# Fab

***Release 1.1.dev0***

**Fab Team**

**Sep 04, 2025**

# CONTENTS

Version 1.1 (release 1.1.dev0).

CHAPTER

ONE

# WHAT IS FAB?

An open source build system for Fortran and C projects. Initially designed to build the Met Office's major scientific applications, LFRic and the UM.

CHAPTER

TWO

# WHY A NEW BUILD TOOL?

There are several existing build tools, such as Make, and many more build system builders such as CMake out there. So why muddy these already turgid waters?

A number of popular solutions to the problem of building software were examined and it became apparent that there was a fundamental mismatch in requirements.

Most build systems are designed from the position that the executable is the product. The source files needed to create that executable are listed because they change infrequently and are merely a means to the end of creating an executable.

Scientific software can be rather different. In this paradigm the source is the product and the executable merely a handy means to access it's power. When building is viewed from this point-of-view maintaining lists of source is restricting because it changes so often.

In this light it makes sense to develop a new build tool which gives primacy to the source rather than the executable. The ultimate goal is to support science developers in their work.

For more, please see the *features* page.

CHAPTER

THREE

# RUNNING FAB

- how to *set up an environment*
- how to *install fab*

## 3.1 Zero config

To run fab with *zero configuration*, type `fab` at the command line, within your project.

```
$ cd /path/to/your/source
$ fab
```

The executable file can be found in the *Fab Workspace*.

## 3.2 Config files

- intro to *config files*
- guide to *writing config files*
- *advanced config*

CHAPTER

# FOUR

# SEE ALSO

- Project wiki
- Fab on PyPI
- Fab on Github
- *Developers guide*

## 4.1 Installing Fab

Once you've *setup an environment*, you can install the latest release of Fab from PyPI:

```
$ pip install sci-fab
```

The minimum Python dependencies (e.g. fparser) will also be installed automatically.

**Note**

When installing from PyPI, please be sure to install **sci-fab**, not *fab*. There is already a package on PyPI called *fab*, which installs something else entirely.

### 4.1.1 Extra features

You can install some extra Python packages to enable more features. This will install.

- matplotlib for producing metrics graphs after a run
- psyclone for building LFRic, and more

```
$ pip install sci-fab[features]
```

### 4.1.2 Configuration

#### Fab workspace

You can optionally tell Fab where it's workspace should live. This can be useful on systems where your home space is on a slower drive, or when you like your build to be next to your source:

```
$ export FAB_WORKSPACE=<fast_drive>/fab_workspace
```

By default, Fab will create a project workspaces inside `~/fab-workspace`.

### 4.1.3 Development

People looking to develop Fab will likely want to *Install from source*

### 4.1.4 Met Office Users

There are instructions specific to people wanting to *use Fab at the Met Office*.

## 4.2 Environment

Fab requires a suitable Python environment in which to run. This page outlines some routes to achieving such an environment.

This page contains general instructions, there are additional instructions for *Met Office* users elsewhere.

### 4.2.1 Requirements

The minimum dependencies for Fab to build a project are:

- Python >= 3.7
- fParser >= 0.16.0
- Fortran compiler
- Linker

If you intend to compile C you will need a C compiler and linker but also:

- libclang
- python-clang

If you want pretty charts to be plotted from the build metrics you will need:

- matplotlib

### 4.2.2 Obtaining Fab

Although you can install the tool from a copy of the source it is usually preferable to use one of the managed packages.

Packages are made available through PyPI and Conda forge and are installed in the usual way for these services. Some examples follow.

#### Somebody Elses Problem

Some of the prerequisites may require system administrator intervention. If they are feeling benevolent your administrator may even be prevailed upon to install the whole lot.

#### Using a Python Virtual Environment

A virtual environment is created in the normal way and *pip* used to install Fab:

```
$ python -m venv $ENV_PATH
$ . $ENV_PATH/bin/activate
$ pip install sci-fab
```

Where *$ENV_PATH* is the location for the new environment.

If you want to compile C you will need to add the prerequisites for that:

```
$ pip install python-clang
```

Note that this requires a suitable *libclang* to be installed on your system which may require system administrator intervention.

Finally, to get plots from the metrics you will need:

```
$ pip install matplotlib
```

#### Using Anaconda

Anaconda can be used in a similar way to the built in Python virtual environment but can also handle your C needs:

```
$ conda create -n FabEnv sci-fab
```

Or you can add Fab to an existing environment:

```
$ conda install -n ExistingEnv sci-fab
```

If you want to compile C then you will need to add the prerequisites for that:

```
$ conda install -n FabEnv python-clang
```

And again, if you want to plot the build metrics:

```
$ conda install -n FabEnv matplotlib
```

#### Using Containers

The dockerfile in envs/docker can be used to create a suitable container for running Fab:

```
$ docker build -t fab $FAB_WC/envs/docker
$ docker run --env PYTHONPATH=/fab -v $FAB_WC/source:/fab -v /home/$USER:/home/$USER -it␣
↪fab bash
```

This was tested on Windows, running Ubuntu in WSL but is not regularly tested so can not be guaranteed.

## 4.3 Using Fab at the Met Office

For users working on Met Office desktop systems the following instructions may prove useful.

### 4.3.1 Using Fab

If you just want to use Fab to build your source then there are a number of fairly simple approaches.

#### Fortran Only

If you don't need to build C, you can use the modules in the "extra software" collection:

```
$ module use $EXTRA_SOFTWARE_MODULE_DIR
$ module load python/3.7.0 support/fparser/0.0.16/python/3.7.0 fab/1.0.0/python/3.7.0
```

The Science I.T. group can provide the *EXTRA_SOFTWARE* path.

With the "fab" module loaded the *fab* command will be available and Fab based build scripts will function.

#### Fortran and C

If you need to build C, or you use conda environments anyway you can use the *Conda* route to Fab-ulousness.

You will need to authenticate with Artifactory the first time you use Conda. There is information regarding this on MetNet or ask Science I.T.

#### Singularity Container

You can run Fab in a singularity container as follows:

```
singularity run oras://$URL/MetOffice/fab/MyImage:latest
```

The necessary URL is that normally used with singularity. Ask Science I.T. for details.

If you need to use git from within the container, you'll need to set a couple of environment variables first:

```
$ export SINGULARITY_BIND="/etc/pki/ca-trust/extracted/pem:/pem"
$ export SINGULARITYENV_GIT_SSL_CAPATH="/pem"
$ singularity shell oras://$URL/fab/MyImage:latest
```

See also *Instructions for building the image*.

### 4.3.2 Developing Fab

If you want to develop Fab then there are some additional steps which may be needed.

#### Mixing Virtual Environment and Environment Modules

If you prefer to use the built-in Python virtual environment tool for development you can combine it with the "extra software" modules. The complication comes because they already make use of such an environment.

First load the environment you want to use to compile your software. The LFRic environment is used as an example.

```
$ module use $EXTRA_SOFTWARE
$ module use $LFRIC_SOFTWARE
$ module load environment/lfric/ifort
```

Then create a new virtual environment, unload the existing one and activate the new.

```
$ python3 -m venv $VENV_DIR/fab_dev
$ module unload python/3.7.0
$ source $VENV_DIR/fab_dev/bin/activate
```

Install all the bits needed to install PSyclone and other parts of the build process.

```
$ pip install --upgrade pip
$ pip install Jinja2
$ pip install six
$ pip install fparser
$ pip install pyparsing
$ pip install sympy
$ pip install jsonschema==3.0.2
$ pip install configparser
```

Finally, install Fab in your working copy. This is done in "editable" mode so that changes you make are immediately available through the environment.

```
$ pip install --editable $FAB_WORKING_COPY
```

#### Mixing Conda and Environment Modules

In order to have both an environment capable of building C files and modern Fortran compilers and the LFRic library stack you will need an awkward amalgamation of conda environment and environment modules.

The conda environment is created as follows:

```
$ conda env create -f envs/conda/dev_env.yml
$ conda activate sci-fab
```

Then *install fab*. This is done before any module commands.

The environment is set up *in a new terminal* as follows:

For use with gfortran:

```
$ module use $LFRIC_MODULES
$ module load environment/lfric/gcc
$ conda activate sci-fab
$ PYTHONPATH=~/.conda/envs/sci-fab/lib/python3.7/site-packages:$PYTHONPATH
```

For use with ifort:

```
$ module use $LFRIC_MODULES
$ module load environment/lfric/ifort
$ conda activate sci-fab
$ PYTHONPATH=~/.conda/envs/sci-fab/lib/python3.7/site-packages:$PYTHONPATH
```

#### PyCharm

Running `pycharm-community` from the command line, after activating an environment using any of the above approaches, PyCharm will be able to run Fab, the tests, etc. You can set the project interpreter to be that in the conda environment.

#### Rose

Various configs for building projects using Rose on SPICE can be found in run_configs.

## 4.4 Developing Fab at the Met Office

A few special notes for Met Office developers.

### 4.4.1 Acceptance tests

For extra confidence, we have acceptance tests in the `run_configs` folder which are not run as part of our automated github testing. You can run them on the VDI using `build_all.py`. However, this will choke your machine for some time. There's a (misnamed) cron you can run nightly, `run_configs/_cron/cron_system_tests.sh`.

There's also a rose suite which runs them on spice in `run_configs/_rose_all`.

### 4.4.2 Build singularity image

The config file in envs/picasso defines the contents of a Singularity image which is built by the experimental Picasso app. We can build this image using a GitHub action, defined in `.github/workflows/picasso_build.yml`.

This action is manually triggered. You have to push a branch to the MetOffice repo, not a fork, then you can trigger the action from your branch. Please remember to clean up the branch when you're finished.

You can see the image in artefactory here.

See also * *Run Singularity* * Picasso

## 4.5 Introduction to Configuration

Use Fab to build your Fortran and C project using a series of *build steps* written in Python.

Here is an example of a build configuration. It uses some ready made configurable steps provided by Fab, and it's easy to create your own custom steps.

```
from fab.build_config import BuildConfig
from fab.steps.analyse import analyse
from fab.steps.compile_fortran import compile_fortran
from fab.steps.find_source_files import find_source_files
from fab.steps.grab.folder import grab_folder
from fab.steps.link import link_exe
from fab.steps.preprocess import preprocess_fortran

with BuildConfig(project_label='<project label>') as state:
    grab_folder(state, src='<path to source folder>')
    find_source_files(state)
    preprocess_fortran(state)
    analyse(state, find_programs=True)
    compile_fortran(state)
    link_exe(state)
```

**Note**

The *find_programs* tells the analysis step to discover all Fortran "program" program units and C "main" functions.

