release | -c -mv6740 --abi=eabi -o3[[2]](#footnote-2) |
|  | debug | -g -c -mv6740 --abi=eabi |
|  | other | -c -mv6740 --abi=eabi |
| C66xx | release | -c -mv6600 --abi=eabi -o3[[3]](#footnote-3) |
|  | debug | -g -c -mv6600 --abi=eabi |
|  | other | -c -mv6600 --abi=eabi |
| ARP32 | release | -kv -kh --silicon\_version=V210 -O3 --symdebug:skeletal |
|  | debug | -kv -kh --silicon\_version=v210 -g |
|  | other | -kv -kh --silicon\_version=v210 |
| Cortex A8 (gcc) | release | -O3 -Wall -fshort-enums -mabi=aapcs -mlittle-endian -mtune=cortex-a8 -mcpu=cortex-a8 -c -fno-common -mfpu=neon -mfloat-abi=hard |
|  | debug | -gstrict-dwarf -g -Wall -fshort-enums -mabi=aapcs -mlittle-endian -mtune=cortex-a8 -mcpu=cortex-a8 -c -fno-common -mfpu=neon -mfloat-abi=hard |
|  | other | -Wall -fshort-enums -mabi=aapcs -mlittle-endian -mtune=cortex-a8 -mcpu=cortex-a8 -c -fno-common -mfpu=neon -mfloat-abi=hard |
| Cortex A15 | release | -O3 -Wall -fshort-enums -mabi=aapcs -mlittle-endian -mtune=cortex-a15 -mcpu=cortex-a15 -c -fno-common -mfpu=neon -mfloat-abi=hard |
|  | debug | -gstrict-dwarf -g -Wall -fshort-enums -mabi=aapcs -mlittle-endian -mtune=cortex-a15 -mcpu=cortex-a15 -c -fno-common -mfpu=neon -mfloat-abi=hard |
|  | other | -Wall -fshort-enums -mabi=aapcs -mlittle-endian -mtune=cortex-a15 -mcpu=cortex-a15 -c -fno-common -mfpu=neon -mfloat-abi=hard |

Table 5: Default Flags for ECU Libraries

|  |  |  |
| --- | --- | --- |
| Hil out-files | Variants | Compiler Flags |
| C674x | release | -c -mv6740 --abi=eabi -o3[[4]](#footnote-4) |
|  | debug | -g -c -mv6740 --abi=eabi |
|  | other | -c -mv6740 --abi=eabi |
| C66xx | release | -c -mv6600 --abi=eabi -o3[[5]](#footnote-5) |
|  | debug | -g -c -mv6600 --abi=eabi |
|  | other | -c -mv6600 --abi=eabi |
| ARP32 | release | n/a |
|  | debug | n/a |
|  | other | n/a |
| Cortex A8 | release | n/a |
|  | debug | n/a |
|  | other | n/a |
| Cortex A15 | release | -O3 -Wall -fshort-enums -mabi=aapcs -mlittle-endian -mtune=cortex-a15 -mcpu=cortex-a15 -c -fno-common -mfpu=neon -mfloat-abi=hard |
|  | debug | -gstrict-dwarf -g -Wall -fshort-enums -mabi=aapcs -mlittle-endian -mtune=cortex-a15 -mcpu=cortex-a15 -c -fno-common -mfpu=neon -mfloat-abi=hard |
|  | other | -Wall -fshort-enums -mabi=aapcs -mlittle-endian -mtune=cortex-a15 -mcpu=cortex-a15 -c -fno-common -mfpu=neon -mfloat-abi=hard |

Table 6: Default Flags for HiL Targets

|  |  |  |  |
| --- | --- | --- | --- |
| PC Simulation | Variants | Compiler Flags | Link Flags |
|  | release | /EHsc /c /errorReport:prompt /O2 /FD /MD /nologo /W3 | /DLL /DEBUG |
|  | debug | /EHsc /c /errorReport:prompt /Od /RTC1 /MDd /nologo /W3 | /DLL /DEBUG |
|  | other | n/a | n/a |

