FuseSoC Documentation

Release 2.4.6

Olof Kindgren

May 10, 2026

CHAPTERS

FuseSoC is a build system for digital hardware (e.g. Verilog or VHDL designs), and a package manager for reusable blocks in hardware designs.

This documentation contains material for different audiences.

The *User Guide* explains how to get started with FuseSoC, starting from the installation.

The *Reference Guide* provides a detailed description of all file formats and APIs.

The *Developer's Guide* is aimed at developers of FuseSoC itself. It explains how to set up a development environment, how the source code is structured, and how patches and bug reports can be submitted to the project.

CHAPTER

ONE

FUSESOC USER GUIDE

The FuseSoC User Guide is aimed at hardware developers utilizing FuseSoC to build and integrate their hardware designs.

Learn how to use FuseSoC in an existing project.

Have you checked out a hardware design project that uses FuseSoC and are trying to understand how to build the design? Get started by *installing FuseSoC*, and then have a look at the *usage documentation*.

Add FuseSoC support to your hardware project.

If you are starting a new hardware design project, or already have source files and are looking for a better way to build your project and integrate third-party components? Get started by *installing FuseSoC*, read a bit about the *concepts and terminology of FuseSoC*, and then move on to *add FuseSoC core description files* to your project.

1.1 Why FuseSoC?

FuseSoC is an award-winning package manager and a set of build tools for HDL (Hardware Description Language) code.

Its main purpose is to increase reuse of IP (Intellectual Property) cores and be an aid for creating, building and simulating SoC solutions.

FuseSoC makes it easier to

- reuse existing cores
- create compile-time or run-time configurations
- run regression tests against multiple simulators
- Port designs to new targets
- let other projects use your code
- set up continuous integration

FuseSoC is non-intrusive Most existing designs don't need any changes to work with FuseSoC. Any FuseSoC-specific patches can be applied on the fly during implementation or simulation

FuseSoC is modular It can be used as an end-to-end flow, to create initial project files for an EDA tool or integrate with your custom workflow

FuseSoC is extendable Latest release support simulating with GHDL, Icarus Verilog, Isim, ModelSim, Verilator and Xsim. It also supports building FPGA images with Altera Quartus, project IceStorm, Xilinx ISE and Xilinx Vivado. Support for a new EDA tool requires ~100 new lines of code and new tools are added continuously

FuseSoC is standard-compliant Much effort has gone into leveraging existing standards such as IP-XACT and vendor-specific core formats where applicable.

FuseSoC is resourceful The standard core library currently consisting of over 100 cores including CPUs, peripheral controllers, interconnects, complete SoCs and utility libraries. Other core libraries exist as well and can be added to complement the standard library

FuseSoC is free software It puts however no restrictions on the cores and can be used to manage your company's internal proprietary core collections as well as public open source projects

FuseSoC is battle-proven It has been used to successfully build or simulate projects such as Nyuzi, Pulpino, VScale, various OpenRISC SoCs, picorv32, osvvm and more.

1.2 Installing FuseSoC

FuseSoC is written in Python and runs on all major operating systems.

1.2.1 System Requirements

Before installing FuseSoC check your system requirements.

- Operating System: Linux, Windows, macOS
- Python 3.6 or newer. (The last version supporting Python 2.7 is FuseSoC 1.10.)
- The Python packages `setuptools` and `pip` need to be installed for Python 3.

1.2.2 Installation under Linux

> **Note**
>
> Do not type the $ symbol shown in the instructions below. The symbol indicates that the command is to be typed into a terminal window. Lines not prefixed with $ show the output of the command. Depending on your system, the output might be different.

FuseSoC is provided as `fusesoc` Python package and installed through pip, the Python package manager. The steps below cover the most common installation cases. Refer to the pip documentation for more advanced installation scenarios.

FuseSoC, like all Python packages, can be installed for the current user, or system-wide for all users. The system-wide installation typically requires root permissions.

Installation for the current user

To install the current stable version of FuseSoC for the current user, open a terminal window and run the following command. If an older version of FuseSoC is found, this version is upgraded to the latest stable release.

```
$ pip3 install --upgrade --user fusesoc
```

Check that the installation worked by running

```
$ fusesoc --version
2.4.6
```

If this command works FuseSoC is installed properly and ready to be used.

If the terminal reports an error about the command not being found check that the directory `~/.local/bin` is in your command search path (`PATH`), or perform a system-wide installation instead (see below).

System-wide installation

FuseSoC can be installed for all users on a system. This operation typically requires root permissions.

```
$ sudo pip3 install --upgrade fusesoc
```

Uninstalling FuseSoC

Use `pip` to remove FuseSoC from your system.

```
$ pip3 uninstall fusesoc
```

1.2.3 Installation under Windows

FuseSoC is provided as `fusesoc` Python package and installed through pip, the Python package manager. Python is not installed by default on Windows, so doing that is the first step. Install the latest version of Python, either from python.org or from the Windows store.

Open up a shell (cmd, powershell, gitbash etc.) and run:

```
$ python --version
```

If the output is something along the lines of `Python 3.10.7`, Python has been successfully installed.

Installation for the current user

To install the current stable version of FuseSoC for the current user, open a shell and run the following command. If an older version of FuseSoC is found, this version is upgraded to the latest stable release.

```
$ pip3 install --upgrade fusesoc
```

Check that the installation worked by running

```
$ fusesoc --version
2.4.6
```

If this command works FuseSoC is installed properly and ready to be used.

The `fusesoc.exe` file should be installed in the Python `Scripts` directory (example: `C:/Users/youruser/AppData/Local/Python/Python310/Scripts`) folder, which should have been automatically added to the search path (`PATH`) when Python was installed.

Uninstalling FuseSoC

Use `pip` to remove FuseSoC from your system.

```
$ pip3 uninstall fusesoc
```

1.2.4 Installation under macOS

You can use `pip` to install as same as Linux.

```
$ pip3 install --upgrade --user fusesoc
```

1.3 Understanding FuseSoC

1.3.1 The components of FuseSoC

FuseSoC is a package manager and a build system for HDL code. The following sections explain the main concepts of FuseSoC, and how they work together to obtain, configure, and build hardware designs.

A fundamental entity in FuseSoC is a core. Cores can be discovered by the FuseSoC package manager in local or remote locations, and combined into a full hardware design by the build system.

FuseSoC's basic building block: cores

A **FuseSoC core** is a reasonably self-contained, reusable piece of IP, such as a FIFO implementation. In other contexts, the same concept is called package (as in deb package, npm package, etc.), or programming library (a term which is used with a different meaning in FuseSoC).

A core in FuseSoC has a name, such as `example:ip:fifo`. The name and other information about the core, such as the list of (e.g. Verilog or VHDL) source files, is contained in a description file called the **core file**. Core files have filenames ending in `.core`.

A core can also have dependencies on other cores. For example, a "FIFO core" can have a dependency on an "SRAM core."

Discover cores: the package manager

FuseSoC cores can be stored in many places, both locally and remote. Finding cores, and producing a list of cores available to the user, is the job of the FuseSoC package manager.

To find cores, the FuseSoC package manager searches through a configurable set of **core libraries**. A core library can be local or remote. In the simplest case, a local core library is just a directory with cores, as it is common in many hardware projects. To support more advanced cases of code re-use and discovery, FuseSoC can also use remote core libraries. Remote core libraries can be a key enabler to IP re-use especially in larger corporate settings, or in free and open source silicon projects.

From cores to a whole system: the build system

The FuseSoC build system resolves all dependencies between cores, starting from a top-level core. A top-level core is technically just another FuseSoC core, but with a special meaning: it is the entry point into the design. The output of the dependency resolution step is a list of source files and other metadata that an EDA tool needs to build (i.e. synthesize, simulate, lint, etc.) the top-level design. The dependency resolution process can be influenced by constraints. Constraints are effectively input variables to the build process, and capture things like the target of the build (e.g. simulation, FPGA synthesis, or ASIC synthesis), the EDA tool to be used (e.g. Verilator or Xilinx Vivado), and much more.

From a file list to a synthesized design: EDAlize it!

After the build system has collected all source files and parameters of the design, it is time to hand off to the build tool, such as Xilinx Vivado for FPGA synthesis, Verilator for Verilog simulation, and so on.

How exactly the hand-off is performed is highly tool-dependent, but FuseSoC abstracts these differences away and users typically don't need to worry about them. For example, when using Vivado FuseSoC creates a Vivado project file and then executes Vivado to run through the synthesis and place and route steps until a final bitstream is produced. For Verilator, it creates a Makefile and then calls Verilator to do its job.

FuseSoC supports many of the proprietary and open source EDA tools commonly used, and can be easily extended to support further ones.

1.3.2 Concepts of the FuseSoC build system

To understand how FuseSoC builds a design it is necessary to understand three basic concepts: tool flows, targets, and build stages.

Tool flows

A tool flow (often abbreviated to just "tool" or "EDA tool") is a piece of software operating on the design to analyze it, simulate it, transform it, etc. Common categories of tools are simulators, synthesis tools, or static analysis tools. For example, Verilator is a tool (a simulation and static analysis tool), as is Xilinx Vivado (an FPGA tool flow), or Synopsys Design Compiler (an ASIC synthesis tool).

FuseSoC tries its best to hide differences between tools to make switching between them as easy as possible. For example, it often only requires a change in the command line invocation of FuseSoC to simulate a design with Synopsys VCS instead of Icarus Verilog. Of course, customization options for individual tools are still available for when they are needed.

Targets

Many things can be done with a hardware design: it can be synthesized for an FPGA, it can be simulated, it can be analyzed by lint tools, and much more. Even though all of these things operate on the same hardware design, there are differences: design parameters (e.g. Verilog defines and parameters, or VHDL generics) are set differently, source files are added or substituted (e.g. a top-level testbench wrapper is added, or a behavioral model of an IP block is exchanged against a hard macro), etc. In FuseSoC, a target is a group of such settings. Users of FuseSoC can freely name targets. Commonly used targets are one for simulation (typically called `sim`), one for FPGA or ASIC synthesis (`synth`), and one for static analysis (`lint`).

Build stages

FuseSoC builds a design in three stages: setup, build, and run.

1. The **setup** stage. In this first step, the design is stitched together and prepared to be handed over to the tool flow.
    1. A dependency tree is produced, starting from the top-level core.
    2. Generators (special cores with dynamic behavior) are called. Cores produced by generators are inserted into the dependency tree as well.
    3. The dependency tree is resolved to produce a flattened view of the design. All design information is written into an EDAM file.
    4. Tool-flow specific setup routines are called.
2. The **build** stage runs the tool flow until some form of output file has been produced.
3. The **run** stage somehow "executes" the build output. What this exactly means is highly tool flow dependent: for simulation flows, the simulation is executed. For static analysis (lint) flows, the lint tool is called and its output is displayed. For FPGA flows, the FPGA is programmed with the generated bitstream.

1.4 Running FuseSoC

FuseSoC is a command-line tool; this section explains how to use it. The following content is aimed at users who already have a hardware design which uses FuseSoC.

1.4.1 Build a design

The `fusesoc run` group of commands is used to setup, build, and (if possible) run a design. The exact actions taken by the individual steps depend on the toolflow.

```
usage: fusesoc run [-h] [--no-export] [--build-root BUILD_ROOT] [--setup] [--build] [--
↪run] [--target TARGET] [--tool TOOL] [--flag FLAG] [--system-name SYSTEM_NAME] system .
↪..

positional arguments:
  system                Select a system to operate on
  backendargs           arguments to be sent to backend

optional arguments:
  -h, --help            show this help message and exit
  --no-export           Reference source files from their current location instead of␣
↪exporting to a build tree
  --build-root BUILD_ROOT
                        Output directory for build. Defaults to build/$VLNV
  --setup               Execute setup stage
  --build               Execute build stage
  --run                 Execute run stage
  --target TARGET       Override default target
  --tool TOOL           Override default tool for target
  --flag FLAG           Set custom use flags. Can be specified multiple times
  --system-name SYSTEM_NAME
                        Override default VLNV name for system
```

When FuseSoC is invoked with the run command, it will create an empty working directory called *work_root* internally, where it by default will create all project files, copy all used source files, build and optionally run the project.