If you prefer to specify which programs are to be built you may specify them using the `root_symbol` argument instead. It takes a list of program names.

Fab is designed to minimise user input by providing sensible defaults. Thus it knows to use the *build tree* created by the preceding step as input for the subsequent step.

Build steps can read and create named collections in the *Artefact Store*. For example, in the snippet above we don't tell the compiler which files to compile, that is generated by previous steps.

More details about steps can be found in the *guide to writing configuration*.

## 4.6 How to Write a Build Configuration

This page walks through the process of writing a build script.

### 4.6.1 Configuration File

Here's a simple configuration without any steps.

Listing 1: build_it

```
1  #!/usr/bin/env python3
2  from logging import getLogger
3
4  from fab.build_config import BuildConfig
5
6  logger = getLogger('fab')
7
8  if __name__ == '__main__':
9
10     with BuildConfig(project_label='<project label>') as state:
11         pass
```

If we want to run the build script from the command line, we give it executable permission with the command `chmod +x build_it`. We also add the shebang directive on line 1, telling our computer it's a Python script.

Pick a project label. Fab creates a *project workspace* with this name.

### 4.6.2 Source Code

Let's tell Fab where our source code is.

We use the *find_source_files()* step for this. We can point this step to a source folder, however, because Fab can sometimes create artefacts alongside the source[1], we usually copy the source into the project workspace first using a *grab* step.

A grab step will copy files from a folder or remote repo into a folder called "source" within the project workspace.

Listing 2: build_it.py

```
1  #!/usr/bin/env python3
2  from logging import getLogger
3
4  from fab.build_config import BuildConfig
5  from fab.steps.find_source_files import find_source_files
6  from fab.steps.grab.folder import grab_folder
7
8  logger = getLogger('fab')
9
10 if __name__ == '__main__':
11
12     with BuildConfig(project_label='<project label>') as state:
13         grab_folder(state, src='<path to source folder>')
14         find_source_files(state)
```

**Note**

Fab tries to minimise user input by providing *sensible defaults*. In this case, the user doesn't have to specify where the code goes. The grab and find_source_files steps already know what to do by default. Sensible defaults can be

[1] See *c_pragma_injector()* for an example of a step which creates artefacts in the source folder.

overridden.

Please see the documentation for *find_source_files()* for more information, including how to exclude certain source code from the build. More grab steps can be found in the *grab* module.

After the find_source_files step, there will be a collection called "INITIAL_SOURCE", in the artefact store.

### 4.6.3 Preprocess

Next we want to preprocess our source code. Preprocessing resolves any *#include* and *#ifdef* directives in the code, which must happen before we analyse it.

Steps generally create and find artefacts in the *Artefact Store*, arranged into named collections. The *preprocess_fortran()* automatically looks for Fortran source code in a collection named *'INITIAL_SOURCE'*, which is the default output from the preceding :funcfind_source_files step. It filters just the (uppercase) .F90 files.

**Note**

Uppercase .F90 are preprocessed into lowercase .f90.

Listing 3: build_it.py

```
#!/usr/bin/env python3
from logging import getLogger

from fab.build_config import BuildConfig
from fab.steps.find_source_files import find_source_files
from fab.steps.grab.folder import grab_folder
from fab.steps.preprocess import preprocess_fortran

logger = getLogger('fab')

if __name__ == '__main__':

    with BuildConfig(project_label='<project label>') as state:
        grab_folder(state, src='<path to source folder>')
        find_source_files(state)
        preprocess_fortran(state)
```

Preprocessed files are created in the *'build_output'* folder, inside the project workspace. After the fortran_preprocessor step, there will be a collection called "preprocessed_fortran", in the artefact store.

### 4.6.4 PSyclone

If you want to use PSyclone to do code transformation and pre-processing (see https://github.com/stfc/PSyclone), you must run preprocess_x90() and *psyclone()*, before you run the *analyse()* step below.

- **For preprocess_x90():**
  You can pass in *common_flags* list as an argument.
- **For *psyclone()*:**
  You can pass in:
  - kernel file roots to *kernel_roots*,

- a function to get transformation script to *transformation_script* (see examples in `~fab.run_configs.lfric.gungho.py` and `~fab.run_configs.lfric.atm.py`),
- command-line arguments to *cli_args*,
- override for input files to *source_getter*,
- folders containing override files to *overrides_folder*.

Listing 4: build_it.py

```
#!/usr/bin/env python3
from logging import getLogger

from fab.build_config import BuildConfig
from fab.steps.find_source_files import find_source_files
from fab.steps.grab.folder import grab_folder
from fab.steps.preprocess import preprocess_fortran
from fab.steps.psyclone import psyclone, preprocess_x90

logger = getLogger('fab')

if __name__ == '__main__':

    with BuildConfig(project_label='<project label>') as state:
        grab_folder(state, src='<path to source folder>')
        find_source_files(state)
        preprocess_fortran(state)
        preprocess_x90(state)
        psyclone(state)
```

After the psyclone step, two new source files will be created for each .x90 file in the *'build_output'* folder. These two output files will be added under `FORTRAN_BUILD_FILES` collection to the artefact store.

### 4.6.5 Analyse

We must *`analyse()`* the source code to determine which Fortran files to compile, and in which order.

The Analyse step looks for source to analyse in two collections:

- `FORTRAN_BUILD_FILES`, which contains all `.f90` found in the source, all `.F90` files we pre-processed into `.f90`, and files created by any additional step (e.g. PSyclone).
- `C_BUILD_FILES`, all preprocessed c files.

Listing 5: build_it.py

```
#!/usr/bin/env python3
from logging import getLogger

from fab.steps.analyse import analyse
from fab.build_config import BuildConfig
from fab.steps.find_source_files import find_source_files
from fab.steps.grab.folder import grab_folder
from fab.steps.preprocess import preprocess_fortran
from fab.steps.psyclone import psyclone, preprocess_x90

```

(continues on next page)

(continued from previous page)

```
logger = getLogger('fab')

if __name__ == '__main__':

    with BuildConfig(project_label='<project label>') as state:
        grab_folder(state, src='<path to source folder>')
        find_source_files(state)
        preprocess_fortran(state)
        preprocess_x90(state)
        psyclone(state)
        analyse(state, root_symbol='<program>')
```

Here we tell the analyser which *Root Symbol* we want to build into an executable. Alternatively, we can use the `find_programs` flag for Fab to discover and build all programs.

After the Analyse step, there will be a collection called `BUILD_TREES`, in the artefact store.

### 4.6.6 Compile and Link

The *compile_fortran()* step compiles files in the `BUILD_TREES` collection. The *link_exe()* step then creates the executable.

Listing 6: build_it.py

```
#!/usr/bin/env python3
from logging import getLogger

from fab.steps.analyse import analyse
from fab.build_config import BuildConfig
from fab.steps.compile_fortran import compile_fortran
from fab.steps.find_source_files import find_source_files
from fab.steps.grab.folder import grab_folder
from fab.steps.link import link_exe
from fab.steps.preprocess import preprocess_fortran
from fab.steps.psyclone import psyclone, preprocess_x90

logger = getLogger('fab')

if __name__ == '__main__':

    with BuildConfig(project_label='<project label>') as state:
        grab_folder(state, src='<path to source folder>')
        find_source_files(state)
        preprocess_fortran(state)
        preprocess_x90(state)
        psyclone(state)
        analyse(state, root_symbol='<program>')
        compile_fortran(state)
        link_exe(state)
```

After the *link_exe()* step, the executable name can be found in a collection called `EXECUTABLES`.

### 4.6.7 ArtefactStore

Each build configuration contains an artefact store, containing various sets of artefacts. The artefact sets used by Fab are defined in the enum *ArtefactSet*. The most important sets are FORTRAN_BUILD_FILES, C_BUILD_FILES, which will always contain all known source files that will need to be analysed for dependencies, compiled, and linked. All existing steps in Fab will make sure to maintain these artefact sets consistently, for example, if a .F90 file is preprocessed, the .F90 file in FORTRAN_BUILD_FILES will be replaced with the corresponding preprocessed .f90 file. Similarly, new files (for examples created by PSyclone) will be added to FORTRAN_BUILD_FILES. A user script can adds its own artefacts using strings as keys if required.

The exact flow of artefact sets is as follows. Note that any artefact sets mentioned here can typically be overwritten by the user, but then it is the user's responsibility to maintain the default artefact sets (or change them all):

1. *find_source_files()* will add all source files it finds to INITIAL_SOURCE (by default, can be overwritten by the user). Any .F90 and .f90 file will also be added to FORTRAN_BUILD_FILES, any .c file to C_BUILD_FILES, and any .x90 or .X90 file to X90_BUILD_FILES. It can be called several times if files from different root directories need to be added, and it will automatically update the *_BUILD_FILES sets.
2. Any user script that creates new files can add files to INITIAL_SOURCE if required, but also to the corresponding *_BUILD_FILES. This will happen automatically if *find_source_files()* is called to add these newly created files.
3. If *c_pragma_injector()* is being called, it will handle all files in C_BUILD_FILES, and will replace all the original C files with the newly created ones. For backward compatibility it will also store the new objects in the PRAGMAD_C set.
4. If *preprocess_c()* is called, it will preprocess all files in C_BUILD_FILES (at this stage typically preprocess the files in the original source folder, writing the output files to the build folder), and update that artefact set accordingly. For backward compatibility it will also store the preprocessed files in PREPROCESSED_C.
5. If *preprocess_fortran()* is called, it will preprocess all files in FORTRAN_BUILD_FILES that end on .F90, creating new .f90 files in the build folder. These files will be added to PREPROCESSED_FORTRAN. Then the original .F90 are removed from FORTRAN_BUILD_FILES, and the new preprocessed files (which are in PREPROCESSED_FORTRAN) will be added. Then any .f90 files that are not already in the build folder (an example of this are files created by a user script) are copied from the original source folder into the build folder, and FORTRAN_BUILD_FILES is updated to use the files in the new location.
6. If preprocess_x90() is called, it will similarly preprocess all .X90 files in X90_BUILD_FILES, creating the output files in the build folder, and replacing the files in X90_BUILD_FILES.
7. If *psyclone()* is called, it will process all files in X90_BUILD_FILES and add any newly created file to FORTRAN_BUILD_FILES, and removing them from X90_BUILD_FILES.
8. The *analyse()* step analyses all files in FORTRAN_BUILD_FILES and C_BUILD_FILES, and add all dependencies to BUILD_TREES.
9. The *compile_c()* and *compile_fortran()* steps will compile all files from C_BUILD_FILES and FORTRAN_BUILD_FILES, and add them to OBJECT_FILES.
10. If *archive_objects()* is called, it will create libraries based on OBJECT_FILES, adding the libraries to OBJECT_ARCHIVES.
11. If *link_exe()* is called, it will either use OBJECT_ARCHIVES, or if this is empty, use OBJECT_FILES, create the binaries, and add them to EXECUTABLES.