Table 7: Default Flags for PC Simulation Targets

**Remark:** There are quite some compiler flags which specify different levels of application, e.g. for the visual studio compiler there are different warning levels /W0, … /W4. In order to prevent the compiler from receiving contradicting compiler flags, there is a compiler flags check for certain flags, which looks as follows:

1. Define the Compiler flags to be checked (done in common\_config.scfg):

# handling of multiply set contradicting compiler flags:

sim\_common\_ccflags\_to\_check = {r'\/W\d' : 'keep\_highest'}

This variable is a dictionary, where the individual keys represent the pattern to look for and the value is the way contradicting flags are handled. Besides ‘keep\_highest’ there is also ‘keep\_local’. The latter means that whenever the component specifies a flag which suits the pattern, it overrides the one set per default by GenericScons.

1. Resolve flag conflicts within the relevant SConscript with the function

resolve\_contradicting\_ccflags (from sconstruct\_helpers.py), e.g.:

# choose possibly contradicting ccflags, see comon\_config.scfg

new\_variant['CCFLAGS'] = resolve\_contradicting\_ccflags(ccflags, sim\_common\_ccflags\_to\_check)

The required compiler flags are then returned.

This functionality is provided by GenericScons, nothing is to be done from component’s side. This mechanism is also applied for optimization flags for dsp-targets, cf. common\_config.scfg and, e.g., scons\_tools\scons\_common\_scripts\algo\lib\_env\_ti\_c674x\SConscript.py.

## Built in macro of different compilers for Visual C++ project

The following tables contain complete lists of built in macro of different compilers

|  |  |  |
| --- | --- | --- |
| **ARM** | **ARP32** | **C6600** |
| \_\_TI\_ ARM\_V7A8\_\_ | \_\_ARP32\_\_ | \_\_TI\_EABI\_\_ |
| \_\_TI\_EABI\_SUPPORT\_\_ | \_\_TI\_EABI\_\_ | \_TMS320C6X |
| \_\_TI\_ ARM\_V7\_\_ |  | \_TMS320C6400 |
| \_\_TI\_ ARM\_\_ |  | \_TMS320C6400\_PLUS |
|  |  | \_TMS320C6600 |
|  |  | \_TMS320C6700 |
|  |  | \_TMS320C6700\_PLUS |
|  |  | \_TMS320C6740 |

**Table 8: Built in macro of compilers**

# GenericScons Option

## GenericScons Flags

There are certain flags for GenericScons available. If they are not specified in the build command, i.e. in “scons.bat <target>”, their respective default values are set. As a rule, those flags may be changed for the build by setting their value on the command line by

scons.bat <flag>=<value> <flag>=<value> <target>

Note that one or more flags may be specified here.

### Flag “verbose”

This flags controls the size of Scons output during the build.

* **verbose=0** (default)**:** output is restricted to which target is being built, which source file is being compiled, and which warnings occurred during build, e.g.

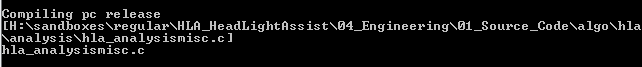


Figure 17: Non-verbose build output

* **verbose=1**: full command line output including all include paths, compiler options and defines is displayed on the command line, e.g.



Figure : Verbose build output

### Flag “skip\_ide\_generation”

This flag defines the way of handling IDE-generation in a build.

* **skip\_ide\_generation=0** (default when building via command line):

IDE generation is enabled.

* **skip\_ide\_generation=1** (default when building via CCS- or VS-IDE):

IDE generation is disabled.

### Flag “build\_from\_ide”

For PC simulation builds via command line and via Visual Studio are distinguished in terms of locations where build results are stored. In addition to the default places under 04\_Build, depending on the situation, the targets are copied to further locations:

* **build\_from\_ide=0** (default when building via command line):
  + Only release targets are copied to 05\_Deliverables and to 05\_Testing.
* **build\_from\_ide=1** (default when building via command line):
  + In case a release target is specified, e.g. “scons.bat cipp\_sim\_swc\_release”, it is copied to 05\_Deliverables and 05\_Testing.
  + In case a non-release target is specified, e.g. “scons.bat cipp\_sim\_swc\_debug”, the target is only copied to 05\_Testing.
  + If no variant is specified, e.g. “scons.bat cipp\_sim\_swc”, nothing is stored to 05\_Deliverables or to 05\_Testing.