Setup, build and run

The process of running EDA tools is divided into three steps called *setup*, *build* and *run*. The *setup* stage creates the working directory and all project files. The *build* stage runs one or more EDA tools to build an artifact, e.g. a GDS, simulation model or FPGA image. The *run* stage is only implemented for some tool flows, such as simulation flows where it runs the simulation. Some FPGA flows uses the *run* stage to program an FPGA device.

Normally FuseSoC runs all three stages, but if the *–setup* flag is added, it will stop after the setup stage and if the *–build* flag is set it will stop after the build stage. Many of the newer backends don't need these flags and will instead only run setup or build when input files or options have changed.

Work root

work_root is a private working directory and should not be shared between different builds. It is however perfectly fine to reuse the working directory for e.g. running several simulations using different runtime options as long as the build-time options and source files are not modified.

By default, FuseSoC will use *build/<sanitized VLNV>/<target>*, where *<sanitized VLNV>* is the top-level VLNV with underscore instead of colon as the separator. For the Flow API, *<target>* is just the name of the target in the core description file, e.g. *sim*. For the old Tool API, it is a combination of target name and the tool backend, e.g. *sim-verilator*. The work root directory can be changed with the *–work-root* option.

Exporting source files

The standard behavior for FuseSoC is to copy all used source files into a subdirectory of the work root. This has three advantages. The work root is self-contained with all the source files and can be copied elsewhere for archival purposes or to build on another machine. No stray files are picked up by mistake from the original source directories. It is always possible to know from exactly which files a build was created. Despite this, there are situations where it is preferable to reference the source files from their original location. This can be done by adding the *–no-export* flag.

1.5 Building a design with FuseSoC

The FuseSoC build system pieces together a hardware design from individual cores.

Building a design in FuseSoC means *calling a tool flow to produce some output, and execute it.* Depending on the *target* and the *tool flow* chosen, the build process can do and produce very different things: it could produce a runnable simulation, generate an FPGA bitstream, or run a static analysis tool to check for common programming errors.

Two steps are required to build a hardware design with FuseSoC:

1. Write one or more FuseSoC core description files. See *Writing core files* for information on how to write core description files.
2. Call `fusesoc run`. FuseSoC is a command-line tool and accessible through the `fusesoc` command. See *Running FuseSoC* for information on how to use the `fusesoc` command.

Typically, FuseSoC support can be added to an existing design without changes to the directory structure or the source files.

The first three sections are recommended reading for all users of FuseSoC. The first section *Writing core files* is an introduction into *core description files* and how to write them. The second and third section, *Interfacing EDA tool flows* and *Dependencies: link cores together for re-use* look at how to customize what the (EDA) *tools* are doing, and how cores can be combined to form a larger system.

The subsequent sections are advanced topics, which are only relevant in some projects.

A full reference documentation on the CAPI2 core file format can be found in the section ref_capi2.

1.5.1 Writing core files

A *core* is described in a core description file, or core file.

Core files are written in *YAML* syntax and follow the FuseSoC's own CAPI (version 2) schema, which describes the structure of core files (e.g. which keys and values are allowed where). Don't worry: using FuseSoC neither requires a full understanding of YAML, nor an up-front knowledge of CAPI. However, some key facts about YAML are important.

Things one should know about YAML

- **Whitespace matters** (as in Python): indentation is used to group settings together to form a hierarchy. The exact amount of whitespace used for indentation does not matter; typically two or four spaces are used.
- Think of a YAML file as a **hierarchical, typed data structure**. There are lists, dictionaries (key/value pairs), integers, strings, etc.
- YAML syntax provides **multiple ways to describe the same structure**. It does not matter to FuseSoC which syntax variant is used. For example, a list of items can be written in the following two, semantically identical ways.

```
[ "item1", "item2" ]
```

  is semantically identical to

```
- "item1"
- "item2"
```

The same is true for dictionaries (key/value pairs).

```
{ key1: "value1", key2: "value2" }
```

is semantically identical to

```
key1: "value1"
key2: "value2"
```

In most cases, the longer (second) form is preferred, as it is easier to make changes while keeping the diff easy to read.

- **We recommend always adding double quotes around strings used as value.** In most cases, strings in YAML do not need to be surrounded by (single or double) quotation marks. However, the exact rules when quoting is needed are not easily summarized without in-depth understanding of the YAML language syntax. At minimum quotation marks are *required* when strings start with an exclamation mark. Always using double quotes avoids having to even think about the exact rules.

  Examples:

```
recommended: "quotedvalue"
required: "!tool_verilator (something)"
possible: unquotedvalue
```

For a quick introduction into most of YAML's features have a look at Learn YAML in Y minutes. The full YAML 1.2 specification is available at yaml.org (it's not an easy read, though).

An example: the blinky core

The following sections explain how to add FuseSoC support to a hardware project. The code is taken from an example design in the FuseSoC source tree in the `tests/userguide/blinky` directory.

The design consists of two SystemVerilog files, a testbench, a Xilinx constraint file (with pin mappings for a Nexys Video FPGA board), and finally, the FuseSoC core file.

```
$ tree tests/userguide/blinky/
tests/userguide/blinky/
├── blinky.core
├── data
│   └── nexys_video.xdc
├── rtl
│   ├── blinky.sv
│   └── macros.svh
└── tb
    └── blinky_tb.sv

3 directories, 5 files
```

To get started, here's the full `blinky.core` file. The following sections will refer back to this example to discuss it in detail.

Listing 1: `blinky.core`, an exemplary core file

```
CAPI=2:
name: fusesoc:examples:blinky:1.0.0
description: Blinky, a FuseSoC example core

filesets:
  rtl:
    files:
      - rtl/blinky.sv
      - rtl/macros.svh:
          is_include_file: true
    file_type: systemVerilogSource

  tb:
    files:
      - tb/blinky_tb.sv
    file_type: systemVerilogSource

  nexys_video:
    files:
      # YAML short form, see rtl/macros.svh above for the longer form.
      - data/nexys_video.xdc: {file_type: xdc}

targets:
  # The "default" target is special in FuseSoC and used in dependencies.
  # The "&default" is a YAML anchor referenced later.
  default: &default
    filesets:
      - rtl
    toplevel: blinky
    parameters:
      - clk_freq_hz

  # The "sim" target simulates the design. (It could have any name.)
  sim:
    # Copy all key/value pairs from the "default" target.
    <<: *default
    description: Simulate the design
    default_tool: icarus
    filesets_append:
      - tb
    toplevel: blinky_tb
    tools:
      icarus:
        iverilog_options:
          - -g2012 # Use SystemVerilog-2012
      modelsim:
        vlog_options:
          - -timescale=1ns/1ns
    parameters:
      - pulses=10
```

(continues on next page)

(continued from previous page)

```
  # The "synth" target synthesizes the design. (It could have any name.)
  synth:
    <<: *default
    description: Synthesize the design for a Nexys Video FPGA board
    default_tool: vivado
    filesets_append:
      - nexys_video
    tools:
      vivado:
        part: xc7a200tsbg484-1
    parameters:
      - clk_freq_hz=100000000

parameters:
  clk_freq_hz:
    datatype    : int
    description : Frequency of the board clock, in Hz
    paramtype   : vlogparam
  pulses:
    datatype    : int
    description : Number of pulses to run in testbench
    paramtype   : vlogparam
```

Naming the core file

The core file can have any name, but it must end in `.core`. It is recommended to choose a file name matching the core name, as discussed below.

The first line: `CAPI=2`

A core file always starts with the line `CAPI=2`. No other content (including comments) is allowed before this line, as FuseSoC uses this line to differentiate between different versions of the CAPI schema. Only CAPI version 2 is specified at the moment.

The core name, version, and description

Each core has a name, given in the `name` key. Core names can be freely chosen, but need to follow a common structure called *VLNV*. *VLNV* stands the four parts of a core name, which are separated by colon (`:`): Vendor, Library, Name, and Version.

Version numbers should be three numbers in the form major.minor.patch and follow *semantic versioning* (SemVer).

Cores can also have a description, given in the `description` key. A description is optional, but recommended.

```
name: fusesoc:examples:blinky:1.0.0
description: Blinky, a FuseSoC example core
```

In this example, the vendor is `fusesoc`, the library is `examples`, and the name of the core is `blinky`. The version is set to `1.0.0`.

Specifying source files

A core typically consists of one or multiple source files. Source files are grouped into file sets under the `filesets` key.

FuseSoC does neither mandate a specific grouping, nor naming of file sets. It is common to use one file set for RTL (design) files, and one for testbench files.

The following example shows a single file set, `rtl`, with a set of common keys.

```
filesets:
  rtl:
    files:
      - rtl/blinky.sv
      - rtl/macros.svh:
          is_include_file: true
    file_type: systemVerilogSource
```

For each named file set, several keys are supported:

- `files`: An ordered list of source files. The list of source files is ordered: the files will be passed to the tool in exactly the given order. This is important, for example, in SystemVerilog, where packages need to be compiled before they can be used by subsequent source files.
- `file_type`: The default file type for all files in the `files` list.
- `depend`: Dependencies on other cores. Dependencies are explained in depth at *Dependencies: link cores together for re-use*.

Source files

For local cores, source files are resolved relative to the location of the core file and must be stored in the same directory as the core file, or in a subdirectory of it. For remote cores, file names are typically relative to the repository or archive root. Source file names cannot be absolute paths, or start with `../`.

Optionally, source files can have attributes; the file `macros.svh` is an example of that. When specifying attributes, end the file name with a colon (`:`), and specify attributes as key-value pairs below it. (Alternatively, the equivalent short form syntax can be used, e.g. `macros.svh: {is_include_file: true}`.)

The most common attributes are:

- `is_include_file`: The file is an include file. In Verilog and C/C++, this means the file is not passed to the tool directly, but instead the file is included by another source file. FuseSoC ensures that the tool finds the include file, e.g. by passing an appropriate include path to the tool.
- `file_type`: Override the default file type of the fileset for this particular file.

Refer to the CAPI2 reference documentation for more details.

File types

A file type describes the type of source file. FuseSoC does not use this information itself, but passes it on to tool backends which then configure the tool appropriately depending on the file type encountered.

Commonly used file types are:

- `verilogSource`: Verilog source code, up to Verilog-2001. Files ending in `.v` or `.vh` should use this type.
- `systemVerilogSource`: SystemVerilog source code (design and test code). Files ending in `.sv` or `.svh` should use this type.
- `vhdlSource`: VHDL source code. Files ending in `.vhd` or `.vhdl` should use this file type.

Refer to the CAPI2 reference documentation for more details.