### 4.6.8 Flags

Preprocess, compile and link steps usually need configuration to specify command-line arguments to the underlying tool, such as symbol definitions, include paths, optimisation flags, etc. See also *Advanced Flags*.

### 4.6.9 C Code

Fab comes with C processing steps. The *preprocess_c()* and *compile_c()* Steps behave like their Fortran equivalents.

However preprocessing C currently requires a preceding step called the *c_pragma_injector()*. This injects markers into the C code so Fab is able to deduce which inclusions are user code and which are system code. This allows system dependencies to be ignored.

See also *Advanced C Code*

### 4.6.10 Further Reading

More advanced configuration topics are discussed in *Advanced Configuration*.

You can see more complicated configurations in the developer testing directory.

## 4.7 Advanced Configuration

A lot can be achieved with simple configurations but some of the more esoteric aspects of software building may require more esoteric Fab features.

### 4.7.1 Understanding the Environment

Fab itself does not support any environment variables. But a user script can obviously query the environment and make use of environment variables, and provide their values to Fab.

### 4.7.2 Configuration Reuse

If you find you have multiple build configurations with duplicated code, it could be helpful to factor out the commonality into a shared module. Remember, your build configuration is just a Python script at the end of the day.

In Fab's example configurations, we have two build scripts to compile GCOM. Much of the configuration for these two scripts is identical. We extracted the common steps into gcom_build_steps.py and used them in build_gcom_ar.py and build_gcom_so.py.

### 4.7.3 Separate grab and build scripts

If you are running many builds from the same source, you may wish to grab your repo in a separate script and call it less frequently.

In this case your grab script might only contain a single step. You could import your grab configuration to find out where it put the source.

Listing 7: my_grab.py

```
my_grab_config = BuildConfig(project_label='<project_label>')

if __name__ == '__main__':
    with my_grab_config:
        fcm_export(my_grab_config, src='my_repo')
```

Listing 8: my_build.py

```
from my_grab import my_grab_config

```

(continues on next page)

(continued from previous page)

```
3
4  if __name__ == '__main__':
5      with BuildConfig(project_label='<project_label>') as state:
6          grab_folder(state, src=my_grab_config.source_root),
```

### 4.7.4 Housekeeping

You can add a *cleanup_prebuilds()* step, where you can explicitly control how long to keep prebuild files. This may be useful, for example, if you often switch between two versions of your code and want to keep the prebuild speed benefits when building both.

If you do not add your own cleanup_prebuild step, Fab will automatically run a default step which will remove old files from the prebuilds folder. It will remove all prebuild files that are not part of the current build by default.

### 4.7.5 Sharing Prebuilds

You can copy the contents of someone else's prebuilds folder into your own.

Fab uses hashes to keep track of the correct prebuilt files, and will find and use them. There's also a helper step called *grab_pre_build()* you can add to your build configurations.

### 4.7.6 PSyKAlight (PSyclone overrides)

If you need to override a PSyclone output file with a handcrafted version you can use the `overrides_folder` argument to the *psyclone()* step.

This specifies a normal folder containing source files. The step will delete any files it creates if there's a matching filename in the overrides folder.

### 4.7.7 Two-Stage Compilation

**The *compile_fortran()* step compiles files in**
passes, with each pass identifying all the files which can be compiled next, and compiling them in parallel.

Some projects have bottlenecks in their compile order, where lots of files are stuck behind a single file which is slow to compile. Inspired by Busby, Fab can perform two-stage compilation where all the modules are built first in *fast passes* using the *-fsyntax-only* flag, and then all the slower object compilation can follow in a single pass.

The *potential* benefit is that the bottleneck is shortened, but there is a tradeoff with having to run through all the files twice. Some compilers might not have this capability.

Two-stage compilation is configured with the *two_stage_flag* argument to the Fortran compiler.

```
1  compile_fortran(state, two_stage_flag=True)
```

### 4.7.8 Managed arguments

As noted above, Fab manages a few command line arguments for some of the tools it uses.

#### Fortran Preprocessors

Fab knows about some preprocessors which are used with Fortran, currently *fpp* and *cpp*. It will ensure the `-P` flag is present to disable line numbering directives in the output, which is currently required for fparser to parse the output.

#### Fortran Compilers

Fab knows about some Fortran compilers (currently *gfortran* or *ifort*). It will make sure the *-c* flag is present to compile but not link.

If the compiler flag which sets the module folder is present, i.e. `-J` for gfortran or `-module` for ifort, Fab will **remove** the flag, with a notification, as it needs to use this flag to control the output location.

### 4.7.9 Compilation Profiles

Fab supports compilation profiles. A compilation profile is essentially a simple string that represents a set of compilation and linking flags to be used. For example, an application might have profiles for *full-debug*, *fast-debug*, and *production*. Compilation profiles can inherit settings, for example *fast-debug* might inherit from *full-debug*, but add optimisations. Compilation profile names are not case sensitive.

Any flag for any tool can make use of a profile, but in many cases this is not necessary (think of options for `rsync`, `git`, `svn`, ...). Fab will internally create a dummy profile, indicated by an empty string "". If no profile is specified, this default profile will be used.

A profile is defined as follows:

```
1  tr = ToolRepository()
2  gfortran = tr.get_tool(Category.FORTRAN_COMPILER, "gfortran")
3
4  gfortran.define_profile("base")
5  gfortran.define_profile("fast-debug", inherit_from="base")
6  gfortran.define_profile("full-debug", inherit_from="fast-debug")
7
8  gfortran.add_flags(["-g", "-std=f2008"], "base")
9  gfortran.add_flags(["-O2"], "fast-debug")
10 gfortran.add_flags(["-O0", "-fcheck=all"], "full-debug")
```

Line 3 defines a profile called `base`, which does not inherit from any other profile. Next, a profile `fast-debug` is defined, which is based on `base`. It will add the flags `-O2` to the command line, together with the inherited flags from base, it will be using `-g -std=f2008 -O2` Finally, a `full-debug` profile is declared, based on `fast-debug`. Due to the inheritance, it will be using the options `-g -std=f2008 -O2 -O0 -fcheck=all`. Note that because of the precedence of compiler flags, the no-optimisation flag `-O0` will overwrite the valued of `-O2`.

Tools that do not require a profile can omit the parameter when defining flags:

```
1  git = config.tool_box[Category.GIT]
2  git.add_flags(["-c", "foo.bar=123"])
```

This effectively adds the flags to the to the dummy profile, allowing them to be used by other Fab functions.

By default, the dummy profile "" is not used as a base class for any other profile. But it can be convenient to set this up to make user scripts slightly easier. Here is an example of the usage in LFRic, where at startup time a consistent set of profile modes are defined for each compiler and linker:

```
tr = ToolRepository()
for compiler in (tr[Category.C_COMPILER] +
                 tr[Category.FORTRAN_COMPILER] +
                 tr[Category.LINKER]):
    compiler.define_profile("base", inherit_from="")
    for profile in ["full-debug", "fast-debug", "production"]:
        compiler.define_profile(profile, inherit_from="base")
```

Line 5 defines a `base` profile, which inherits from the dummy profile. Then a set of three profiles are defined, each inheriting from `base`, and therefore in turn from the dummy profile.

Later, the Intel Fortran compiler and linker `ifort` are setup as follows:

```
1  tr = ToolRepository()
2  ifort = tr.get_tool(Category.FORTRAN_COMPILER, "ifort")
3  ifort.add_flags(["-stand", "f08"],           "base")
4  ifort.add_flags(["-g", "-traceback"],        "base")
5  ifort.add_flags(["-O0", "-ftrapuv"],         "full-debug")
6  ifort.add_flags(["-O2", "-fp-model=strict"], "fast-debug")
7  ifort.add_flags(["-O3", "-xhost"],           "production")
8
9  linker = tr.get_tool(Category.LINKER, "linker-ifort")
10 linker.add_lib_flags("yaxt", ["-lyaxt", "-lyaxt_c"])
11 linker.add_post_lib_flags(["-lstdc++"])
```

The setup of the compiler does not use the dummy profile at all, so it will stay empty. It is up to the user to decide how to use the profiles, it would be entirely valid not to use the `base` profile, but instead to use the dummy. But when setting up the linker, no profile is specified. So line 10 and 11 will set these flags for the dummy. Because of `base` inheriting from the dummy, and any other profile inheriting from `base`, this means these linker flags will be used for all profiles. It would be equally valid to define these flags for the `base` profile:

```
linker = tr.get_tool(Category.LINKER, "linker-ifort")
linker.add_lib_flags("yaxt", ["-lyaxt", "-lyaxt_c"], "base")
linker.add_post_lib_flags(["-lstdc++"], "base")
```

This design was chosen because the most common use case for profiles involves changing compiler flags. Linker flags are typically left unaltered, so it is more intuitive for a user to omit profile modes for the linker.

The advantage of supporting the profile modes for linker is that you can specify profile modes that require additional linking options. One example is GNU's address sanitizer, which requires to add the compilation option `-fsanitize=address`, and the linker option `-static-libasan`.

```
tr = ToolRepository()
gfortran = tr.get_tool(Category.FORTRAN_COMPILER, "gfortran")
...
gfortran.define_profile("memory-debug", "full-debug")
gfortran.add_flags(["-fsanitize=address"], "memory-debug")
linker = tr.get_tool(Category.LINKER, "linker-gfortran")
linker.add_post_lib_flags(["-static-libasan"], "memory-debug")
```

This way, by just changing the profile, compilation and linking will be affected consistently.