### Flag “build\_from\_vs”

For the build from Visual Studio all the build commands for hil simulation targets are not necessary to read.

* **build\_from\_vs=0** (default when building via command line):
  + All build commands are read.
* **build\_from\_vs=1** (default when building via Visual Studio):
  + All build commands related to HiL simulation targets are not read.

### Flag “build\_from\_ccs”

For the build from Code Composer Studio all the build commands for PC simulation targets are not necessary to read.

* **build\_from\_ccs=0** (default when building via command line):
  + All build commands are read.
* **build\_from\_ccs=1** (default when building via Code Composer Studio):
  + All build commands related to PC simulation targets are not read.

### Flag “custom\_sim\_copy”

This is a custom implementation of the copy behavior of PC simulation targets to the testing folder under 05\_Testing.

* **custom\_sim\_copy=0** (default): see paragraph 5.1.3.
* **custom\_sim\_copy=1:**
  + Release versions are stored under 05\_Testing\06\_Test\_Tools\mts\_measurement\dll\algo\xxx\_sim
  + Non-release versions are stored under

05\_Testing\06\_Test\_Tools\mts\_measurement\dll\algo\xxx\_sim\variant

It is possible to generate visual studio solutions which per default build with the flag ”custom\_sim\_copy” being enabled. To this end generate the Visual Studio IDE via the command

scons.bat custom\_sim\_copy=1 xxx\_ide\_msvc

**NOTE:**

Build results under 05\_Testing\06\_Test\_Tools\mts\_measurement\dll\algo\xxx\_sim\variant must **not** be checked in to MKS!!

### Flag “profile”

There is a profiling mode available which provides information on how much time is spent parsing all the individual shared SConscript-files.

* **profile=0** (default)**:** No profiling mode.
* **profile=1:** Profiling mode switched on. Builds are performed in a standard way. In addition, after reading all SConscript.py-files, the profiling result can be found in *03\_Workspace\algo\xxx\profile.txt.*

This feature is mainly advantageous when actually developing new features for GenericScons or for improving build time. If the reading time for certain parts/commands of a SConscript-file is desired, the following blocks

if profile:

StartSconscriptEntry(<hierarchy>, <info>)

if profile:

FinishSconscriptEntry()

shall be wrapped around the parts to be measured.

The parameter <hierarchy> shall be substituted by the relative hierarchical position of the SConscript in question compared to the SConstruct file. More precisely, SConstruct-file has hierarchy=0, all SConscript-files which are called directly by SConstruct have hierarchy=1, all SConscript-files, which are called by one of the latter SConscript-files have hierarchy=2, etc. It allows a hierarchical time output in profile.txt.

The parameter <info> is optional; it allows including more information on the measured parts to the resulting profile.txt.

### Flag “-h”

#### Case of component-specific version of *scons.bat*

Component-specific scons.bat concept is changed, now there is no component-specific scons.bat file.

#### Case of shared version of *scons.bat*

A generic version of *scons.bat* can be found under *scons\_tools\scons\_templates\03\_Workspace\algo\xxx.* This file needs to be copied to

*03\_Workspace\algo\xxx* from where it willcall shared batch file present under *scons\_tools\scons\_common\_scripts\batch.*

scons.bat -h

does the following:

* **Case 1:** *03\_Workspace\algo\xxx\aliases.txt* exists and is up-to-date.

aliases.txt is displayed on the command line

* **Case 2:** *03\_Workspace\algo\xxx\aliases.txt* does **not** exist or is **not** up-to-date.

The complete GenericScons environment is read, an up-to-date version of aliases.txt is generated and displayed on the command line, nothing more is built.

### Flag “install\_to\_deliverables\_dir”

This flag supports SCons in selecting the correct variant .lib, .dll, .pdb, and .sdl to be installed from 04\_Build to 05\_Deliverables.