Targets

A *target* can be seen as something you would like to do with the source code in the core: synthesize it, simulate it, lint it. Targets are specified as dictionaries under the `targets` top-level key.

```
targets:
  # The "default" target is special in FuseSoC and used in dependencies.
  # The "&default" is a YAML anchor referenced later.
  default: &default
    filesets:
      - rtl
    toplevel: blinky
    parameters:
      - clk_freq_hz

  # The "sim" target simulates the design. (It could have any name.)
  sim:
    # Copy all key/value pairs from the "default" target.
    <<: *default
    description: Simulate the design
    default_tool: icarus
    filesets_append:
      - tb
    toplevel: blinky_tb
    tools:
      icarus:
        iverilog_options:
          - -g2012 # Use SystemVerilog-2012
      modelsim:
        vlog_options:
          - -timescale=1ns/1ns
    parameters:
      - pulses=10

  # The "synth" target synthesizes the design. (It could have any name.)
  synth:
    <<: *default
    description: Synthesize the design for a Nexys Video FPGA board
    default_tool: vivado
    filesets_append:
      - nexys_video
    tools:
      vivado:
        part: xc7a200tsbg484-1
    parameters:
      - clk_freq_hz=100000000
```

The blinky example shown above defines three targets: the `default` target, a `sim` target to simulate the design, and a `synth` target to synthesize it. Many designs also define a `lint` target to run static analysis jobs. The `sim` and `synth` targets are optional and could have had any name. The `default` target is special and required.

Within a target

Within each target block multiple keys determine what the target does. The most common keys are:

- `filesets`: An ordered list of file sets (source files) included in the target.
- `description` (optional): A description of the target.
- `toplevel` (optional): The name of the design toplevel. (For advanced scenarios it is possible to specify a list of multiple toplevels instead of just a single one.)
- `default_tool` (optional): The default tool to be used to build the target. The tool can also be set or overridden through a FuseSoC command-line argument.
- `tools` (optional): Tool-specific settings, grouped by tool name.
- `parameters` (optional): Parameters (Verilog parameters and defines, VHDL generics, etc.) to be passed to the design, or forced to a certain value.

The `filesets_append` key is part of an inheritance schema and explained further in section *Inheritance and the default target*.

The `default` target

The `default` target is the only required target. It serves two purposes:

- The `default` target is used if no other target is explicitly selected when running FuseSoC.
- The contents of the `default` target are used if the core is used as dependency (described in detail in *Dependencies: link cores together for re-use*).

All reusable code in the core should go into the `default` target: RTL files, lint waivers, reusable constraints, etc.

Inheritance and the `default` target

Importantly, other targets in the same core do *not* inherit the contents of the `default` target automatically. To achieve such inheritance behavior, FuseSoC provides a flexible inheritance mechanism, based on YAML anchors/references, YAML << merge operator, and a FuseSoC-specific list append feature.

The *blinky.core* shows the recommended template to inherit configuration between targets.

1. Add `&default` after the `default:` text. This defines a YAML anchor named `default`, which can be referenced later in the file.
2. Add a line `<<: *default` to the target where you want to inherit from `default` (the `sim` and `synth` targets in the example code). This line will effectively "copy over" all configuration under the `default` target.

As always with inheritance the interesting questions are around overriding behavior.

- Settings (keys) given in the target which inherits from `default` override the keys in `default`. For example, the `toplevel` key in the `sim` target is overridden to be `tb`. Note that no merging of setting data structure is performed.
- For settings which are lists, for example the `filesets` key, FuseSoC provides a way to combine lists by adding `_append` to the name of the key.

  This behavior is best explained by example. The `filesets` list in the `default` target consists of a single item, `rtl`. The `sim` target wants to append the item `tb` (a file set with testbench files) to the list. To do so, it specifies the special `filesets_append` key with a partial list. When evaluating the core file, FuseSoC appends the contents of `sim.fileset_append` at the end of `default.fileset` to form a list with two items: `rtl`, and `tb`. The same behavior works for all lists in core files.

Signed core files

The fusesoc cli tool can produce a cryptographic signature for a core (currenly using ssh keys). The signature is stored in a separate file named as the signed core but with `.sig` appended.

For verifying signatures a file called a *trustfile* is used. The trustfile contains a list of trusted users and their public ssh keys, one per line:

```
$ cat tests/signature_files/trustfiles/trustfile_1ed_and_2
user1@example.com ssh-ed25519␣
↪AAAAC3NzaC1lZDI1NTE5AAAAIFL1l8BB+EnUUkwMtMMqv1HFw8q2SZ7ERuGcFYt8VtqL
user2@example.com ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIP6w+b8Xue0Jo/
↪Hskd9B+Ttr6g9rq99qbZV/01q+FgED
```

The location of this file can be set in the `[main]` section of the `fusesoc.conf` file:

```
[main]
ssh-trustfile = /home/anders/FuseSoCdb/git/fusesoc/tests/signature_files/trustfiles/
↪trustfile
```

This setting can also be overridden using the command line option `--ssh-trustfile`.

The signing process is accomplished with the `core sign` command, which takes a core name and a path to a private ssh key as arguments.

The signing status of a core is shown in the `core list` and `core show` commands in the fusesoc cli tool.

1.5.2 Interfacing EDA tool flows

A design described by FuseSoC core description files is intended to be used by one or more EDA tools such as simulators, synthesis tools, linters etc. FuseSoC uses Edalize to configure and run EDA tools. FuseSoC users still need to select which EDA tool or flow of tools to be run and in many cases also provide configuration to Edalize. This is done from the target section of the core description files.

> **Note**
>
> Edalize currently exposes two different APIs called the *tool API* and *flow API* respectively. These have different configuration keys in the core description files. The new flow API is intended to become the default API. However, not all Edalize backends have been converted to the new API, so the old tool API remains in use. If both the *default_tool* key (from the tool API) and the *flow* key (from the flow API) is defined in a target, the flow API will take precedence.

Edalize flow API

Edalize supports many different EDA tools and combinations of tools working together, called flows. A flow could e.g. be an FPGA bitstream flow where one EDA tool is used for synthesis, another one for place & route and a third one to convert the P&R database into an image that can be loaded into the FPGA. Another example could be a simulation flow, where the simulator itself is just one tool, but where a code conversion tool is used to preprocess the input to the simulator, e.g. ghdl to convert VHDL to Verilog for tools that don't handle the former well enough.

FuseSoC abstracts away many differences between *tools* and tries to provide sane defaults to build many designs out of the box with no further configuration required. However, not all tool-specific details can be hidden.

At the same time, a certain level of tool-specific configurability is required to make full use of the features available in different tools.

There are two categories of options available for the Edalize flows. *Flow options* that affect how the tools are chained together (the flow graph) and *tool options* for the individual tools to be run as part of the flow. This means that since the flow options influence which tools that will be run, some tool options only become available for certain combinations of flow options.

Only one flow can be defined for a target, but the flow itself can be configured in different ways. The example below shows how the *test* target selects and configures a flow. The *flow* key selects the flow itself. The selected *sim* flow has a *flow option* called *tool* that decides which simulator to use. *iverilog_options* and *vlog_options* are tool options for Icarus Verilog and Siemens QuestaSim/ModelSim and will be passed to the appropriate tool if it is present in the flow graph.

This setup selects icarus as the tool, which means the *vlog_options* will not be used. However, all *flow options* and *tool options* are also automatically available on the command-line, which means that passing *–tool=modelsim* as a backend parameter will override the tool setting from the target. The same can be done for the two tool options, with the difference that for flow or tool option that are lists, any additional values passed on the command-line will append rather than replace the values in the core description file.

```
# An excerpt from a core file.
# ...
targets:
  test:
    # ...
    flow: sim
    flow_options:
        tool: icarus
        iverilog_options:
          - -g2012 # Use SystemVerilog-2012
        vlog_options:
          - -timescale=1ns/1ns
```

The available options for any flow can be found in the Edalize documentation. They can also be found by running a target with the *–help* flag. *fusesoc run –target=<some target> <some core> –help*

Edalize tool API

> **Note**
>
> Refer to the CAPI2 reference documentation or run `fusesoc tool list` for a list of all available tools and their options.

FuseSoC abstracts away many differences between *tools* and tries to provide sane defaults to build many designs out of the box with no further configuration required. However, not all tool-specific details can be hidden. At the same time, a certain level of tool-specific configurability is required to make full use of the features available in different tools. Tool options are FuseSoC's way of customizing the way tools are used to build the design.

When calling `fusesoc run` on the command line any tool can be chosen to build a design with the `--tool` argument. If no tool-specific configuration is given in the core file, the default tool configuration is used, which might or might not work for a given design.

To customize tool behavior a tool-specific section can be added to a core file at `targets.TARGETNAME.tools.TOOLNAME`. The name of the tool (`TOOLNAME`) must match FuseSoC's internal tool name (as passed to `fusesoc run --tool=TOOLNAME`). Depending on the tool different options are available. Refer to the CAPI2 reference documentation for a list of all available tools and their options.

Most tool backends provide a way to set command line options to influence how the tools are called. Typically, these keys are called `BINARYNAME_options`, and they take a list of arguments as value.

The example below shows how tool options for Icarus Verilog (`icarus`) and Mentor ModelSim (`modelsim`) are set.

- The `iverilog` binary will be called with the `-g2012` command-line argument, indicating that SystemVerilog 2012 support should be enabled.
- Similarly, for ModelSim the argument `-timescale=1ns/1ns` will be passed to the `vlog` binary, which elaborates the design.

```
# A fragment from blinky.core
# ...
targets:
  sim:
    # ...
    tools:
      icarus:
        iverilog_options:
          - -g2012 # Use SystemVerilog-2012
      modelsim:
        vlog_options:
          - -timescale=1ns/1ns
```

Note

Where to find tool- or flow-specific code in FuseSoC

The tool- and flow-specific code is provided by the Edalize library. Most files, such as project files and Makefiles, are templates within edalize and can be improved easily if necessary. Please open an issue at the edalize issue tracker on GitHub to suggest improvements to tool-specific code.

1.5.3 Dependencies: link cores together for re-use

For a long time, productivity gains in hardware designs have been achieved primarily by re-using existing code, a.k.a. IP blocks. Re-use is also engrained into FuseSoC: re-usable hardware design components are packaged into cores and then used in other designs. In FuseSoC (and many other package managers), re-use is achieved by expressing dependencies between FuseSoC cores.

This section explains how dependencies are specified, how they are resolved by FuseSoC, and how they can be constrained.

A dependency example: DualBlinky

We introduced the basic FuseSoC features by creating a reusable core called Blinky. To illustrate the concept of dependencies in FuseSoC we employ another example: DualBlinky, the "dual-core" version of Blinky. Again, all source code is available in the FuseSoC source tree in the `tests/userguide/dualblinky` directory.

```
$ tree tests/userguide/dualblinky
tests/userguide/dualblinky
├── data
│   └── nexys_video.xdc
├── dualblinky.core
└── rtl
    └── dualblinky.sv

2 directories, 3 files
```

The core file is shown in full below.

Listing 2: `dualblinky.core`, 2x Blinky

```
CAPI=2:
name: fusesoc:examples:dualblinky:1.0.0
description: DualBlinky, a FuseSoC example with dependencies

filesets:
  rtl:
    files:
      - rtl/dualblinky.sv
    file_type: systemVerilogSource
    depend:
      - fusesoc:examples:blinky

  nexys_video:
    files:
      - data/nexys_video.xdc:
          file_type: xdc

targets:
  default: &default
    filesets:
      - rtl
    toplevel: dualblinky

  synth:
    <<: *default
    description: Synthesize the design for a Nexys Video FPGA board
    default_tool: vivado
    filesets_append:
      - nexys_video
    tools:
      vivado:
        part: xc7a200tsbg484-1
    parameters:
      clk_freq_hz: 100000000

parameters:
  clk_freq_hz:
    datatype: int
    description: Frequency of the board clock, in Hz
    paramtype: vlogparam
```

Specifying a dependency

Dependencies in FuseSoC are expressed between a file set and a core. They are listed in a *core file* as in the `filesets.FILESET_NAME.depend` section.