### 4.7.10 Tool arguments

Sometimes it is necessary to pass additional arguments when we call a software tool.

#### Linker flags

Probably the most common instance of the need to pass additional arguments is to specify 3rd party libraries at the link stage. The linker tool allow for the definition of library-specific flags: for each library, the user can specify the required linker flags for this library. In the linking step, only the name of the libraries to be linked is then required. The linker object will then use the required linking flags. Typically, a site-specific setup set (see for example https://github.com/MetOffice/lfric-baf) will specify the right flags for each site, and the application itself only needs to list the name of the libraries. This way, the application-specific Fab script is independent from any site-specific settings.

Still, an application-specific script can also overwrite any site-specific setting, for example if a newer version of a dependency is to be evaluated.

The settings for a library are defined as follows:

```
    tr = ToolRepository()
    linker = tr.get_tool(Category.LINKER, "linker-ifort")

    linker.add_lib_flags("yaxt", ["-L/some_path", "-lyaxt", "-lyaxt_c"])
    linker.add_lib_flags("xios", ["-lxios"])
```

This will define two libraries called `yaxt` and `xios`. In the link step, the application only needs to specify the name of the libraries required, e.g.:

```
link_exe(state, libs=["yaxt", "xios"])
```

The linker will then use the specified options.

A linker object also allows to define options that should always be added, either as options before any library details, or at the very end. For example:

```
    linker.add_pre_lib_flags(["-L/my/common/library/path"])
    linker.add_post_lib_flags(["-lstdc++"])
```

The pre_lib_flags can be used to specify library paths that contain several libraries only once, and this makes it easy to evaluate a different set of libraries. Additionally, this can also be used to add common linking options, e.g. Cray's `-Ktrap=fp`.

The post_lib_flags can be used for additional common libraries that need to be linked in. For example, if the application contains a dependency to C++ but it is using the Fortran compiler for linking, then the C++ libraries need to be explicitly added. But if there are several libraries depending on it, you would have to specify this several times (forcing the linker to re-read the library several times). Instead, you can just add it to the post flags once.

The linker step itself can also take optional flags:

```
link_exe(state, flags=['-Ktrap=fp'])
```

These flags will be added to the very end of the the linker options, i.e. after any other library or post-lib flag. Note that the example above is not actually recommended to use, since the specified flag is only valid for certain linker, and a Fab application script should in general not hard-code flags for a specific linker. Adding the flag to the linker instance itself, as shown further above, is the better approach.

### Path-specific flags

For preprocessing and compilation, we sometimes need to specify flags *per-file*. These steps accept both common flags and *path specific* flags.

```
...
compile_fortran(
    common_flags=['-O2'],
    path_flags=[
        AddFlags('$output/um/*', ['-I' + '/gcom'])
    ],
)
```

This will add `-O2` to every invocation of the tool, but only add the `*/gcom*` include path when processing files in the `*<project workspace>/build_output/um*` folder.

Path matching is done using Python's fnmatch. The `$output` is a template, see *AddFlags*.

We can currently only *add* flags for a path.

**Note**

This can require some understanding of where and when files are placed in the *build_output* folder: It will generally match the structure you've created in *`<project workspace>/source`*, with your grab steps.

Early steps like preprocessors generally read files from *`source`* and write to *`build_output`*.

Later steps like compilers generally read files which are already in *`build_output`*.

For more information on where files end up see *Folder Structure*.

### 4.7.11 Folder Structure

It may be useful to understand how Fab uses the *Project Workspace* and in particular where it creates files within it.

```
<your $FAB_WORKSPACE>
   <project workspace>
      source/
      build_output/
         *.f90 (preprocessed Fortran files)
         *.mod (compiled module files)
         _prebuild/
            *.an (analysis results)
            *.o (compiled object files)
            *.mod (mod files)
      metrics/
      my_program
      log.txt
```

The *project workspace* folder takes its name from the project label passed in to the build configuration.

The *source* folder is where grab steps place their files.

The *build_output* folder is where steps put their processed files. For example, a preprocessor reads `.F90` from *source* and writes `.f90` to *build_output*.

The *_prebuild* folder contains reusable output. Files in this folder include a hash value in their filenames.

The *metrics* folder contains some useful stats and graphs. See *Metrics*.

### 4.7.12 C Pragma Injector

The C pragma injector creates new C files with `.prag` file extensions, in the source folder. The C preprocessor looks for the output of this step by default. If not found, it will fall back to looking for `.c` files in the source listing.

```
...
c_pragma_injector(state)
preprocess_c(state)
...
```

### 4.7.13 Custom Steps

If you need a custom build step, you can create a function with the @step decorator.

Some example custom steps are included in the Fab testing configurations. For example a simple example was created for building JULES.

The *root_inc_files()* step copies all `.inc` files in the source tree into the root of the source tree, to make subsequent preprocessing flags easier to configure.

That is a simple example that doesn't need to interact with the *Artefact Store*. Sometimes inserting a custom step means inserting a new *Artefact Collection* into the flow of data between steps.

We can tell a subsequent step to read our new artefacts, instead of using it's default *Artefacts Getter*. We do this using the `source` argument, which most Fab steps accept. (See *Collection names*)

```
@step
def custom_step(state):
    state.artefact_store['custom_artefacts'] = do_something(state.artefact_store['step 1␣
↪artefacts'])


with BuildConfig(project_label='<project label>') as state:
    fab_step1(state)
    custom_step(state)
    fab_step2(state, source=CollectionGetter('custom_artefacts'))
```

Steps have access to multiprocessing methods through the *run_mp()* helper function. This processes artefacts in parallel.

```
@step
def custom_step(state):
    input_files = state.artefact_store['custom_artefacts']
    results = run_mp(state, items=input_files, func=do_something)
```

### 4.7.14 Collection names

Most steps allow the collections they read from and write to to be changed.

Let's imagine we need to upgrade a build script, adding a custom step to prepare our Fortran files for preprocessing.

```
find_source_files(state)  # this was already here

# instead of this
# preprocess_fortran(state)

# we now do this
my_new_step(state, output_collection='my_new_collection')
preprocess_fortran(state, source=CollectionGetter('my_new_collection'))

analyse(state)  # this was already here
```

### 4.7.15 Parser Workarounds

Sometimes the parser used by Fab to understand source code can be unable to parse valid source files due to bugs or shortcomings. In order to still be able to build such code a number of possible work-arounds are presented.

#### Unrecognised Dependencies

If a language parser is not able to recognise a dependency within a file, then Fab won't know the dependency needs to be compiled.

For example, some versions of fparser don't recognise a call on a one-line if statement.

We can manually add the dependency using the *unreferenced_deps* argument to *analyse()*.

Pass in the name of the called function. Fab will find the file containing this symbol and add it, *and all its dependencies*, to every *Build Tree*.

```
...
analyse(state, root_symbol='my_prog', unreferenced_deps=['my_func'])
...
```

#### Unparsable Files

If a language parser is not able to process a file at all, then Fab won't know about any of its symbols and dependencies. This can sometimes happen to *valid code* which compilers *are* able to process, for example if the language parser is still maturing and can't yet handle an uncommon syntax.

In this case we can manually give Fab the analysis results using the *special_measure_analysis_results* argument to *analyse()*.

Pass in a list of *FortranParserWorkaround* objects, one for every file that can't be parsed. Each object contains the symbol definitions and dependencies found in one source file.

```
...
analyse(
    config,
    root_symbol='my_prog',
    special_measure_analysis_results=[
        FortranParserWorkaround(
            fpath=Path(state.build_output / "path/to/file.f90"),
            module_defs={'my_mod'}, symbol_defs={'my_func'},
            module_deps={'other_mod'}, symbol_deps={'other_func'}),
    ])
...
```

In the above snippet we tell Fab that `file.f90` defines a module called `my_mod` and a function called `my_func`, and depends on a module called `other_mod` and a function called `other_func`.

#### Custom Step

An alternative approach for some problems is to write a custom step to modify the source so that the language parser can process it. Here's a simple example, based on a real workaround where the parser gets confused by a variable called *NameListFile*.

```
@step
def my_custom_code_fixes(state):
    fpath = state.source_root / 'path/to/file.F90'
```

(continues on next page)

(continued from previous page)

```
    in = open(fpath, "rt").read()
    out = in.replace("NameListFile", "MyRenamedVariable")
    open(fpath, "wt").write(out)

with BuildConfig(project_label='<project_label>') as state:
    # grab steps first
    my_custom_code_fixes(state)
    # find_source_files, preprocess, etc, afterwards
```

A more detailed treatment of *Custom Steps* is given elsewhere.

## 4.8 Site-Specific Configuration

A site might have compilers that Fab doesn't know about, or prefers a different compiler from the Fab default. Fab abstracts the compilers and other programs required during building as an instance of a *Tool* class. All tools that Fab knows about, are available in a *ToolRepository*. That will include tools that might not be available on the current system.

Each tool belongs to a certain category of `Category`. A *ToolRepository* can store several instances of the same category.

At build time, the user has to create an instance of *ToolBox* and pass it to the *BuildConfig* object. This toolbox contains all the tools that will be used during the build process, but it can only store one tool per category. If a certain tool should not be defined in the toolbox, the default from the *ToolRepository* will be used. This is useful for many standard tools like *git*, *rsync* etc that de-facto will never be changed. Fab will check if a tool is actually available on the system before adding it to a ToolBox. This is typically done by trying to run the tool with some testing parameters, for example requesting its version number. If this fails, the tool is considered not to be available and will not be used (unless the user explicitly puts a tool into the ToolBox).

**Note**

If you need to use for example different compilers for different files, you would implement this as a *meta-compiler*: implement a new class based on the existing *Compiler* class, which takes two (or more) compiler instances. Its *compile_file()* method can then decide (e.g. based on the path of the file to compile, or a hard-coded set of criteria) which compiler to use.

### 4.8.1 Category

All possible categories are defined in `Category`. If additional categories should be required, they can be added.