* Default behavior is to install variant release .lib, .dll, .pdb, and .sdl from 04\_Build to 05\_Deliverables.
* If “install\_to\_deliverables\_dir” is set with a variant name then corresponding variant .lib, .dll, .pdb, and .sdl get installed from 04\_Build to 05\_Deliverables.

scons.bat install\_to\_deliverables\_dir=<variant> <target>

**Remark:** Even if the shared version of *scons.bat* is not used, an up-to-date version of *aliases.txt* is generated during every successful build.

### 5.1.10 Flag “no\_err\_warning”

This flag displays the error and warning summary report at the end of build.log file for each scons target.

* no\_err\_warning=0: (default) No error and warning summary report.
* no\_err\_warning=1: Error and warning summary report is displayed at the end of build.log.

## GenericScons SubAlgo-Feature

Some components maintain different versions of their algorithms. More precisely, for each subalgorithm (“SubAlgo”), the source code is essentially identical, but it is read using different define-sets by the individual compilers.

### Initiation of SubAlgo-Feature

This is done in *04\_Engineering\03\_Workspace\algo\xxx\sconstruct\_config.scfg*.

Let us have a look at how this is accounted for in component GB:

# define component name

component\_name = "gb"

# list of algo name in this component, when there is more than one algo in one component

# if algo\_name\_list is not defined, the default value is the component\_name

mono\_algo\_name = "cb"

stereo\_algo\_name = "scb"

algo\_name\_list = [mono\_algo\_name, stereo\_algo\_name]

So the variables **component\_name** and **algo\_name\_list** shall be defined according the component’s needs.

*Attention:* If there are multiple SubAlgos, it is important that no **algo\_name** matches exactly the **component\_name!**

Providing these definitions makes GenericScons read all Sconscript.py-files which are called directly from SConstruct **and** which are **not** related to PC simulation targets, as many times as there are SubAlgos. That means, in the case of GB those SConscript.py-files are read twice, once for SubAlgo “CB” and once for SubAlgo “SCB”.

Concerning PC simulation targets, they have to be defined at an extra spot in *04\_Engineering\03\_Workspace\algo\xxx\sconstruct\_config.scfg.* This needs to be done according to the following example also originating from GB:

#--------------------------------------------------------------------

# define algo/component specific project lists

#--------------------------------------------------------------------

# define list of source folders for simulation targets to be built,

# Note 1: the following definitions are merely examples, please adapt according to component

# paths have to be relative to 03\_Workspace/algo/xxx\_sim!

# Note 2: All paths need to be relative to SConstruct, i.e. start with "#" (#=SConstruct-location)

# if there are multiple algos in one component (e.g. CB), create a dictionary as follows:

sim\_proj\_list={mono\_algo\_name:["#../"+component\_name+"\_sim/sim\_swc\_"+component\_name,

"#../"+component\_name+"\_sim/sim\_swc\_"+component\_name+"\_vis"

],

stereo\_algo\_name:["#../"+component\_name+"\_sim/sim\_swc\_"+component\_name ]

}

In the dictionary **sim\_proj\_list** for each SubAlgo the required PC simulation targets are defined by the corresponding project folder.

Note that it is not necessary to have the same set of PC simulation projects for all SubAlgos.

*Remark:* In case there is only one SubAlgo in a component, i.e. we have essentially **component\_name=algo\_name**, then variable **sim\_proj\_list** is not a dictionary, but a list and **sim\_proj\_list\_defines** does not exist at all. Please also refer to the template *02\_Development\_Tools\scons\_tools\scons\_templates\03\_Workspace\algo\xxx\sconstruct\_config.scfg.*

### Setting the define sets for SubAlgos

For all build targets which are **not** related to PC simulation, this has to be defined in the corresponding .scfg-file. For the C674x-Library this can be accomplished by:

# create release variant

release\_variant = "release"

release\_ccflags = "--abi=eabi -O3 --define=c6747 --display\_error\_number --diag\_warning=225 --optimizer\_interlist --opt\_for\_speed=5 -k"

if algo\_name == "cb" :

release\_cppdefines = ["CB\_MODULE”]

if algo\_name == "scb" :

release\_cppdefines = ["SCB\_MODULE]

and accordingly for other variants.

Of course, in principle, also additional defines or extra compiler flags can be set in addition at this stage. This distinction of cases shall be present for all relevant build targets **not** being related to PC simulation. (Otherwise the build targets for SubAlgos will be the same!)