The example below shows how to create a dependency between the `fusesoc:examples:blinky` core and the `rtl` file set of the `fusesoc:examples:dualblinky:1.0` core.

```
# Excerpt of dualblinky.core
```

(continues on next page)

(continued from previous page)

```
# ...

filesets:
  # ...
  rtl:
    # ...
    depend:
      - ">=fusesoc:examples:blinky:1.0"
```

Note

YAML requires quotation marks for strings with special characters, as they are used in version constraints. Both single (') and double quotes (") can be used.

File ordering

File ordering (compilation order) is important in many hardware design projects. The following rules apply.

- Files from dependencies are inserted into the file list before the files in the file set where the dependency is declared.
- The order in which dependencies are listed in the `depend` section does not imply any ordering. That is, specifying `depend: [A, B]` does not guarantee that files from core `A` are included before the ones from core `B`. (If such an order is desired, make core `B` depend on `A`.)

What happens if a dependency is specified?

Declaring a dependency includes the dependent core in the build. More specifically, the following sections specified in the `default` target of the dependent core are included:

- `filesets`: File sets to include.
- `hooks`: A list of hooks to execute.
- `generate`: List of generators.
- `parameters`: List of available parameters.
- `vpi`: List of VPI objects.

Notably *not* included are the `tools`, `toplevel`, `description`, and `default_tool` sections of the `default` target. Also, no target other than `default` is considered when including a dependency.

Version constraints

Version constraints specify which version of a dependent core can be used, and which versions are incompatible.

Within a *core file*, version constraints are expressed by prefixing a core name with a version comparison operator. The following version comparison operators are available.

Table 1: Version comparison operators

Operator	Meaning	Example
=	exactly	`=fusesoc:examples:blinky:1.2`: exactly version 1.2
<	less (lower) than	`<fusesoc:examples:blinky:1.2`: any version before 1.2, e.g. 0.9
<=	at most (less than or equal to)	`<=fusesoc:examples:blinky:1.2`: at most version 1.2, e.g. 0.9, or 1.2
>=	at least (greater than or equal to)	`>=fusesoc:examples:blinky:1.2`: version 1.2, or any newer version, e.g. 1.2, or 10.7
>	more (higher) than	`>fusesoc:examples:blinky:1.2`: any version after 1.2, e.g. 1.3, or 10.7
^	Caret requirement: any version less than the next major version (see below)	`^fusesoc:examples:blinky:1.2`: >=1.2.0 <2.0.0
~	Tilde requirement: allow updates to the current version (see below)	`~fusesoc:examples:blinky:1.2`: >=1.2.0 <1.3.0

Notes:

- If no operator is specified, then = is assumed. So the = operator is effectively optional.
- If no version number is given any version is accepted, i.e. >= `0.0.0`.

Caret requirements

Caret requirements allow semantic versioning-compatible updates to a specified version. An update is allowed if the new version number does not modify the left-most non-zero digit in the major, minor, patch grouping.

Tilde requirements

Tilde requirements specify a minimal version with some ability to update. If you specify a major, minor, and patch version or only a major and minor version, only patch-level changes are allowed. If you only specify a major version, then minor- and patch-level changes are allowed.

Semantic versioning (SemVer)

A common scenario when declaring dependency is the following: "Core B depends on a version of core A which has the same interface as version 1.0.0, but may contain additional bug fixes in the implementation of that interface." Version numbers and dependencies alone cannot express this relationship, as they are (by default) meaningless. Equally, tools have a very hard time determining such compatibility accurately. Instead, humans are needed to attach meaning to version numbers, and that's where semantic versioning comes in.

Semantic versioning is a convention that gives meaning to version numbers. Being a convention, semantic versioning is not enforced by tooling, but relies on cooperation and a shared understanding between authors of reusable IP cores. Effectively, semantic versioning allows authors to encode in the version number information such as "this version breaks API compatibility", "this version is backwards compatible with a certain previous version", etc.

A detailed explanation of semantic versioning is available at semver.org. The basics, however, are quickly explained. Semantic versioning expects version numbers with three components, MAJOR, MINOR, and PATCH, such as 1.0.3. With this structure in place, follow these guidelines:

> Given a version number MAJOR.MINOR.PATCH, increment the:
>
> 1. MAJOR version when you make incompatible API changes,

2. MINOR version when you add functionality in a backwards compatible manner, and
3. PATCH version when you make backwards compatible bug fixes.

Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

—Semantic Versioning 2.0.0 (Summary)

1.5.4 Filters: Make system-wide modifications to the EDAM structure

The last thing FuseSoC does before handing over to Edalize, is to prepare an EDAM file containing a description of the complete system and everything that the EDA tools need to know. It is sometimes useful to make system-wide changes after the system is assembled, and this is where filters come in. Filters are additional tasks that can be run to analyze and modify the EDAM structure, and through that structure the filters have access to the complete dependency tree and all source files.

FuseSoC ships with a few built-in filters for generally useful tasks:

- autotype: Sets file types according to the file name prefix for files that don't have an explicit file type set.
- custom: Runs the command specified with the environment variable *FUSESOC_CUSTOM_FILTER*. Two arguments are passed to the command. The first specifies the EDAM (yaml) file the custom command is supposed to read. The second specifies the name of the file that the filter should use to return the modified EDAM struct.
- dot: Creates a GraphViz dot file of the dependency tree

All filters are run from the work root directory.

Note

Technically, an Edalize frontend can perform the exact same task as a FuseSoC EDAM filter. The difference is more philosophical in that a filter can be seen as something that fixes up the system before it is ready to be consumed by an EDA tool, while an Edalize frontend typically *is* an EDA tool. The filters will also possibly have access to more FuseSoC internals in the future.

Using filters

There are three way to enable which filters to be applied. These three options have various use-cases.

Filters in targets

Filters specified in a target section of a core are typically required for the the build to work. In order to enable a filter for a target, add a list of filters using the *filters* key.

```
# An excerpt from a core file.
targets:
  sim:
    # ...
    filters: [autotype, dot] # Apply the autotype and dot filters in that order
```

Filters in the configuration file

Filters can be set in the configuration file as a list of space-separated strings in the *main* section. These can be set with the fusesoc config cli, e.g. *fusesoc config filters "filter1 filter2"*. Filters set in the config file are typically used when all targets of the cores in the workspace need some filter to be applied, or just as a convenienence for users who like to have som particular filter always enabled. These filters are applied after the ones from the core file targets.

Filters on the command-line

It is also possible to set filters on the command-line. This is typically used for one-off filters, e.g. to generate some debug info. They are enabled with the *–filter* parameters, e.g. *fusesoc run –filter=dot –target=sim ...*. The *–filter* parameter can be specified multiple times to add more filters. Filters on the command-line are applied after the ones from the configuration file.

Creating additional filters

A filter is a Python module that contains a class with the same name as the module, but capitalized. The class needs to implement a function called *run* which takes the EDAM struct as the first argumen and the work root as the second argument. The function needs to return the new EDAM struct even if it is unmodified.

Filter template:

```
import logging
import os

logger = logging.getLogger(__name__)


class Customfilter:
    def run(self, edam, work_root):
        # Print file sizes of all verilog files in the system
        for f in edam["files"]:
            if f["file_type"].startswith("verilogSource"):
                size = os.path.getsize(os.path.join(work_root, f["name"]))
                print(f"{f['name']} is {size} bytes")

        # Add an additional parameter
        edam["parameters"]["my_number"] = {"datatype" : "int",
                                           "paramtype": "vlogdefine",
                                           "default" : 5446}

        # Change the system name
        edam["name"] = "bestsystemever"

        # Return modified EDAM struct
        return edam
```

FuseSoC implements support for implicit namespace packages (https://peps.python.org/pep-0420/) This means that subclasses that logically belong to FuseSoC can be distributed over several physical locations and is something we can take advantage of to add new filters outside of the FuseSoC code base.

In order to do that we will create a directory structure that mirrors the structure of FuseSoC like the example below:

```
externalplugin/
    fusesoc/
        filters/
            customfilter.py
            anothercustomfilter.py
```

There are two common options for making the above *customfilter.py* and *anothercustomfilter.py* available to FuseSoC.

The first way is to add the *externalplugin* path to `PYTHONPATH`. The other is to add a *setup.py* in the *externalplugin* directory and install the filter plugin with pip as with other Python packages.

A *setup.py* in its absolutely most minimal form is listed below and is enough to install the plugin as a package in development mode using `pip install --user -e .` from the *externalplugin* directory.:

```
from setuptools import setup
setup()
```

A real *setup.py* like the one used by FuseSoC normally contains a lot more information.

1.5.5 Flags: constraints in dependencies

Flags in FuseSoC are global boolean variables that influence the dependency resolution and other aspects of the build. Flags can be used in multiples places in *core files* to take conditional actions. For example, dependencies and files can be included depending on a flag, different toplevels can be selected, and much more.

Facts about flags:

- Flags are boolean variables, they are either "set" (true) or "unset" (false).
- Flags are global, i.e. they apply to the whole build. All cores see the same flags under the same name.
- A valid flag name starts with a letter, and consists only of letters from the English alphabet, numbers, and the underscore (_).
- Flag names are case-sensitive. It is recommended to use lower-case letters only.

Setting flags

Built-in flags are made available automatically. User-defined flags can be set as a dict of key/value pairs in the flags section of a target. These are turned into flags called <*key*>_<*value*> internally. This is intended to set default values for flags that normally needs to have them define. Finally, flags can be applied on the command line and these override user-defined flags in targets and built-in flags.

Built-in flags

When running `fusesoc run` some flags are automatically made available.

- `tool_TOOLNAME`: A flag that is set only if a particular tool is used during the build process. `TOOLNAME` is replaced with the name of the used tool. For example, when building a design for Xilinx Vivado, `tool_vivado` will be set.
- `target_TARGETNAME`: A flag that is set if a particular target is being built. `TARGETNAME` is replaced with the name of the target that is being built. For example, if the target `synth` is being built, the flag `target_synth` will be set.

User-defined flags

Flags can be set when building a design with `fusesoc run` by passing the `--flag` argument.

To set a flag, specify only the flag name, or alternatively, prefix it with a plus, i.e. `--flag FLAGNAME` or `--flag +FLAGNAME`. To unset a flag, prefix the name of the flag with a hyphen, i.e. pass `--flag -FLAGNAME`.

The `--flag` argument can be used multiple times to set or unset more than one flag.

Examples:

```
# Set the flag "my_flag".
fusesoc run --flag "my_flag" fusesoc:examples:blinky:1.0.0

# Alternative: set the flag "my_flag"
```

(continues on next page)

(continued from previous page)

```
fusesoc run --flag "+my_flag" fusesoc:examples:blinky:1.0.0

# Set two flags, "my_flag" and "my_other_flag"
fusesoc run --flag "my_flag" --flag "my_other_flag" fusesoc:examples:blinky:1.0.0

# Unset the flag "my_flag"
fusesoc run --flag "-my_flag" fusesoc:examples:blinky:1.0.0
```

Note

Order matters! The FuseSoC command line is "context sensitive." Place the `--flags` argument after the `run` command, but before the name of the core you are building.

Default values for flags can be set directly in the flags section of a target, like this.

```
# An excerpt from a core file.
targets:
  sim:
    # ...
    flags:
      fpu : true # Always enable the FPU for simulations. This will turn into a flag␣
↪called fpu
      sram : sim # Use simulation models for SRAM. This will turn into a flag called␣
↪sram_sim
```

Using flags

Flags can be used in *core files* to influence the build as part of a CAPI2 expression.

- To test if a flag is set (true) and return `my_string` in that case, use the expression `my_flag ? (my_string)`.
- To test if a flag is not set (false) and return `my_string` in that case, use the expression `!my_flag ? (my_string)`.