### 4.8.2 Tool

Each tool must be derived from *Tool*. The base class provides a *run* method, which any tool can use to execute a command in a shell. Typically, a tool will provide one (or several) custom commands to be used by the steps. For example, a compiler instance provides a *compile_file()* method. This makes sure that no tool-specific command line options need to be used in any Fab step, which will allow the user to replace any tool with a different one.

New tools can easily be created, look at *Gcc* or *Icc*. Typically, they can just be created by providing a different set of parameters in the constructor.

### 4.8.3 Tool Repository

The *ToolRepository* implements a singleton to access any tool that Fab knows about. A site-specific startup section can add more tools to the repository:

Listing 9: ToolRepository

```
from fab.tools import ToolRepository

# Assume the MpiF90 class as shown in the previous example

tr = ToolRepository()
tr.add_tool(MpiF90)   # the tool repository will create the instance
```

Compiler and linker objects define a compiler suite, and the *ToolRepository* provides *set_default_compiler_suite()* which allows you to change the defaults for compiler and linker with a single call. This will allow you to easily switch from one compiler to another. If required, you can still change any individual compiler after setting a default compiler suite, e.g. you can define *intel-classic* as default suite, but set the C-compiler to be *gcc*.

### 4.8.4 Tool Box

The class *ToolBox* is used to provide the tools to be used by the build environment, i.e. the *BuildConfig* object:

Listing 10: ToolBox

```
from fab.tools import Category, ToolBox, ToolRepository

tr = ToolRepository()
tr.set_default_compiler_suite("intel-classic")
tool_box = ToolBox()
ifort = tr.get_tool(Category.FORTRAN_COMPILER, "ifort")
tool_box.add_tool(ifort)
c_compiler = tr.get_default(Category.C_COMPILER)
tool_box.add_tool(c_compiler)

config = BuildConfig(tool_box=tool_box,
                     project_label=f'lfric_atm-{ifort.name}', ...)
```

The advantage of finding the compilers to use in the tool box is that it allows a site to replace a compiler in the tool repository (e.g. if a site wants to use an older gfortran version, say one which is called *gfortran-11*). They can then remove the standard gfortran in the tool repository and replace it with a new gfortran compiler that will call *gfortran-11* instead of *gfortran*. But a site can also decide to not support a generic *gfortran* call, instead adding different gfortran compiler with a version number in the name.

If a tool category is not defined in the *ToolBox*, then the default tool from the *ToolRepository* will be used. Therefore, in the example above adding *ifort* is not strictly necessary (since it will be the default after setting the default compiler suite to *intel-classic*), and *c_compiler* is the default as well. This feature is especially useful for the many default tools that Fab requires (git, rsync, ar, ...).

Listing 11: ToolBox

```
tool_box = ToolBox()
default_c_compiler = tool_box.get_tool(Category.C_COMPILER)
```

There is special handling for compilers and linkers: the build configuration stores the information if an MPI and/or OpenMP build is requested. So when a default tool is requested by the ToolBox from the ToolRepository (i.e. when the user has not added specific compilers or linkers), this information is taken into account, and only a compiler that will fulfil the requirements is returned. For example, if you have *gfortran* and *mpif90-gfortran* defined in this order in the ToolRepository, and request the default compiler for an MPI build, the *mpif90-gfortran* instance is returned, not *gfortran*. On the other hand, if no MPI is requested, an MPI-enabled compiler might be returned, which does not affect the final result, since an MPI compiler just adds include- and library-paths.

### 4.8.5 Compiler Wrapper

Fab supports the concept of a compiler wrapper, which is typically a script that calls the actual compiler. An example for a wrapper is *mpif90*, which might call a GNU or Intel based compiler (with additional parameter for accessing the MPI specific include and library paths.). An example to create a *mpicc* wrapper (note that this wrapper is already part of Fab, there is no need to explicitly add this yourself):

Listing 12: Defining an mpicc compiler wrapper

```
class Mpicc(CompilerWrapper):
    def __init__(self, compiler: Compiler):
        super().__init__(name=f"mpicc-{compiler.name}",
                         exec_name="mpicc",
                         compiler=compiler, mpi=True)
```

The tool system allows several different tools to use the same name for the executable, as long as the Fab name is different, i.e. the *mpicc-{compiler.name}*. The tool repository will automatically add compiler wrapper for *mpicc* and *mpif90* for any compiler that is added by Fab. If you want to add a new compiler, which can also be invoked using *mpicc*, you need to add a compiler wrapper as follows:

Listing 13: Adding a mpicc wrapper to the tool repository

```
my_new_compiler = MyNewCompiler()
ToolRepository().add_tool(my_new_compiler)
my_new_mpicc = Mpicc(MyNewCompiler)
ToolRepository().add_tool(my_new_mpicc)
```

When creating a completely new compiler and compiler wrapper as in the example above, it is strongly recommended to add the new compiler instance to the tool repository as well. This will allow the wrapper and the wrapped compiler to share flags. For example, a user script can query the ToolRepository to get the original compiler and modify its flags. These modification will then automatically be applied to the wrapper as well:

Listing 14: Sharing flags between compiler and compiler wrapper

```
tr = ToolRepository()
my_compiler = tr.get_tool(Category.C_COMPILER, "my_compiler")
my_mpicc = tr.get_tool(Category.C_COMPILER, "mpicc-my_compiler")

my_compiler.add_flags(["-a", "-b"])

assert my_mpicc.flags == ["-a", "-b"]
```

### 4.8.6 TODO

At this stage path-specific compiler flags are still set in the corresponding Fab steps, and it might make more sense to allow their modification and definition in the compiler objects. This will allow a site to define their own set of default flags to be used with a certain compiler by replacing or updating a compiler instance in the Tool Repository. There is some support for querying path-specific flag from the site-specific config, but TODO #313 will make this much nicer.

Also, a lot of content in this chapter is not actually about site-specific configuration. This should likely be renamed or split when updating and refactoring the documentation.

## 4.9 Features

Fab is an open source Python project. Feel free to get involved.

### 4.9.1 Dependency Analysis

Fab automatically discovers your C and Fortran source files. You don't need to manually specify and maintain an ordered list of files, which can become problematic in a large project.

Fab determines the dependency hierarchy, including through C-Fortran interfacing, and thus determines the Fortran compile order. C is compiled in a single pass.

### 4.9.2 Incremental Build

Fab "watermarks" each artefact with a checksum of its inputs. Subsequent builds avoid reprocessing by searching for watermarks in the prebuild folder.

### 4.9.3 Sharing Prebuilds

Artefacts from previous builds can be shared between users, either by copying the prebuild folder or using the `grab_pre_build()` step.

### 4.9.4 Extensible

It's easy to add custom steps to your build script, e.g manipulating code, calling new tools, etc.

### 4.9.5 Zero Configuration

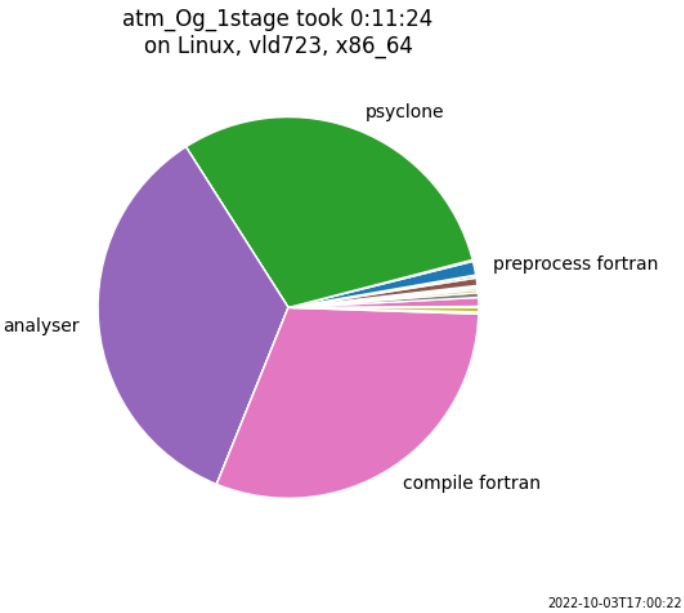
It's possible to run `fab` from the command line, in your source folder, for a default build operation. For more complicated builds you may write a build script.

### 4.9.6 Git, SVN and FCM

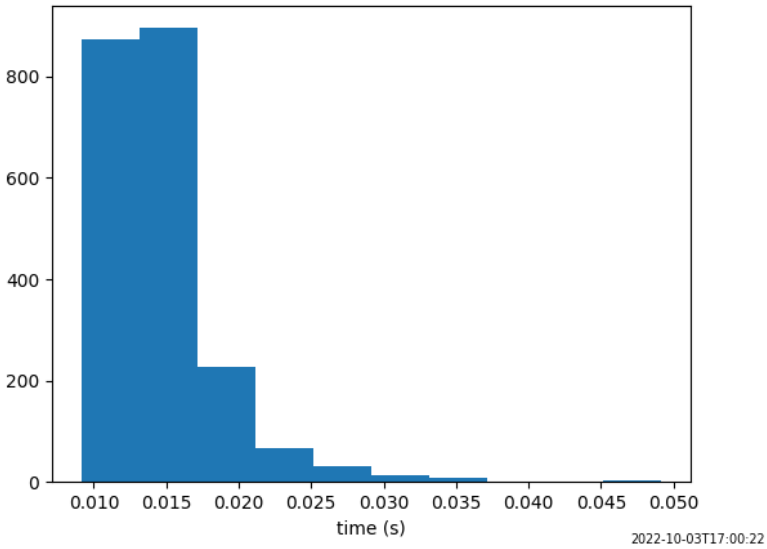
Fab can fetch and merge source from remote repositories.