Setting the define set for PC simulation is more comfortable, as this needs to be done in only one location and then applies to all PC simulation targets. Again this happens in *04\_Engineering\03\_Workspace\algo\xxx\sconstruct\_config.scfg,* where a dictionary of additional and possibly varying define sets are declared, see the example of GB:

sim\_proj\_list\_defines = {mono\_algo\_name : ["CB\_MODULE"],

stereo\_algo\_name : ["SCB\_MODULE"]

}

### Handling PC build targets not being shared to the project – “utilities”

In case there are also targets which should not be shared in the project, those shall be defined under “utils\_proj\_list” and/or “utils64\_proj\_list” in form of a corresponding python dictionary. Please refer to the template *04\_Engineering\03\_Workspace\algo\xxx\sconstruct\_config.scfg,* for an example.

# Build results

Results of SCons build process are stored in 04\_Build and some of them copied to 05\_Deliverables and to 05\_Testing directories.

The following figures indicate where to find all the individual build results.

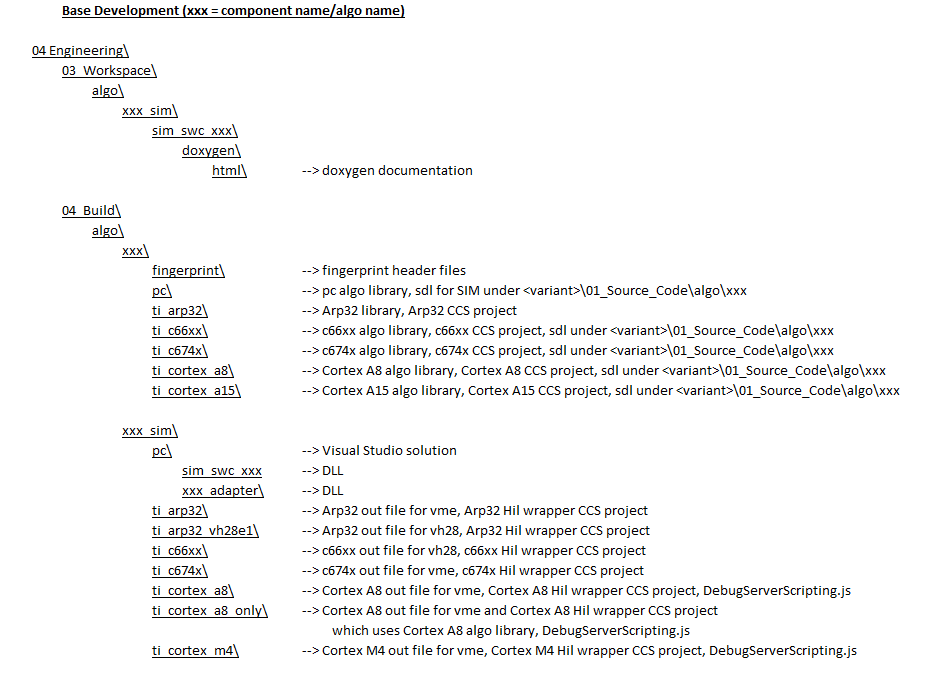


Figure 19: Build products of SCons in 04\_Build folder

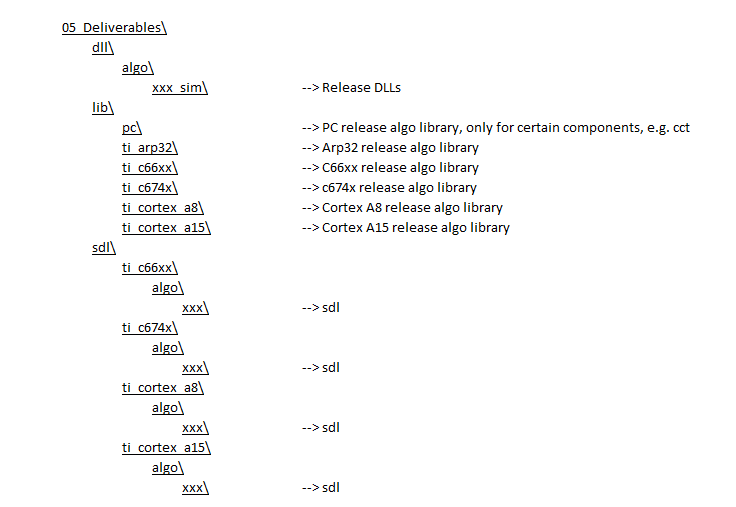


Figure 20: Build products of SCons in 05\_Deliverables

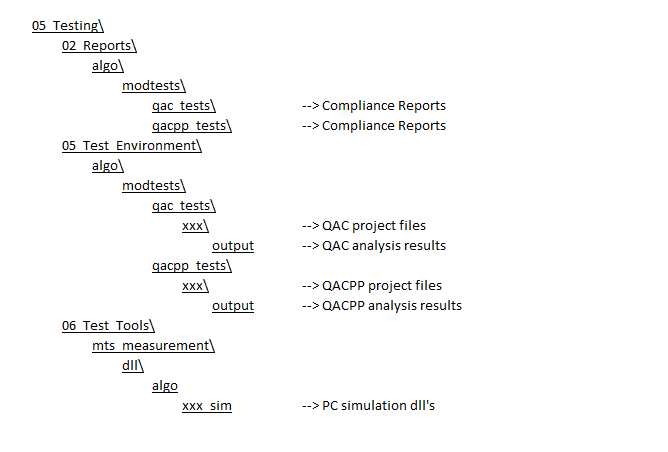


Figure 21: Build products of SCons under 05\_Testing

The build results under 05\_Testing\06\_Test\_Tools can vary depending on the GenericSconsflag being used, see Section 6.3.