Note

Quotation marks are required for strings starting with an exclamation mark (`!`), but *always recommended*.

CAPI2 expressions are *not* full ternary operators, i.e. no "else" branch is available. Use two inverted expressions for if/else support.

The following use cases show some ways in which flags can be used in core description files.

Use case: Conditional dependencies

Flags can be used to influence which dependencies are included in a build. A typical use case for this feature are dependencies which are tool-specific.

The following example shows how to depend on a core named `fusesoc:examples:xilinx-fifo` (presumably implementing a Xilinx-specific FIFO) only if the `vivado` tool is used. Otherwise a core providing a tool-agnostic implementation is used.

```
# An excerpt from a core file.
filesets:
  rtl:
    # ...
    depend:
      - "tool_vivado ? (fusesoc:examples:xilinx-fifo)"
      - "!tool_vivado ? (fusesoc:examples:generic-fifo)"
```

Use case: Conditionally include files

Similar to the way dependencies can be conditionally specified source files can also be included conditionally.

The following example shows how to include a file `rtl/fifo_xilinx.sv` only if Vivado is used, and `rtl/fifo_generic.sv` otherwise.

```
# An excerpt from a core file.
filesets:
  rtl:
    # ...
    files:
      - "tool_vivado ? (rtl/fifo_xilinx.sv)"
      - "!tool_vivado ? (rtl/fifo_generic.sv)"
```

Use case: Conditional filesets in a target

Use flags in CAPI2 expressions in the `targets.TARGETNAME.filesets` block to conditionally include file sets.

The code snippet below shows how to include a fileset `verilator_tb` only if the `verilator` tool is used; otherwise the fileset `any_other_tool_tb` is included.

```
# An excerpt from a core file.
targets:
  # ...
  tb:
    # ...
    filesets:
      # Always include the rtl and tb filesets.
      - "rtl"
      - "tb"
      # Include the verilator_tb fileset only if verilator is used.
      - "tool_verilator ? (verilator_tb)"
      # Include the any_other_tool_tb fileset for all other tools.
      - "!tool_verilator ? (any_other_tool_tb)"
```

Use case: Conditionally choose a toplevel

Flags can also be used to choose the name a design toplevel based on certain conditions.

In the following code snippet, the user-defined flag `experimental_toplevel` is used to switch between two toplevels.

```
# An excerpt from a core file.
name: "fusesoc:examples:my_core"
targets:
```

(continues on next page)

(continued from previous page)

```
  # ...
  synth:
    toplevel:
      - "experimental_toplevel ? (top_experimental)"
      - "!experimental_toplevel ? (top_production)"
    # ...
```

With this setup in place users can choose which toplevel they want to build by passing the `--flag` command line argument to `fusesoc`, as illustrated in the following example.

```
# Build top_experimental
fusesoc run --flag experimental_toplevel --target synth fusesoc:examples:my_core

# Build top_production
fusesoc run --target synth fusesoc:examples:my_core
```

Further use cases

Flags can be used in more places than shown here. To find all valid places where flags can be used, refer to the ref_capi2. Expressions with flags can be used whenever the data type is `StringWithUseFlags`, `StringWithUseFlagsOrDict`, or `StringWithUseFlagsOrList`.

1.5.6 Generators: produce and specialize cores on demand

Todo

This section was taken from older documentation and needs to be adjusted in style and content for the refactored user guide.

FuseSoC core files list files that are natively used by the backend, such as VHDL/(System)Verilog files, constraints, tcl scripts, hex files for $readmemh, etc. There are however many cases where these files need to be created from another format. Examples of this are Chisel/MyHDL/Migen source code which output verilog, C programs that are compiled into a format suitable to be preloaded into memories or any kind of description formats used to create HDL files. For these cases FuseSoC supports generators, which are a mechanism to generate core files on the fly during the FuseSoC build flow. Since there are too many custom programs to generate HDL files, it is not feasible to include all of them in FuseSoC. Instead they are implemented as stand-alone programs residing within cores, which can be invoked by FuseSoC. Generators consists of three parts:

- The generator itself, which is a stand-alone program residing inside a core. It needs to accept a yaml file with a defined structure described below as its first (and only) argument. The generator will output a valid .core file and output files in the directory where it is called.
- The core that contains a generator must define a section in its core file to let FuseSoC know that it has a generator and how to invoke it.
- A core using a generator must contain a section that describes which parameters to send to the generator, and each target must list which generators to use

Creating a generator

Generators can be written in any language. The only requirement is that they are executable on the command line and accepts a yaml file for configuration as its first argument. This also means that generators can be used outside of FuseSoC to create cores. The yaml file contains the configuration needed for the generator. The following options are defined for the configuration file.

Key	Description
gapi	Version of the generator configuration file API. Only 1.0 is defined
files_root	Directory where input files are found. FuseSoC sets this to the calling core's file directory
vlnv	Colon-separated VLNV identifier to use for the output core. A generator is free to ignore this and use another VLNV.
parameters	Everything contained under the parameters key should be treated as instance-specific configuration for the generator

Example yaml configuration file:

```
files_root: /home/user/cores/mysoc
gapi: '1.0'
parameters:
  masters:
    dbus:
      slaves: [sdram_dbus, uart0, gpio0, gpio1, spi0]
    or1k_i:
      slaves: [sdram_ibus, rom0]
  slaves:
    gpio0: {datawidth: 8, offset: 2432696320, size: 2}
    gpio1: {datawidth: 8, offset: 2449473536, size: 2}
    rom0: {offset: 4026531840, size: 1024}
    sdram_dbus: {offset: 0, size: 33554432}
    sdram_ibus: {offset: 0, size: 33554432}
    spi0: {datawidth: 8, offset: 2952790016, size: 8}
    uart0: {datawidth: 8, offset: 2415919104, size: 32}
vlnv: ::mysoc-wb_intercon:0
```

The above example is for a generator that creates verilog code for a wishbone interconnect.

Registering a generator

When FuseSoC scans the flattened dependency tree of a core, it will look for a section called *generators* in each core file. This section is used by cores to notify FuseSoC that they contain a generator and describe how to use it.

The generators section contain a map of generator sections so that each core is free to define multiple generators. The key of each generator decide its name

The following keys are valid in a generator section.

Key	Description
command	The command to run (relative to the core root) to invoke the generator. FuseSoC will pass a yaml configuration file as the first argument when calling the command.
interpreter	If the command requires an interpreter (e.g. python or perl), this will be used called, with the string specified in *command* as the first argument, and the yaml file as the second argument.
cache_type	If the result of the generator should be considered cacheable. Legal values are *none*, *input* or *generator*.
file_input_pa	All parameters that are file inputs to the generator. This option can be used when *cache_type* is set to *input* if fusesoc should track if these files change.

Example generator section from a CAPI2 core file

```
generators:
  wb_intercon_gen:
    interpreter: python
    command: sw/wb_intercon_gen
```

The above snippet will register a generator with the name wb_intercon_gen. This name will be used by cores that wish to invoke the generator. When the generator is invoked it will run *python /path/to/core/sw/wb_intercon_gen* from the sw subdirectory of the core where the generators section is defined.

Calling a generator

The final piece of the generators machinery is to run a generator with some specific parameters. This is done by creating a special section in the core that wishes to use a generator and adding this section to the targets that need it. Using the same example generator as previously, this section could look like the example below:

```
generate:
  wb_intercon:
    generator : wb_intercon_gen
    parameters:
      masters:
        or1k_i:
          slaves:
            - sdram_ibus
            - rom0
        dbus:
          slaves: [sdram_dbus, uart0, gpio0, gpio1, spi0]

      slaves:
        sdram_dbus:
          offset : 0
          size : 0x2000000

        sdram_ibus:
          offset: 0
          size: 0x2000000

        uart0:
          datawidth: 8
          offset: 0x90000000
          size: 32
```

(continues on next page)

(continued from previous page)

```
      gpio0:
        datawidth: 8
        offset: 0x91000000
        size: 2

      gpio1:
        datawidth: 8
        offset: 0x92000000
        size: 2

      spi0:
        datawidth: 8
        offset: 0xb0000000
        size: 8

      rom0:
        offset: 0xf0000000
        size: 1024
```

The above core file snippet will register a parametrized generator instance with the name wb_intercon. It will use the generator called *wb_intercon_gen* which FuseSoC has previously found in the dependency tree. Everything listed under the *parameters* key is instance-specific configuration to be sent to the generator.

Just registering a generate section will not cause the generator to be invoked. It must also be listed in the target and the generator to be used must be in the dependency tree. The following snippet adds the parameterized generator to the *default* target and adds an explicit dependency on the core that contains the generator. As CAPI2 cores only allow filesets to have dependencies, an empty fileset for this purpose must be created

```
filesets:
  wb_intercon_dep:
    depend:
      [wb_intercon]

targets:
  default:
    filesets : [wb_intercon_dep]
    generate : [wb_intercon]
```

When FuseSoC is launched and a core target using a generator is processed, the following will happen for each entry in the target's *generate* entry.

1. A key lookup is performed in the core file's *generate* section to find the generator configuration
2. FuseSoC checks that it has registered a generator by the name specified in the *generator* entry of the configuration.
3. FuseSoC calculates a unique VLNV for the generator instance by taking the calling core's VLNV and concatenating the name field with the generator instance name.
4. A directory is created under <cache_root>/generator_cache with a sanitized version of the calculated VLNV along with a SHA256 hash of the input yaml file data appended. This directory is where the output from the generator eventually will appear.
5. If the generator has *cache_type* set to *input* fusesoc will check if a cached output already exists. In this case item 6 and 7 will be omitted. See section *Generator Cache* for more information.

6. A yaml configuration file is created in the generator output directory. The parameters from the instance are passed on to this file. FuseSoC will set the files root of the calling core as *files_root* and add the calculated vlnv.
7. FuseSoC will switch working directory to the generator output directory and call the generator, using the command found in the generator's *command* field and with the created yaml file as command-line argument.
8. When the generator has successfully completed (or a cached run already exists), FuseSoC will scan the generator output directory for new .core files. These will be injected in the dependency tree right after the calling core and will be treated just like regular cores, except that any extra dependencies listed in the generated core will be ignored.
9. If the generator is marked as set as cacheable (*input* or *generator*) the directory (along with content) created under item 4 will be kept, otherwise it will be deleted.

Generator Cache

Instead of fusesoc rerunning a generator each time and producing the same result it is possible to configure fusesoc to cache generator output and try to detect if a new run would produce the same output. Since there is no generic way of doing this that will fit all generators a couple of different methods for caching and detecting changes are available.

The *generators* option *cache_type* is used for configuring type of caching. If set to *none* (or if option is omitted) no caching will be used. If set to *input* fusesoc will calculate a SHA256 hash of the generator input yaml file data and use this hash for detecting if something has changed and a rerun would be needed. This would happen if some data in the core file *generate* section, for instance *parameters*, has changed.

If *cache_type* is set to *generator* fusesoc will pass the responsibility for detecting if the previous run to the generator is still up to date. In this mode the generator will always be called and the output directory will be saved.

In addition, when *cache_type* is set to *input* it is also possible to configure fusesoc to detect changes in file input data to a generator. This is done by using the *generators* option *file_input_parameters* which tells fusesoc which parameters are used to pass input files to the generator.