### 4.9.7 Metrics

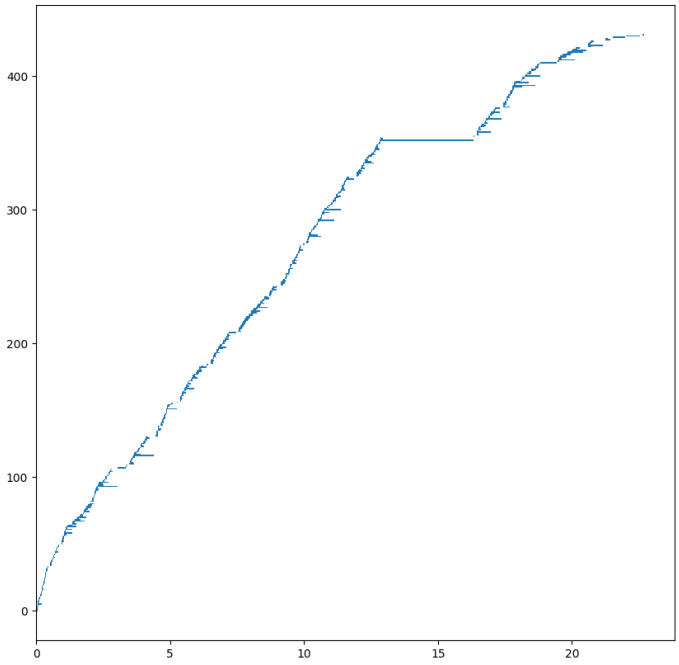
Fab will record the time taken by each step and plot them on a pie chart.




Some steps also record metrics about their individual artefacts. For example below, the Fortran preprocessor created this histogram from the processing time of individual files,




and the Fortran compiler created this busby-style graph showing a compilation bottleneck.




### 4.9.8 Limitations

Known problems at the time of writing. For further issues see the issue tracker.

#### Fortran single-line IF calls

Whilst fab can automatically determine Fortran dependencies from module use statements, and from standalone call statements, it doesn't currently detect a dependency from a call statement on a single-line if statement: `IF (x .GT. 0) CALL foo()`. Please see here for *the workaround*.

#### Name Clash

Fab currently assumes there are no name clashes in your project by the time we reach certain build steps:

- C and Fortran symbols go into one symbol table so there can be no duplicate symbol names by the time we reach the analysis stage.
- Fortran mod files are created in a flat folder, so Fortran module names must be unique by the time we reach the compile stage.
- C and Fortran object files are both compiled into *.o* files so there can be no duplicates, such as *util.c* and *util.f90*, by the time we reach the compile stage.

There may be duplicates earlier in the build process. For example, there may two versions of a module, each wrapped in a *#ifdef* so that one of them is empty after preprocessing (empty files are ignored during analysis).

Another approach is to use file filtering in the `find_source_files` step.

#### Fortran Include Folders and Incremental Build

Fab generates a hash of Fortran **.mod* file dependencies, and notices if a dependency changes, triggering a recompile. However, it can only currently do this for Fortran **.mod* files inside the project workspace (or source_root override). It will *not* notice if a Fortran **.mod* changes in an include folder elsewhere.

An example is the UM build which uses GCom's mpl.mod. This issue is raised in #192.

## 4.10 Introduction to the Fab Base Class

Fab provides a base class, which provides a command line interface to the Fab build system. It is written in Python, and provides a pre-defined framework for building binary files and libraries. It adds the following features:

- It is object-oriented, making it easy to re-use and extend build scripts.
- It is driven by a command line interface. All options required for a build can be specified via command line options. For backward compatibility, it will use certain environment variables as default if they are defined.
- It supports site-specific configuration.
- It is very easy to extend.

Creating a build script using the Fab base class will require writing a Python script, and building an executable or library means executing this Python script. Knowledge of Fab is required when using the base class, since certain operations like checking out source files will use Fab commands.

### 4.10.1 Object-Oriented Design

The Fab base class is used to derive application-specific build script from. These build scripts can (and in general will need to) overwrite certain functions to tune the behaviour. For example, the base class provides a list of useful command line options. But any application-specific script can add additional command line options.

### 4.10.2 Command Line Options

The base class provides a list of commonly needed command line options. Any application-specific build script can add additional command line options. Invoking the base class itself with the `-h` command line option gives a description of the all options:

```
usage: fab_base.py [-h] [--suite SUITE] [--available-compilers] [--fc FC] [--cc CC] [--
↪ld LD] [--fflags FFLAGS] [--cflags CFLAGS] [--ldflags LDFLAGS] [--nprocs NPROCS]
                   [--mpi] [--no-mpi] [--openmp] [--no-openmp] [--openacc] [--host HOST]␣
↪[--site SITE] [--platform PLATFORM]

A Fab-based build system. Note that if --suite is specified, this will change the␣
↪default for compiler and linker

options:
  -h, --help            show this help message and exit
  --suite SUITE, -v SUITE
                        Sets the default suite for compiler and linker (default: None)
  --available-compilers
                        Displays the list of available compilers and linkers (default:␣
↪False)
  --fc FC, -fc FC       Name of the Fortran compiler to use (default: $FC)
  --cc CC, -cc CC       Name of the C compiler to use (default: $CC)
  --ld LD, -ld LD       Name of the linker to use (default: $LD)
```

(continues on next page)

(continued from previous page)

```
  --fflags FFLAGS, -fflags FFLAGS
                        Flags to be used by the Fortran compiler. The command line flags␣
↪are appended after compiler flags defined in a site-specific setup and after getting
                        flags from the environment variable $FFLAGS. Therefore, this can␣
↪be used to overwrite certain flags. (default: None)
  --cflags CFLAGS, -cflags CFLAGS
                        Flags to be used by the C compiler. The command line flags are␣
↪appended after compiler flags defined in a site-specific setup and after getting flags
                        from the environment variable $CFLAGS. Therefore, this can be␣
↪used to overwrite certain flags. (default: None)
  --ldflags LDFLAGS, -ldflags LDFLAGS
                        Flags to be used by the linker. The command line flags are␣
↪appended after linker flags defined in a site-specific setup and after getting flags␣
↪from
                        the environment variable $LDFLAGS. Therefore, this can be used␣
↪to overwrite certain flags. (default: None)
  --nprocs NPROCS, -n NPROCS
                        Number of processes (default is 1) (default: 1)
  --mpi, -mpi           Enable MPI (default: True)
  --no-mpi, -no-mpi     Disable MPI (default: True)
  --openmp, -openmp     Enable OpenMP (default: True)
  --no-openmp, -no-openmp
                        Disable OpenMP (default: True)
  --openacc, -openacc   Enable OpenACC (default: True)
  --host HOST, -host HOST
                        Determine the OpenACC or OpenMP: either 'cpu' or 'gpu'.␣
↪(default: cpu)
  --site SITE, -s SITE  Name of the site to use. (default: $SITE or 'default')
  --platform PLATFORM, -p PLATFORM
                        Name of the platform of the site to use. (default: $PLATFORM or
↪'default')
```

Some command line option have an environment variable as default (e.g. `-cc` uses `$CC` as default). If the corresponding environment variable is specified, its value will be used as default. If the variable is not defined, the argument is considered to be not specified.

### Processing in the Fab Base Class

This chapter describes the processing of an application script using the Fab base class. The knowledge of this process will indicate how a derived, application-specific build script can overwrite methods to customise the build process.

The full class documentation is at the end of this chapter.

The constructor sets up ultimately the Fab `BuildConfig` for the build. It takes the name of the application as argument. The name of the application will be used when creating the name of the build directory and it is also the default `root_symbol` when analysing the source code if the script creates an executable (see *analyse_step*).

The actual build is then started calling the `build` method of the created script. A typical outline of a build script is therefore:

```
from fab.fab_base import FabBase

class ExampleBuild(FabBase):
    # Additional methods to be overwritten or added
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    # Adjust logging level as appropriate
    logger = logging.getLogger('fab')
    logger.setLevel(logging.DEBUG)
    example = ExampleBuild(name="example")
    example.build()
```

The two main steps, construction and building, are described next.

### Constructor

As mentioned, the constructor will ultimately create the Fab `BuildConfig` object, which is then used to compile the application. The setup of this `BuildConfig` can be modified by overwriting the appropriate methods in a derived class, and site- and platform-specific setup can be done in a site- and platform-specific configuration script.

### Defining `site` and `platform`

The method `define_site_platform_target()` is first called. This method parses the arguments provided by the user and looks only for the options `--site` and `--platform` - any other option is for now ignored. These two arguments are then used to define a `target` attribute, which is either `default` or in the form `{site}_{platform}`. This name is used to identify the directory that contains the site-specific configuration script to be executed. The following three property getters can be used to access the values:

**property** FabBase.**platform:** str | None

**Returns**
the platform, or None if not specified.

**property** FabBase.**site:** str | None

**Returns**
the site, or None if no site is specified.

**property** FabBase.**target:** str

**Returns**
the target (=”site-platform”), or “default” if nothing was specified.

### Site-specific Configuration

After defining the site and platform, a site- and platform-specific configuration script is executed (if available). By default, the base class will try to import a module called `config` from the path `"site_specific/{target}"` (see *above* for the definition of `target`), relative to the directory in which the application script is located. By overwriting the method `setup_site_specific_location` an application can setup its own directories by adding paths to `sys.path`.

FabBase.**setup_site_specific_location**()

This method adds the required directories for site-specific configurations to the Python search path. This implementation will search the call tree to find the first call that’s not from Fab, i.e. the user script. It then adds `site_specific` and `site_specific/default` to the directory in which the user script is located. An application can overwrite this method to change this behaviour and point at site-specific directories elsewhere.

**Return type**
None

If the import is successful, the base class creates an instance of the `Config` class from the module. This all happens here:

`FabBase.site_specific_setup()`

Imports a site-specific config file. The location is based on the attribute `target` (which is set to be `{site}_{platform}" based on the command line options, and the path is specified in ``setup_site_specific_location`).

**Return type**
`None`

Remember if no site- and no platform-name is specified, it will import the `Config` class from the directory called `site_specific/default`.

Chapter *Site-Specific Configuration* describes this class in more detail. An example for the usage of site-specific configuration is to add new compilers to Fab's `ToolRepository`, or set the default compiler suite for a site.

### Defining command line options

After executing the site-specific configuration file, a Python argument parser is created with all command line options in the method `define_command_line_options`:

`FabBase.define_command_line_options(parser=None)`

Defines command line options. Can be overwritten by a derived class which can provide its own instance (to easily allow for a different description).

**Parameters**
**parser** (`Optional[ArgumentParser]`) – optional a pre-defined argument parser. If not, a new instance will be created. (default: `None`)

**Return type**
`ArgumentParser`

This method can be overwritten if an application want to add additional command line flags. The method gets an optional Python `ArgumentParser`: a derived class might want to provide its own instance to provide a better description message for the argument parser's help message. See *here* for an example.