# Errors

## Finding Error

Before SCons builds any targets, it reads all SConscript files. When there is an error in the SConscript file or any other files of GenericScons, SCons stops and reports the error with the information which script causes the error, including the scripts in the 02\_Development\_Tools\scons\_tools\. When tracing the error, the very last script in the error message indicates the one that contains the problem. This script has been called by the preceding script and so on. The actual error is displayed at the start of the error report.

When scons\_tools folder is copied or linked from another component, the path of the scons\_tools might be incorrect as shown below. This is because python uses directly the compiled python file (.pyc) in 02\_Development\_Tools\scons\_tools\. To fix this problem, remove all .pyc files.

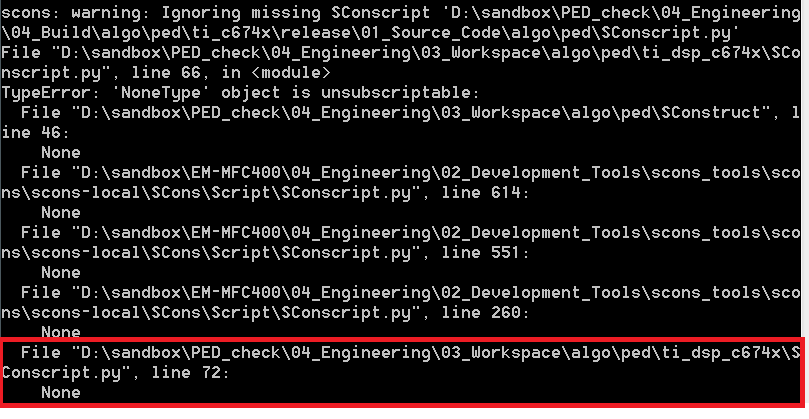


Figure : Finding Errors

## Typical Errors

This section is devoted to errors that typically occur. This list is updated continuously. Please also have a look at the subsequent section, where we provide a list of FAQ’s

### Build results are stored to the Source directory

Detailed Description: GenericScons is supposed to build a target whose source files are stored in 04\_Engineering\<some directory>. When SCons compiles those sources, the corresponding object files are stored in the same directory 04\_Engineering\<some directory>, but they should be stored somewhere under 04\_Engineering\04\_Build\.