Example *generators* section with *cache_type* and *file_input_parameters*:

```
generators:
  mytest_gen:
    interpreter: python
    command: mytest_gen.py
    cache_type: input
    file_input_parameters: file_input_param1 file_input_param2
```

Example *generate* section using the above generator.

```
generate:
  mytest:
    generator: mytest_gen
    parameters:
      some_param: 123
      file_input_param1: input_file_1
      file_input_param2: /path/to/input_file_2
```

In the above example fusesoc would calculate the SHA256 hash for *input_file_1* (relative *files_root*) and */path/to/input_file_2* (absolute path). This hash would then be saved in the generator cache directory in a file called *.fusesoc_file_input_hash*. During subsequent runs fusesoc would then compare the current input hash with the saved hash to determine if the generator output still is valid or if the generator needs to be run again.

If needed, the *generator_cache* directory under *cache_root* can be cleaned by running *fusesoc gen clean*.

1.5.7 Virtual Cores: Provide a common interface

Todo

Document virtual cores.

1.5.8 Mappings: Replace cores in the dependency tree

Mappings allow a user of a core to substitute dependencies of the core with other cores without having to edit any of the core or it's dependencies files. An example use case is making use of an existing core but substituting one of it's dependencies with a version that some desired changes (e.g. bug fixes).

If you are looking to provide a core with multiple implementations, virtual cores is the recommended and more semantic solution. See *Virtual Cores: Provide a common interface* for more information on virtual cores. Note: virtual cores can also be substituted in mappings.

Declaring mappings

Each core file can have one mapping. An example mapping core file:

```
name: "local:map:override_fpu_and_fifo"
mapping:
  "vendor:lib:fpu":  "local:lib:fpu"
  "vendor:lib:fifo": "local:lib:fifo"
```

The example above is a core file with only a mapping property, but any core file may contain a mapping in addition to other properties (e.g. filesets, targets & generators).

Applying mappings

To apply a mapping, provide the VLNV of the core file that contains the desired mapping with *fusesoc run*'s *–mapping* argument. Multiple mappings may be provided as shown below.

```
fusesoc run \
  --mapping local:map:override_fpu_and_fifo \
  --mapping local:map:another_mapping \
  vendor:top:main
```

1.5.9 Hooks: intercept the build process

Todo

Document hooks.

1.5.10 VPI Support

Todo

Document VPI support.

1.6 The FuseSoC package manager

1.6.1 Core libraries

A collection of one or more cores in a directory tree is called a core library. FuseSoC supports working with multiple core libraries. The locations of the libraries are specified in the FuseSoC configuration file, `fusesoc.conf`

To find a configuration file, FuseSoC will first look for `fusesoc.conf` in the current directory, and if there is no file there, it will search next in `$XDG_CONFIG_HOME/fusesoc` (i.e. `~/.config/fusesoc` on Linux and `%HOMEPATH%\.config\fusesoc` on Windows) and lastly in `/etc/fusesoc`

A specific configuration file can be selected in two ways:

- Set the `FUSESOC_CONFIG` environment variable to the path of the configuration file.
- Pass `--config <path>` on the command line.

The `--config` command-line option takes precedence over the `FUSESOC_CONFIG` environment variable.

By running `fusesoc library add fusesoc_cores https://github.com/fusesoc/fusesoc-cores` after FuseSoC is installed, the standard libraries will be installed, and a default configuration file will be created in `$XDG_CONFIG_HOME/fusesoc/fusesoc.conf` on Linux and `%HOMEPATH%\.config\fusesoc\fusesoc.conf` on Windows with the following contents:

```
[library.fusesoc-cores]
sync-uri = https://github.com/fusesoc/fusesoc-cores
sync-type = git
```

1.6.2 Core search order

Once FuseSoC has found its configuration file, it will parse the library sections in the order they appear in the file. Library sections are all sections named *library.<library name>*. The following keys are valid in the library sections.

location

Specifies the library's location in the file system (required)

auto-sync

Boolean value specifying if the library should be automatically updated when running *fusesoc library update* (optional. defaults to *true*)

sync-uri

The URI for non-local libraries where to fetch the library (optional)

sync-type

The type of library. Can be set to *git* or *local*. A missing value indicates a *local* library. (optional)

Additional library locations can be added on the command line by setting the `--cores-root` parameter when FuseSoC is launched. The library locations specified from the command-line will be parsed after those in `fusesoc.conf`

For each library location, FuseSoC will recursively search for files with a *.core* suffix. Each of these files will be parsed and added to the in-memory FuseSoC database if they are valid `.core` files.

Several `.core` files can reside in the same directory and they will all be parsed.

If several cores with the same VLNV identifier are encountered the latter will replace the former. This can be used to override cores in a library with an alternative core in another library by specifying them in a library that will be parsed later, either temporarily by adding `--cores-root` to the command-line, or permanently by adding the other library at the end of fusesoc.conf

If FuseSoC encounters a file called *FUSESOC_IGNORE* in a directory, this directory and all subdirectories will be ignored.

1.7 Common Problems and Solutions

1.7.1 Making changes to cores in a library

A common situation is that a user wants to use their own copy of a core, instead of the one provided by a library, for example to fix a bug or add new functionality. The following steps can be used to achieve this:

Example. Replace a core in a library with a user-specified version

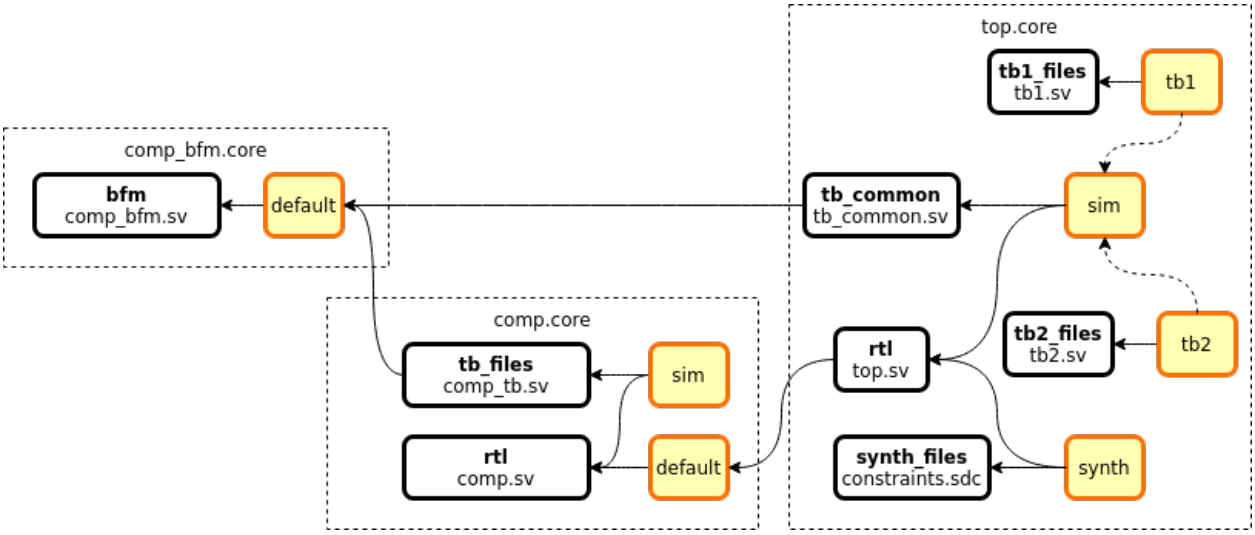
1. Create a new directory to keep the user-copies of the cores (this directory will be referred to as `$corelib` from now on)
2. Download the core source (the repository or URL can be found in the `[provider]` section of the original core)
3. *If the downloaded core already contains a .core file, this step is ignored* Copy the original .core file to the root of the downloaded core. Edit the file and remove the `[provider]` section. (This will stop FuseSoC from downloading the core and use files from the directory containing the .core file instead)
4. Add `$corelib` to the end of your library search path, either by editing `fusesoc.conf` or by adding `--cores-root=$corelib` to the command-line arguments
5. Verify that the new core is found by running fusesoc core-info $core. Check the output to see that "Core root:" is set to the directory where the core was downloaded

1.7.2 Dependency tree for a core with optional components

Many cores have a part that is only used in some flows. This could for example be a BFM, VIP or some kind of behavioral model that is only used in simulation flows. Or it could be timing constraints for synthesis. As long as these are only by the core itself, it's easiest to put them into different filesets and let each simulation or synthesis target pick the right subset of filesets.

A more complicated situation arises when a user uses this core as a dependency and wants to have different filesets for different flows in the toplevel core. In this case, it is typically better to split out the optional part into its own cores and have the toplevel filesets depend on the different cores.

Example Let's assume a core (comp) comes with a BFM. We want to use the BFM when doing block-level simulations of the core and when doing full system simulations which uses this core. We don't want to have the BFM present when doing full system synthesis. The most general solution is to split out the BFM to a separate core file. The following example shows a setup with a component (comp.core) that has a BFM (comp_bfm.core) which is used inside a larger system (top.core). The component has a testbench target (sim) and the larger system has two testbenches (tb1 and tb2) which both uses the BFM and the component. The larger system can also be synthesized without the BFM.



Listing 3: comp_bfm.core

```
CAPI=2:

name : ::comp_bfm:0

filesets:
  bfm:
    files: {comp_bfm.sv : {file_type : systemVerilogSource}}

targets:
  default:
    filesets : [bfm]
```

Listing 4: comp.core

```
CAPI=2:

name : ::comp:0

filesets:
  rtl:
    files: {comp.sv : {file_type : systemVerilogSource}}

  tb_files:
    files: {comp_tb.sv : {file_type : systemVerilogSource}}
    depend: [comp_bfm]

targets:
  default:
    filesets : [rtl]

  sim:
    filesets : [rtl, tb_files]
```

Listing 5: top.core

```
CAPI=2:

name : ::top:0

filesets:
  rtl:
    files: {top.sv : {file_type : systemVerilogSource}}
    depend: [comp]

  tb_common:
    files: {tb_common.sv : {file_type : systemVerilogSource}}
    depend: [comp_bfm]

  tb1_files:
    files: {tb1.sv : {file_type : systemVerilogSource}}

  tb2_files:
    files: {tb1.sv : {file_type : systemVerilogSource}}

  synth_files:
    files: {constraints.sdc : {file_type : SDC}}

targets:
  sim: &sim
    filesets : [rtl, tb_files]

  tb1:
    <<: *sim
    filesets_append : [tb1_files]

  tb2:
    <<: *sim
    filesets_append : [tb2_files]

  synth:
    filesets : [rtl, synth_files]
```

An alternative solution to the above, is to use flags, as described in *Flags: constraints in dependencies*, where the inclusion of the optional files are controlled by flags. These flags can be assigned default values in the top level targets for convenience.

CHAPTER

TWO

FUSESOC REFERENCE MANUAL

2.1 CAPI2

Core API Version 2

2.1.1 Properties

- **description** *(string)*: Short description of core.
- **license**
  - **One of**
    - *string*: SPDX license identifier. See https://spdx.org/licenses/ for valid values.
    - *object*: Custom defined license. Cannot contain additional properties.
      - **name** *(string, required)*
      - **text** *(string, required)*
- **filesets**: Refer to #/$defs/filesets.
- **generate**: Refer to #/$defs/generate.
- **generators**: Refer to #/$defs/generators.
- **name** *(string, required)*: VLNV identifier for core.
- **parameters**: Refer to #/$defs/parameters.
- **provider**: Refer to #/$defs/provider.
- **scripts**: Refer to #/$defs/scripts.
- **targets**: Refer to #/$defs/targets.
- **vpi** *(object)*: A VPI (Verilog Procedural Interface) library is a shared object that is built and loaded by a simulator to provide extra Verilog system calls. This section describes what files and external libraries to use for building a VPI library.
  - **^.+$** *(object)*: Cannot contain additional properties.
    - **^filesets(_append)?$**: Filesets containing files to use when compiling the VPI library. Refer to #/$defs/string_array.
    - **^libs(_append)?$**: External libraries to link against. Refer to #/$defs/string_array.
- **virtual**: VLNV of a virtual core provided by this core. Versions are currently not supported, only the VLN part is used. Refer to #/$defs/string_array.
- **mapping** *(object)*: .