A special case is the definition of compilation profiles (like `fast-debug` etc). The base class will query the site-specific configuration object using `get_valid_profiles()` to receive a list of all valid compilation profile names. This allows each site to specify its own profile modes.

### Parsing command line options

Once all command line options are defined in the parser, the parsing of the command line options happens in:

`FabBase.handle_command_line_options(parser)`

Analyse the actual command line options using the specified parser. The base implementation will handle the *–suite* parameter, and compiler/linker parameters (including the usage of environment variables). Needs to be overwritten to handle additional options specified by a derived script.

**Parameters**
**parser** (*argparse.ArgumentParser*) – the argument parser.

**Return type**
`None`

The result of the parsing is stored in an attribute, which can be accessed using the `args` property of the script instance.

Again, this method can be overwritten to handle the added application-specific command line options.

Once the application's `handle_command_line_options` has been executed, the method with the same name in the site-specific config file will also be called. It gets the argument namespace information from Python's ArgumentParser as argument:

`Config.`**`handle_command_line_options`**`(`*`args`*`)`

Additional callback function executed once all command line options have been added. This is for example used to add Vernier profiling flags, which are site-specific.

**Parameters**
**args** (*argparse.Namespace*) – the command line options added in the site configs

**Return type**
None

This can be used for further site-specific modifications, e.g. it might add additional flags for the compiler or linker. *Handling a new command line option* shows an example of doing this.

### Defining project name

By default, the base class will use `"{name}-{self.args.profile}-$compiler"` as the name for the project directory, i.e. the name of the project as specified in the constructor, followed by the compilation profile and compiler name (`$compiler` is a Python template parameter and will be replaced by Fab).

A user script can overwrite `define_project_name` and define a different name:

`FabBase.`**`define_project_name`**`(`*`name`*`)`

This method defines the project name, i.e. the directory name to use in the Fab workspace. It defaults to *name-profile-compiler*.

**Parameters**
**name** (str) – the base name of the project as specified by the caller.

**Return type**
str

**Returns**
the project name

Here an example where `-mpi` is added if MPI has been enabled on the command line. It calls the base class to add the compilation profile and compiler name.

```
def define_project_name(self, name: str) -> str:

    if self.args.mpi:
        name = name + "-mpi"
    return super().define_project_name(name)
```

### `BuildConfig` creation

After parsing the command line options, the base class will first create a Fab `ToolBox` which contains the compiler and linker selected by the user (see Fab documentation for details). Then it will create the `BuildConfig` object, providing the `ToolBox` and the appropriate command line options:

```
label = f"{name}-{self.args.profile}-$compiler"
self._config = BuildConfig(tool_box=self._tool_box,
                           project_label=label,
                           verbose=True,
                           n_procs=self.args.nprocs,
```

(continues on next page)

(continued from previous page)

```
                            mpi=self.args.mpi,
                            openmp=self.args.openmp,
                            profile=self.args.profile,
                            )
```

### Building

While Fab provides a very flexible way in which the different phases of the build process can be executed, the base clas provides a fixed order in which these steps happen (though of course the user could overwrite the `build` method to provide their own order). If additional phases need to be inserted into the build process, this can be done by overwriting the corresponding steps, see *Adding a new phase into the build process* for an example.

The naming of the steps follows the Fab naming, but adds a `_step` as suffix to distinguish the methods from the Fab functions. Typically, an application will need to overwrite at least some of these methods (for example to specify the source files). This will require either adding calls to Fab methods, or just calling the base-class. Details will be provided in each section below.

#### grab_files_step

This step is responsible for copying the required source files into the Fab work space (under the source folder). This method's template in the base class should not be called, otherwise it will raise an exception, since any script must specify where to get the source code from. Typically, in this step various Fab functions are used to get the source files:

**grab_folder**
: Fab's `grab_folder` recursively copies a file directory into the fab work space. It requires that the source files have been made available already, e.g. either as a local working copy, or a checkout from a repository.

**git_checkout**
: Fab's `git_checkout` checks out a git repository, and puts the files into the working directory.

**svn_export, svn_checkout**
: Fab provides these two interfaces to svn, and similar to `git_checkout` these will either export or checkout a Subversion repository.

**grab_archive**
: This wii unpack common archive formats like `tar`, `zip`, `tztar` etc.

**fcm_export, fcm_checkout**
: Compatibility layer to the old fcm configuration. This basically runs the corresponding Subversion commands.

A script can obviously use any other Python function to get or create source files.

#### find_source_files_step

This step is responsible for identifying the source files that are to be used in the build process. While Fab has the ability to analyse the source tree and determine the minimal necessary set of files, it is possible that different versions of the same file would be found in the source tree (e.g. different version of the same file coming from different repositories that have been checked out). Since Fab does not support using the same file name more than once (and since in general it would lead to inconsistency if the same file name is used), Fab provides the ability to include or exclude files from its source directory in the Fab work space.

TODO: link to Fab's documentation

This is typically done by specifying a list of path files. Each element in this list can be either an `Exclude` or an `Include` object, indicating that files of a specified pattern should be included or excluded. An example code:

```
path_filters = [
    Exclude('my_folder'),
    Include('my_folder/my_file.F90'),
]
```

These path files are then passed to Fab's `find_source_files` function. For example:

```
# Setting up path_filters as shows above
find_source_files(self.config,
                  path_filters=([Exclude('unit-test', '/test/')] +
                                path_filters))
```

This step will not affect any files, it will just set up Fab's `ArtefactStore` to be aware of the available source files.

Often, suites will provide FCM configuration that include a long list of files to exclude (and include) to avoid adding duplicated files into a complex build environment based on many source repositories.

```
extract.path-excl[um] = / # everything
extract.path-incl[um] =                                  \
    src/atmosphere/AC_assimilation/iau_mod.F90           \
    src/atmosphere/PWS_diagnostics/pws_diags_mod.F90 \
    src/atmosphere/aerosols/aero_params_mod.F90          \
    ...
```

For convenience during porting, Fab provides a small tool to interface with existing FCM configuration files. This tool can read existing FCM configuration files, and convert the `path-incl` and `path-excl` directives into Fab's `Exclude` and `Include` objects. Example usage:

```
extract_cfg = FcmExtract(self.lfric_apps_root / "build" / "extract" /
                         "extract.cfg")

science_root = self.config.source_root / 'science'
path_filters = []
for section, source_file_info in extract_cfg.items():
    for (list_type, list_of_paths) in source_file_info:
        if list_type == "exclude":
            # Exclude in this application removes the whole
            # app (so that individual files can then be picked
            # using include)
            path_filters.append(Exclude(science_root / section))
        else:
            # Remove the 'src' which is the first part of the name
            # in this script, which we don't have here
            new_paths = [i.relative_to(i.parents[-2])
                         for i in list_of_paths]
            for path in new_paths:
                path_filters.append(Include(science_root /
                                            section / path))
```

#### define_preprocessor_flags_step

This method is called before preprocessing, and it allows the application to specify all flags required for preprocessing all C, and Fortran files.

FabBase.**define_preprocessor_flags_step**()

Top level function that sets preprocessor flags. The base implementation does nothing, should be overwritten.

**Return type**
None

The base class provides its own method of adding preprocessor flags:

FabBase.**add_preprocessor_flags**(*list_of_flags*)

This function appends a preprocessor flags to the internal list of all preprocessor flags, which will be passed to Fab's various preprocessing steps (for C, Fortran, and X90).

Each flag can be either a str, or a path-specific instance of Fab's AddFlags object. For the convenience of the user, this function also accepts a single flag or a list of flags.

No checking will be done if a flag is already in the list of flags.

**Parameters**
**list_of_flags** (Union[*AddFlags*, str, List[*AddFlags*], List[str]]) – the preprocessor flag(s) to add. This can be either a `str` or an `AddFlags`, and in each case either a single item or a list.

**Return type**
None

Flags can be specified either as a single flag, or as a list of flags. Each flag can either be a simple string, which is a command line option for the compiler, or a path-specific flag using Fab's `AddFlags` class (TODO: link to fab). Example code:

```
def define_preprocessor_flags(self):
    super().define_preprocessor_flags()

    self.add_preprocessor_flags(['-DUM_PHYSICS',
                                 '-DCOUPLED',
                                 '-DUSE_MPI=YES'])

    path_flags = [AddFlags(match="$source/science/jules/*",
                           flags=['-DUM_JULES', '-I$output']),
                  AddFlags(match="$source/large_scale_precipitation/*",
                           flags=['-I$relative/include',
                                  '-I$source/science/shumlib/common/src'])]

    self.add_preprocessor_flags(path_flags)
    # Add a preprocessor flag depending on compilation profile:
    if self.args.profile == "full-debug":
        self.add_preprocessor_flags("-DDEBUG")
```

### preprocess_c_step

There is usually no reason to overwrite this method. It will use the preprocessor flags defined in the previous `define_preprocessor_flags_step` and preprocess all C files.

FabBase.**preprocess_c_step**()

Calls Fab's preprocessing of all C files. It passes the common and path-specific flags set using add_preprocessor_flags.

**Return type**
None

### preprocess_fortran_step

There is usually no reason to overwrite this method. It will use the preprocessor flags defined in the previous `define_preprocessor_flags_step` and preprocess all Fortran files.

FabBase.**preprocess_fortran_step**()

Calls Fab's preprocessing of all fortran files. It passes the common and path-specific flags set using add_preprocessor_flags.

**Return type**
None

### analyse_step

This steps does the complete dependency analysis for the application. There is usually no reason for an application to overwrite this step.

In case of creating a binary, the analyse step will use the root symbol, which defaults to the name of the application, but can be changed using `set_root_symbol`. This implies that `set_root_symbol` must be called before `analyse_step` is called, e.g. it can be called from any method called from the constructor (including defining and handling command line options).

FabBase.**analyse_step**(*find_programs=False*)

Calls Fab's analyse. It passes the config and root symbol for Fab to analyze the source code dependencies.

**Find_programs**
if set and an executable is created (see link_target), the flag will be set in Fab's analyse step, which means it will identify all main programs automatically.

**Return type**
None

### compile_c_step

This step compiles all C files. There is usually no reason for an application to overwrite this step.