Possible Resolution: Open the file(s), where the those sources are defined. It might be the case that they are set in the following way:

# algo source files

xxx\_algo\_sources = """

#../../../01\_Sorce\_Code/algo/xxx/frame/xxx\_main.cpp

"""

All source files should be defined relatively to the .scfg-file, where they are set, i.e. the above excerpt should substituted by something of the following form:

# algo source files

xxx\_algo\_sources = """

frame/xxx\_main.cpp

"""

This has to do with the behavior of **VariantDir.**

Having done those changes, the problem should disappear.

## Frequently asked questions

**Scons output:**

* *Question*: I would like to see the full command executed by SCons, how can I do that?

*Answer:* Set the output flag “verbose” when building the target by

Scons verbose=1 <target>

Then the full command line is displayed. Per default “verbose” is set to zero, so a more condensed command line is printed on the screen in order to facilitate looking for errors and warnings.

**Modifications in build settings**

* *Question*: How can I change the compiler options for my pc simulation target?

*Answer*: Open file **simenv\_config.scfg** in the corresponding folder and modify the variable <variant>\_ccflags.

* *Question*: How can I change the defines used for building my pc simulation target?

*Answer*: There are two possibilities:

1. Use the command line and build with

scons define=Define1,Define2 <target>

This would be a good move to try out a new define, when you want to compare the results with the case the define is not used.

1. Open file **simenv\_config.scfg** in the corresponding folder and modify the variable <variant>\_cppdefines. This would be the way to go if that define should be used by the whole team.

* *Question*: So far we haven’t built an ARM library for Cortex A15, but we need it now, what needs to be done?

*Answer*: There are three things to be done:

1. Component-specific settings for compilation

Create folder 03\_Workspace\algo\xxx\ti\_arm\_cortex\_a15 and add and modify file “ti\_arm\_cortex\_a15\_lib\_config.scfg” from scons\_tools\scons\_templates

1. Activation of Build of Cortex A15-library

In 03\_Workspace\algo\xxx\sconstruct\_config.scfg add the line

ca15\_used = True

1. Activate copy of corresponding SConscript

In 03\_Workspace\algo\xxx\sconscript\_setup\_config.scfg add the block

{

"name" : "cortex\_a15\_lib",

"copy" : True,

"dest\_folder" : "04\_Engineering/03\_Workspace/algo/" + component\_name + "/ti\_arm\_cortex\_a15",

"source" : ["algo/lib\_env\_ti\_cortex\_a15/SConscript.py"]

},

* *Question:* How can I integrate the build of a new build target?

*Answer:* For this purpose,GenericScons generally requires the following information:

* + 1. **A flag saying that this new target needs being built:**

This is done in *04\_Engineering\03\_Workspace\algo\xxx\sconstruct\_config.scfg.*

In case a new ECU library is integrated, add the corresponding flag, e.g. **ca8\_used** in case of a new Cortex A8-Library.

In case of a new PC simulation target, the corresponding path need to be appended to the variable **sim\_proj\_list.**

Please refer to the template *02\_Development\_Tools\scons\_tools\scons\_templates\03\_Workspace\algo\xxx\ sconstruct\_config.scfg.*

* + 1. **Copy command for corresponding shared SConscript.py-files to the project.**

This is done in *04\_Engineering\03\_Workspace\algo\xxx\sconscript\_setup\_config.*

Typically a block whose status is set to False needs being set to **True** or a new block needs being added.

Please refer to the existing *04\_Engineering\03\_Workspace\algo\xxx\sconscript\_setup\_config* and/or to the template *02\_Development\_Tools\scons\_tools\scons\_templates\03\_Workspace\algo\xxx\sconscript\_setup\_config.scfg*

* + 1. **Target-specific settings.**

Depending on the target certain .scfg-files shall be added to the project folder. For detailed information on which .scfg-files are to be added please refer to Section 3.3.2 or to the individual templates in *02\_Development\_Tools\scons\_tools\scons\_templates.* As a rule, target-specific settings are defined in .scfg-files which are located in the corresponding project folder.

**Other**:

* *Question*: Why are my source files built in the same directory as the source files are located in?

*Answer*: Probably the corresponding source file list needs some changes. Consider, e.g., 01\_Source\_Code\algo\xxx\algo\_lib\_file\_list.scfg:

All source files shall be defined with respect to the folder they are located in. If instead

was declared, then those files are built to the very same folder which they lie in. This behavior is caused by the use of “VariantDir” (see paragraph 3.2.2).