  - **^.+$** *(string)*

2.1.2 Definitions

- **string_array** *(array)*
  - **Items** *(string)*
- **any_type**
  - **Any of**
    - *string*
    - *number*
    - *boolean*
    - *array*
    - *object*
- **files** *(array)*: Files in fileset. Length must be at least 1.
  - **Items**
    - **One of**
      - *string*
      - *object*: Number of properties must be equal to 1. Cannot contain additional properties.
      - **^.+$** *(object)*: Path to file. Cannot contain additional properties.
      - **define** *(object)*: Defines to be used for this file. These defines will be added to those specified in the target parameters section. If a define is specified both here and in the target parameter section, the value specified here will take precedence. The parameter default value can be set here with `param=value`. Cannot contain additional properties.
      - **^.+$**
      - **Any of**
      - *string*
      - *number*
      - *boolean*
      - **is_include_file** *(boolean)*: Treats file as an include file when true.
      - **include_path** *(string)*: Explicitly set an include directory, relative to core root, instead of the directory containing the file.
      - **file_type** *(string)*: File type. Overrides the file_type set on the containing fileset.
      - **logical_name** *(string)*: Logical name, i.e. library for VHDL/SystemVerilog. Overrides the logical_name set on the containing fileset.
      - **tags**: Tags, special file-specific hints for the backends. Appends the tags set on the containing fileset. Refer to #/$defs/string_array.
      - **copyto** *(string)*: Copy the source file to this path in the work directory.
- **filesets** *(object)*: A fileset represents a group of files with a common purpose. Each file in the fileset is required to have a file type and is allowed to have a logical_name which can be set for the whole fileset or individually for each file. A fileset can also have dependencies on other cores, specified in the depend section. Cannot contain additional properties.

  - **^.+$** *(object)*: Name of fileset. Cannot contain additional properties.
    - **file_type** *(string)*: Default file_type for files in fileset.
    - **logical_name** *(string)*: Default logical_name (i.e. library) for files in fileset.
    - **tags**: Default tags for files in fileset. Refer to #/$defs/string_array.
    - **^files(_append)?$**: Refer to #/$defs/files.
    - **^depend(_append)?$**: Dependencies of fileset. Refer to #/$defs/string_array.
- **generate** *(object)*: The elements in this section each describe a parameterized instance of a generator. They specify which generator to invoke and any generator-specific parameters.
  - **^.+$** *(object)*: Name of generator to use. Cannot contain additional properties.
    - **generator** *(string, required)*: The generator to use. Note that the generator must be present in the dependencies of the core.
    - **position** *(string)*: Where to insert the generated core. Legal values are *first*, *prepend*, *append* or *last*. *prepend* (*append*) will insert core before (after) the core that called the generator. Must be one of: "first", "prepend", "append", or "last".
    - **parameters** *(object)*: Generator-specific parameters. `fusesoc gen show $generator` might show available parameters. .
- **generators** *(object)*: Generators are custom programs that generate FuseSoC cores. They are generally used during the build process, but can be used stand-alone too. This section allows a core to register a generator that can be used by other cores.
  - **^.+$** *(object)*: Name of generator. Cannot contain additional properties.
    - **command** *(string, required)*: The command to run (relative to the core root).
    - **interpreter** *(string)*: If the command needs a custom interpreter (such as python) this will be inserted as the first argument before command when calling the generator. The interpreter needs to be on the system PATH; specifically, shutil.which needs to be able to find the interpreter).
    - **cache_type** *(string)*: If the result of the generator should be considered cacheable. Legal values are *none*, *input* or *generator*. Must be one of: "none", "input", or "generator".
    - **file_input_parameters** *(string)*: All parameters that are file inputs to the generator. This option can be used when *cache_type* is set to *input* if fusesoc should track if these files change.
    - **description** *(string)*: Short description of the generator, as shown with `fusesoc gen list`.
    - **usage** *(string)*: A longer description of how to use the generator, including which parameters it uses (as shown with `fusesoc gen show $generator`).
- **parameters** *(object)*: Available parameters.
  - **^.+$** *(object)*: Cannot contain additional properties.
    - **datatype** *(string, required)*: Parameter datatype. Legal values are *bool*, *file*, *int*, *str*. *file* is same as *str*, but prefixed with the current directory that FuseSoC runs from. Must be one of: "bool", "file", "int", "real", or "str".
    - **default**: Default value.
      - **One of**
      - *boolean*
      - *string*
      - *number*

        * **description** *(string)*: Description of the parameter, as can be seen with `fusesoc run --target=$target $core --help`.
        * **paramtype** *(string, required)*: Specifies type of parameter. Legal values are *cmdlinearg* for command-line arguments directly added when running the core, *generic* for VHDL generics, *plusarg* for verilog plusargs, *vlogdefine* for Verilog `` `define `` or *vlogparam* for verilog top-level parameters. All paramtypes are not valid for every backend. Consult the backend documentation for details.
        * **scope** *(string)*: **Not used** : Kept for backwards compatibility.

- **provider** *(object)*: Provider of core.
  - **Any of**
    * *object*: github Provider. Cannot contain additional properties.
      · **name** *(string, required)*: Must be: `"github"`.
      · **user** *(string, required)*
      · **repo** *(string, required)*
      · **version** *(string, required)*
      · **patches**: Refer to #/$defs/string_array.
      · **cachable** *(boolean)*
    * *object*: local Provider. Cannot contain additional properties.
      · **name** *(string, required)*: Must be: `"local"`.
      · **patches**: Refer to #/$defs/string_array.
      · **cachable** *(boolean)*
    * *object*: git Provider. Cannot contain additional properties.
      · **name** *(string, required)*: Must be: `"git"`.
      · **repo** *(string, required)*
      · **version** *(string)*
      · **patches**: Refer to #/$defs/string_array.
      · **cachable** *(boolean)*
    * *object*: opencores Provider. Cannot contain additional properties.
      · **name** *(string, required)*: Must be: `"opencores"`.
      · **repo_name** *(string, required)*
      · **repo_root** *(string, required)*
      · **revision** *(string, required)*
      · **patches**: Refer to #/$defs/string_array.
      · **cachable** *(boolean)*
    * *object*: svn Provider. Cannot contain additional properties.
      · **name** *(string, required)*: Must be: `"svn"`.
      · **url** *(string, required)*
      · **revision** *(string)*

· **ignore_externals** *(boolean)*

· **patches**: Refer to #/$defs/string_array.

· **cachable** *(boolean)*

* *object*: url Provider. Cannot contain additional properties.

· **name** *(string, required)*: Must be: `"url"`.

· **url** *(string, required)*

· **user-agent** *(string)*

· **verify_cert** *(string)*

· **filetype** *(string, required)*

· **patches**: Refer to #/$defs/string_array.

· **cachable** *(boolean)*

- **scripts** *(object)*: A script specifies how to run an external command that is called by the hooks section together with the actual files needed to run the script. Scripts are always executed from the work root.
  - **^.+$** *(object)*: Cannot contain additional properties.
    * **env** *(object)*: Map of environment variables to set before launching the script. Cannot contain additional properties.
      · **^.+$** *(string)*
    * **^cmd(_append)?$**: List of command-line arguments. Refer to #/$defs/string_array.
    * **^filesets(_append)?$**: Filesets needed to run the script. Refer to #/$defs/string_array.
- **targets** *(object)*: A target is the entry point to a core. It describes a single use-case and what resources that are needed from the core such as file sets, generators, parameters and specific tool options. A core can have multiple targets, e.g. for simulation, synthesis or when used as a dependency for another core. When a core is used, only a single target is active. The *default* target is a special target that is always used when the core is being used as a dependency for another core or when no `--target=` flag is set.
  - **^.+$** *(object)*: Cannot contain additional properties.
    * **default_tool** *(string)*: Default tool to use unless overridden with `--tool=` This key is used by the Edalize Tool API and is ignored if the Flow API is used instead.
    * **description** *(string)*: Description of the target.
    * **flow** *(string)*: Edalize backend flow to use for target. Setting this key enables the flow API instead of the legacy Tool API.
    * **flow_options** *(object)*: Tool- and flow-specific options. Used by the Flow API. The Edalize documentation contains information on available options for different flows (https://edalize.readthedocs.io/en/latest/edam/api.html#flow-options).
      · **^.+$**: Refer to #/$defs/any_type.
    * **hooks** *(object)*: Script hooks to run when target is used. Cannot contain additional properties.
      · **^pre_build(_append)?$**: Scripts executed before the *build* phase. Refer to #/$defs/string_array.
      · **^post_build(_append)?$**: Scripts executed after the *build* phase. Refer to #/$defs/string_array.

        - **^pre_run(_append)?$**: Scripts executed before the *run* phase. Refer to #/$defs/string_array.
        - **^post_run(_append)?$**: Scripts executed after the *run* phase. Refer to #/$defs/string_array.

    * **tools** *(object)*: Tool-specific options for target. Used by the legacy Tool API. The contents of this section is handled by Edalize, and a list of available tool options for each tool can be found in the Edalize documentation (https://edalize.readthedocs.io/en/latest/edam/api.html#tool-options).
        - **^.+$** *(object)*
        - **^.+$**: Refer to #/$defs/any_type.
    * **toplevel**: Top-level module. Normally a single module/entity but can be a list of several items.
        - **Any of**
        - *string*
        - : Refer to #/$defs/string_array.
    * **flags** *(object)*: Default values of flags.
        - **^.+$**: Refer to #/$defs/any_type.
    * **^filesets(_append)?$**: File sets to use in target. Refer to #/$defs/string_array.
    * **^filters(_append)?$**: EDAM filters to apply. Refer to #/$defs/string_array.
    * **^generate(_append)?$** *(array)*: Parameterized generators to run for this target with optional parametrization.
        - **Items**
        - **Any of**
        - *string*
        - *object*
    * **^parameters(_append)?$**: Parameters to use in target. The parameter default value can be set here with `param=value`. Refer to #/$defs/string_array.
    * **^vpi(_append)?$**: VPI modules to build and include for target. Refer to #/$defs/string_array.

2.2 Migration guide

FuseSoC strives to be backwards-compatible, but as new features are added to FuseSoC, some older features become obsolete. This chapter contains information on how to migrate away from deprecated features to keep the core description files up-to-date with the latest best practices.

2.2.1 Migrating from CAPI1 to CAPI2

Why

FuseSoC's *.core* files are written in a "language" called CAPI. The current version of CAPI is version 2, also called CAPI2. Going forward, only the newer CAPI2 file format will be supported, which simplifies the use and implementation of FuseSoC greatly.

When

In FuseSoC 1.x, both CAPI1 and CAPI2 are supported. Starting with FuseSoC 2 only CAPI2 will be supported. To be able to update to the next version of FuseSoC seamlessly you need to migrate your existing CAPI1 core files to CAPI2. We recommend doing this migration *now* while still running FuseSoC 1.x.

How

FuseSoC ships with an automated conversion tool from CAPI1 to CAPI2, which provides a solid starting point for the conversion process. However, even though CAPI2 supports almost all features of CAPI1, there isn't always a 1:1 mapping between the two formats. Therefore the automatic conversion won't be always correct, and a bit of manual cleanup work will be needed.