# Build Farm

Build farm scripts which are located in 04\_Engineering\02\_Development\_Tools\algo\_build\_cmd\ are extended to support SCons. To build using SCons, only the xml file of each algo component in 04\_Engineering\03\_Workspace\algo\xxx\_sim\ needs to be modified.

04\_Engineering\03\_Workspace\algo\cipp\_sim\ make\_cipp\_sim.xml (with scons target)

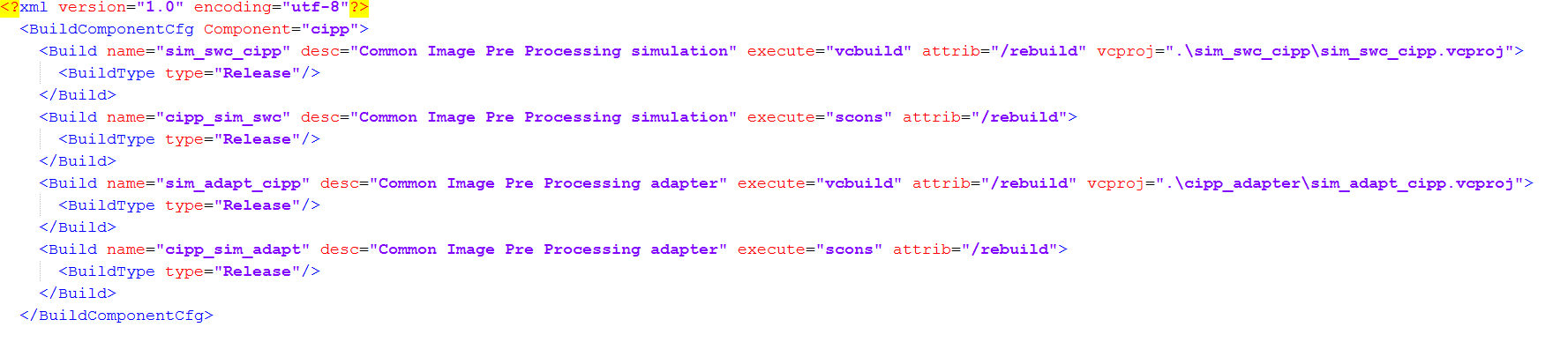


Figure 23: Example of xml file with SCons target

Limitation:

1. Only target with variant (“release”), for example xxx\_sim\_swc can be build using build farm. Target which does not have variant, for example xxx\_ide can not be build using build farm.
2. Only release variant is build using build farm.

To start building using build farm, go to 04\_Engineering\03\_Workspace\algo\ folder of MFC400 project and execute “**make\_all\_sim.cmd**”. This cmd file executes build farm script using the configuration in “**make\_all\_sim.xml”**, which contains xml file of each algo component.

Abbreviations

|  |  |
| --- | --- |
| Abbreviation | Description |
| DSP / BIOS | TI operating system for DSP |
| CCS | Code Composer Studio |
| SRS | Software Requirements Specification |
| API | Application Programming Interface |
| C / C++ | Programming language |
| CAN | Controller Area Network |
| LIN | Local Interconnect Network |
| APP | Application |
| uC | Microcontroller |
| DSP | Digital Signal Processor |
| MTS | Measurement, Testing and Simulation |

Table : Abbreviations

1. It is strongly advised that the absolute path of 04\_Engineering\03\_Workspace\algo\xxx does not exceed 100 characters, since – depending on the component – some build commands may exceed the maximum length prescribed by Windows. [↑](#footnote-ref-1)
2. Optimization flag can be overwritten by component-specific compiler flags in corresonding .scfg-files. [↑](#footnote-ref-2)
3. Optimization flag can be overwritten by component-specific compiler flags in corresonding .scfg-files. [↑](#footnote-ref-3)
4. Optimization flag can be overwritten by component-specific compiler flags in corresonding .scfg-files. [↑](#footnote-ref-4)
5. Optimization flag can be overwritten by component-specific compiler flags in corresonding .scfg-files. [↑](#footnote-ref-5)