To convert a single core file, follow these steps:

1. Ensure you're running the latest version of FuseSoC 1.x.
2. Ensure that you have a backup of your core file, or have committed the current version in a version control system.
3. Run `fusesoc migrate-capi1-to-capi2 --inplace your_core_file.core` to convert the file automatically.
4. Open `your_core_file.core` to check the conversion output and adjust it as necessary (e.g. add back comments, adjust the ordering of statements, etc.)

You can find documentation on the CAPI1 as well as on CAPI2 in the reference manual. We also recommend to have a look at the *Writing core files* section in the user guide for current best practices on writing CAPI2 core files.

To convert all core files in a directory, Linux users can run the following command:

```
find your_coredir -iname '*.core' -exec fusesoc migrate-capi1-to-capi2 --nowarn --
↪inplace {} \;
```

If you get stuck in the conversion process, or if a CAPI1 feature you rely on isn't available in CAPI2, please get in touch by filing an issue on GitHub.

2.2.2 Migrating from .system files

Why

The synthesis backends required a separate .system file in addition to the .core file. There is however very little information in the .system file, it was never properly documented and some information is duplicated from the .core file. For these reasons a decision was made to drop the .system file and move the relevant information to the .core file instead.

When

`.system` files are no longer needed as of FuseSoC 1.6

The `.system` file will still be supported for some time to allow users to perform the migration, but any equivalent options in the `.core` file will override the ones in `.system`

How

Perform the following steps to migrate from .system files

1. Move the `backend` parameter from the `main` section in the `.system` file to the `main` section in the `.core` file
2. Move the backend section (i.e. `icestorm`, `ise`, `quartus` or `vivado`) to the `.core` file
3. Move `pre_build_scripts` from the `scripts` section in the `.system` file to `pre_synth_scripts` in the `scripts` section in the `.core` file.

4. Move `post_build_scripts` from the `scripts` section in the `.system` file to `post_impl_scripts` in the `scripts` section in the `.core` file.

2.2.3 Migrating from plusargs

Why

Up until FuseSoC 1.3, verilog plusargs were the only way to set external run-time parameters. Cores could register which plusargs they supported through the `plusargs` section. This mechanism turned out to be too limited, and in order to support public/private parameters, defines, VHDL generics etc, `parameter` sections were introduced to replace the `plusargs` section.

When

`parameter` sections were introduced in FuseSoC 1.3

The `plusargs` section is still supported to allow time for migrations

How

Entries in the `plusargs` section are described as `<name> = <type> <description>`. For each of these entries, create a new section with the following contents

```
[parameter <name>]
datatype = <type>
description = <description>
paramtype = plusarg
```

The `parameter` sections also support the additional tags `default`, to set a default value, and `scope` to select if this parameter should be visible to other cores (`scope=public`) or only when this core is used as the toplevel (`scope=private`).

2.2.4 Migrating to filesets

Why

Originally only verilog source files were supported. In order to make source code handling more generic, filesets were introduced. Filesets are modeled after IP-XACT filesets and each fileset lists a group of files with similar purpose. Apart from supporting more file types, the filesets contain some additional control over when to use the files. The verilog section is still supported for some time to allow users to perform the migration.

When

`fileset` sections were introduced in FuseSoC 1.4

The `verilog` section is still supported to allow time for migrations

How

Given a `verilog` section with the following contents:

```
[verilog]
src_files = file1.v file2.v
include_files = file3.vh file4.vh
tb_src_files = file5.v file6.v
tb_include_files = file7.vh file8.vh
tb_private_src_files = file9.v file10.v
```

these will be turned into multiple file sets. The names of the file sets are not important, but should reflect the usage of the files.

```
[fileset src_files]
files = file1.v file2.v
file_type = verilogSource

[fileset include_files]
files = file3.vh file4.vh
file_type = verilogSource
is_include_file = true

[fileset tb_src_files]
files = file5.v file6.v
file_type = verilogSource
usage = sim

[fileset tb_include_files]
files = file7.vh file8.vh
file_type = verilogSource
is_include_file = true
usage = sim

[fileset tb_private_src_files]
files = file9.v file10.v
file_type = verilogSource
scope = private
usage = sim
```

If not specified, `usage = sim synth` and `scope = public`

These filesets can be further combined by setting some per-file attributes

```
[fileset src_files]
files =
 file1.v
 file2.v
 file3.vh[is_include_file]
 file4.vh[is_include_file]
file_type = verilogSource

[fileset public_tb_files]
files = file5.v file6.v file7.vh[is_include_file] file8.vh[is_include_file]
file_type = verilogSource
usage = sim

[fileset tb_files]
files = file9.v file10.v
file_type = verilogSource
scope = private
usage = sim
```

`file_type` can also be overridden on a per-file basis (e.g. `file2.v[file_type=verilogSource-2005] file3.vh[is_include_file,file_type=systemVerilogSource]`), but scope and usage are set for each fileset.

2.2.5 Migrating from verilator define_files

Why

Files specified as `define_files` in the verilator core section were treated as verilog files containing `` `define `` statements to C header files with equivalent #define statements. While there are use-cases for this functionality, the actual implementation is limited and makes assumptions that makes it difficult to maintain in the FuseSoC code base. The decision is therefore made to deprecate this functionality and instead require the user to make the conversion.

When

`verilator define_files` are no longer converted in FuseSoC 1.7

How

The following stand-alone Python script will perform the same function. It can also be executed as a `pre_build` script to perform the conversion automatically before a build

```
def convert_V2H( read_file, write_file):
    fV = open (read_file,'r')
    fC = open (write_file,'w')
    fC.write("//File auto-converted the Verilog to C. converted by FuseSoC//\n")
    fC.write("//source file --> " + read_file + "\n")
    for line in fV:
        Sline=line.split('`',1)
        if len(Sline) == 1:
            fC.write(Sline[0])
        else:
            fC.write(Sline[0]+"#"+Sline[1])
    fC.close
    fV.close

import sys
if __name__ == "__main__":
    convert_V2H(sys.argv[1], sys.argv[2])
```

2.2.6 Redefining build_root

Why

As an aid for scripts executed during the build process, a number of environment variables were defined. Unfortunately this was done without too much thought and as time moved on, some of these turned out to be a maintenance burden without bringing much benefit, and in some cases without ever being used.

At the same time, the introduction of VLNV and dependency ranges has introduced non-determinism in where the output of a build ends up. For these reasons, it was determined to redefine the rarely used *build_root* variable to point to the the directory containing the work root and exported files. A *–build-root* command-line switch is introduced to explicitly set a build_root. Setting *build_root* in *fusesoc.conf* will keep working the same way as before, but the command-line switch takes precedence. CAPI1 cores will no longer export the *BUILD_ROOT* environment variable.

These changes affects the following cases:

- Relying on the *BUILD_ROOT* variable in scripts called from CAPI1 cores.

When

build_root was redefined after the release of FuseSoC 1.9.1

How

Any scripts that previously relied on `$BUILD_ROOT` will have to be updated. Note that due to other changes in FuseSoC most of them were unlikely to work at this point anyway.

2.3 Glossary

In the context of FuseSoC some terms have special meaning. This glossary section explains some of the jargon used.

CAPI2
: Short for "Core API version 2." The schema or "language" used in *core files*. The ref_capi2 describes the syntax in detail.

core
: A core is a reasonably self-contained, reusable piece of IP, such as a FIFO implementation. See also *FuseSoC's basic building block: cores*.

core file
core description file
: A file describing a *core*, including source files, available targets, etc. A core file is a *YAML* file which follows *CAPI2* schema. Core files names must end in `.core`.

semantic versioning
SemVer
: Semantic versioning is a convention to give meaning to version numbers. See *Semantic versioning (SemVer)* and semver.org.

stage
build stage
: See *Build stages*.

target
: See *Targets*.

tool
tool flow
: See *Tool flows*.

VLNV
: Vendor, Library, Name, and Version: the format used for *core* names. In core names, the four parts are separated by colons, forming a name like `vendor:library:name:version`.

  See also *The core name, version, and description*.

YAML
: YAML is (among other things) a markup language, commonly used for configuration files. It is used in FuseSoC in various places, especially for *core description files* and for EDAM files.

  Read more about YAML on Wikipedia or on yaml.org.

CHAPTER

THREE

FUSESOC DEVELOPER'S GUIDE

3.1 Development Setup

> **Note**
>
> To make changes to a backend, e.g. to the way a simulator or synthesis tool is called, you need to modify edalize, not fusesoc. Edalize is a separate project, see https://github.com/olofk/edalize for more information.

3.1.1 Get the code

The FuseSoC source code is maintained in a git repository hosted on GitHub. To improve FuseSoC itself, or to test the latest unreleased version, it is necessary to clone the git repository first.

```
cd your/preferred/source/directory
git clone https://github.com/olofk/fusesoc
```

3.1.2 Setup development environment

> **Note**
>
> If you have already installed FuseSoC, remove it first using `pip3 uninstall fusesoc`.

To develop FuseSoC and test the changes, the fusesoc package needs to be installed in editable or development mode. In this mode, the `fusesoc` command is linked to the source directory, and changes made to the source code are immediately visible when calling `fusesoc`.

```
# Install all Python packages required to develop fusesoc
pip3 install --user -r dev-requirements.txt

# Install Git pre-commit hooks, e.g. for the code formatter and lint tools
pre-commit install

# Install the fusesoc package in editable mode
pip3 install --user -e .
```

> **Note**

All commands above use Python 3 and install software only for the current user. If, after this installation, the `fusesoc` command cannot be found adjust your `PATH` environment variable to include `~/.local/bin`.

After this installation is completed, you can

- edit files in the source directory and re-run `fusesoc` to immediately see the changes,
- run the unit tests as outlined in the section below, and
- use linter and automated code formatters.

3.1.3 Formatting and linting code

The FuseSoC code comes with tooling to automatically format code to conform to our expectations. These tools are installed and called through a tool called pre-commit. No setup is required: whenever you do a `git commit`, the necessary tools are called and your code is automatically formatted and checked for common mistakes.

To check the whole source code `pre-commit` can be run directly:

```
# check and fix all files
pre-commit run -a
```

3.1.4 Running tests

The FuseSoC contains unit tests written using the pytest framework. To run the tests in an isolated environment it is recommended to run pytest through tox, which first creates a package of the source code, installs it, and then runs the tests. This ensures that packaging and environment errors are less likely to slip through.

```
cd fusesoc/source/directory

# Run all tests in an isolated environment (recommended)
tox

# All arguments passed to tox after -- are passed to pytest directly.
# E.g. run a single test: use filename::method_name, e.g.
tox -- tests/test_capi2.py::test_capi2_get_tool --verbose

# Alternatively, tests can be run directly from the source tree.
# E.g. to run a single test: use filename::method_name, e.g.
python3 -m pytest
```

Refer to the pytest documentation for more information how tests can be run.

Note

In many installations you can replace `python3 -m pytest` with the shorter `pytest` command.

3.1.5 Building the documentation

The FuseSoC documentation (i.e., the thing you're reading right now) is built from files in the `doc` directory in the FuseSoC source repository. The documentation is written reStructuredText, and Sphinx is used to convert the documentation into different output formats, such as HTML or PDF.

The most convenient way of working on documentation is to have a browser window open with the rendered documentation next an editor where you work on the reStructuredText files. Run the following command to build the documentation:

```
cd fusesoc/source/directory
tox -e doc-autobuild
```

The documentation is now built and can be accessed in a browser. Look for a line similar to `[sphinx-autobuild] Serving on http://127.0.0.1:8000` and point your browser to the link, e.g. `http://127.0.0.1:8000`.

Whenever a change to a documentation file is detected the documentation will be rebuilt automatically and the refreshed in the browser without the need for further manual action (it might take a couple seconds, though).

INDEX