FabBase.**compile_c_step**(*common_flags=None*, *path_flags=None*)

Calls Fab's compile_c. It passes the config for Fab to compile all C files. Optionally, common flags, path-specific flags and alternative source can also be passed to Fab for compilation.

**Return type**
None

### compile_fortran_step

This step compiles all Fortran files. As it takes a list of path-specific flags as an argument, child classes can overwrite this method to pass additional path-specific flags.

FabBase.**compile_fortran_step**(*common_flags=None*, *path_flags=None*)

Calls Fab's compile_fortran. It passes the config for Fab to compile all Fortran files. Optionally, common flags, path-specific flags and alternative source can also be passed to Fab for compilation.

**Parameters**
**path_flags** (Optional[List[*AddFlags*]]) – optional list of path-specific flags to be passed to Fab compile_fortran, default is None. (default: None)

**Return type**
None

#### archive_objects

This step creates an archive with all compiled object files.

**Warning**

Due to https://github.com/MetOffice/fab/issues/310 it is not recommended to create archives. Therefore, this step is for now not executed at all!

#### link_step

This step links all required object files into the executable or library. There is usually no reason for an application to overwrite this method.

FabBase.**link_step**()

Calls Fab's archive_objects for creating static libraries, or link_shared_object for creating shared libraries, or link_exe for creating executable binaries. The outputs will be placed in the Fab workspace, either using the name or root_symbol passed to the Fab build config.

**Return type**
None

### Site-specific Configuration Files

This chapter describes the design of the site-specific configuration files. It starts with the concept, and then includes some examples.

### Concepts for site-specific setup

Fab's base class supports site-specific setup, and it is is based on using a site name and a platform name. For example, The UK Met Office traditionally uses `meto` as site name, and then a different platform name, e.g. `xc40` or `ex1a`. The Fab base class uses a specific setup directory based on the concatenation of these names. In the example above, this would be `site_specific/meto_xc40` or `site_specific/meto_ex1a`. The site and platform can be specified as command line option (see *Command Line Options*). All these directories are stored under the `site_specific` directories to keep the directory structure cleaner.

If no site name is specified, `default` is used as site. And similarly, if no platform is specified, `default` is used as platform (resulting e.g. in `site_specific/meto-default` etc). If neither site nor platform is specified, the name `site_specific/default` is used.

Fab comes with a template for a `site_specific` setup. It only contains setting for the `default` site.

### Default configuration

It is strongly recommended for each application to have a default configuration file, which will define for example compiler profiles, and typical compiler flags. Any site-specific configuration file should then inherit from this default, but can also enhance the setup done by the default.

```
from default.config import Config as DefaultConfig

class Config(DefaultConfig):
    '''Make intel-classic the default compiler
    '''
    def __init__(self):
        super().__init__()
```

(continues on next page)

(continued from previous page)

```
        tr = ToolRepository()
        tr.set_default_compiler_suite("intel-classic")

        # Add a new compiler to the ToolRepository. It is
        # a compiler wrapper available for ifort and gfortran
        # on this site.
        for ftn in ["ifort", "gfortran"]:
            compiler = tr.get_tool(Category.FORTRAN_COMPILER, ftn)
            tr.add_tool(Tauf90(compiler))
```

### Callbacks in configuration files

The base class adds several calls to the site-specific configuration file, allowing site-specific changes to the build process. These callbacks are described here.

### Constructor

The constructor receives no parameter, and happens rather early in the processing chain (see *Defining site and platform*), i.e. at a stage where not even all command line options have been defined. Besides general setting up the object, adding new tools to Fab's `ToolRepository` can be done here.

#### get_valid_profiles

This method is called by `FabBase` when defining the command line options. It defines the list of valid compilation profile modes. This is used in setting up Python's `ArgumentParser` to only allow valid arguments.

`Config.`**`get_valid_profiles`**`()`

Determines the list of all allowed compiler profiles. The first entry in this list is the default profile to be used. This method can be overwritten by site configs to add or modify the supported profiles.

**Return type**
`List[str]`

**Returns**
list of all supported compiler profiles.

A well written default configuration file will take newly defined profiles into account and set them up automatically. See *Adding new compilation profiles* for an extended example.

#### handle_command_line_options

This method is called immediately after calling the application-specific `handle_command_line_options` method.

`Config.`**`handle_command_line_options`**`(`*`args`*`)`

Additional callback function executed once all command line options have been added. This is for example used to add Vernier profiling flags, which are site-specific.

**Parameters**
**`args`** (*`argparse.Namespace`*) – the command line options added in the site configs

**Return type**
`None`

It allows site-specific changes based on the specified command line options. An example is that selecting a hardware target (`--host` command line option) like GPU or CPU will require different compiler options. The following example will store all command line options of the user, and use them later when setting up the compiler:

```
def handle_command_line_options(self, args: argparse.Namespace) -> None:
    # Keep a copy of the args, so they can be used when
    # initialising compilers
    self._args = args
```

### update_toolbox

The `update_toolbox` method is called after the Fab `ToolBox` and `BuildConfig` objects have been created. All command line options have been parsed, and selected compilers have been added to the `ToolBox`.

`Config.`**`update_toolbox`**(*build_config*)

Set the default compiler flags for the various compiler that are supported.

**Parameters**
**build_config** (*BuildConfig*) – the Fab build configuration instance

**Return type**
None

Here is an example of defining the appropriate compilation profiles for all compilers and linkers:

```
def update_toolbox(self, build_config: BuildConfig) -> None:

    for compiler in (tr[Category.C_COMPILER] +
                     tr[Category.FORTRAN_COMPILER] +
                     tr[Category.LINKER]):
        compiler.define_profile("base", inherit_from="")
        for profile in self.get_valid_profiles():
            compiler.define_profile(profile, inherit_from="base")
```

This sets up a hierarchy where each of the valid compilation profiles inherits from a `base` profile. And they are defined for all compilers, even if they might not be available. This will make sure that using compilation modes work in a Fab compiler wrapper, since it is possible that the wrapped compiler is not available, i.e. not in `$PATH`, but the wrapper is. Additionally, using `get_valid_profiles` also means that any additional profiles defined from a derived class will automatically be created. If a different hierarchy is requested (e.g. `memory-profile` might want to inherit from `full-debug`, this needs to be updated in the inheriting class).

After the profiling modes, a `default` class should setup all compilers (including the various flags for the different compilation profiles). To continue the example from above, shown here is the code that uses the saved command line options from the user to setup flags for an Nvidia compiler:

```
def update_toolbox(self, build_config: BuildConfig) -> None:

    setup_nvidia(build_config, self.args)


def setup_nvidia(build_config: BuildConfig,
                 args: argparse.Namespace) -> None:

    tr = ToolRepository()
    nvfortran = tr.get_tool(Category.FORTRAN_COMPILER, "nvfortran")

    if args.openacc or args.openmp:
        host = args.host.lower()
    else:
```

(continues on next page)

(continued from previous page)

```
        # Neither openacc nor openmp specified
        host = ""

    flags = []
    if args.openacc:
        if host == "gpu":
            flags.extend(["-acc=gpu", "-gpu=managed"])
        else:
            # CPU
            flags.extend(["-acc=cpu"])
    ...
    nvfortran.add_flags(flags, "base")
```

### Examples

This chapter will contain some commented examples using the Fab base class.

### Common Usage Pattern

### Creating a new site and platform

To define a new site and platform, first create a directory with Fab base class is. In this directory, create a file called the name `f"{site}_{platform}"` in the directory in which the `config.py`, which must define a class called `Config`.

In general, it is recommended that any new config should inherit from the default configuration (since this will typically provide a good setup for various compilers). For example:

```
from default.config import Config as DefaultConfig

class Config(DefaultConfig):
    """
    A new site- and platform-specific setup file
    that sets the compiler default for this site to be
    intel-classic.
    """

    def __init__(self):
        super().__init__()
        tr = ToolRepository()
        tr.set_default_compiler_suite("intel-classic")
```

More methods can be overwritten to allow further customisation.

### Adding more command line options and better help messages

As outlined in *Defining command line options*, an application can implement its own `define_command_line_options` method. An example which adds a new `revision` flag:

```
class JulesBuild(FabBase):

    def define_command_line_options(self,
                                    parser: Optional[ArgumentParser] = None
                                    ) -> ArgumentParser:
```

(continues on next page)

(continued from previous page)

```
        """
        :param parser: optional a pre-defined argument parser. If not, a
            new instance will be created.
        """
        parser = super().define_command_line_options(parser)
        parser.add_argument(
            "--revision", "-r", type=str, default="vn7.8",
            help="Sets the Jules revision to checkout.")
        return parser
```

Since the Python argument parser also allows specification of a help message, this example can be extended as follows to provide a better message:

```
import argparse

class JulesBuild(FabBase):

    def define_command_line_options(self,
                                    parser: Optional[ArgumentParser] = None
                                    ) -> ArgumentParser:
        """
        :param parser: optional a pre-defined argument parser. If not, a
            new instance will be created.
        """

        # Allow another derived class to provide its own parser (with its
        # own description message). If not, create a parser with a better
        # description:
        if not parser:
            parser = argparse.ArgumentParser(
                description=("A Fab-based build system for Jules."),
                formatter_class=argparse.ArgumentDefaultsHelpFormatter)

        super().define_command_line_options(parser)
        parser.add_argument(
            "--revision", "-r", type=str, default="vn7.8",
            help="Sets the Jules revision to checkout.")
        return parser
```

### Handling a new command line option

As indicated in *Defining command line options*, the method `handle_command_line_options` can be overwritten to handle newly added command line options. Extending the previous examples of a Jules build script, here is how the revision of Jules is stored and then used:

```
class JulesBuild(FabBase):

    def handle_command_line_options(self, parser):
        """
        Grab the requested (or default) Jules revision to use and
        store it in an attribute.
        """
```

(continues on next page)

(continued from previous page)

```
        super().handle_command_line_options(parser)
        self._revision = self.args.revision

    def grab_files(self):
        """
        Extracts all the required source files from the repositories.
        """
        git_checkout(
            self.config,
            src="git@github.com:MetOffice/jules",
            revision=self._revision)
```

Strictly speaking, it is not necessary to store a command line option that is already included in `args` as a separate attribute as shown above - after all, the revision parameter could also be taken from `self.args.revision` instead. It is only done to make the code a little bit easier to read, and make this part of the code independent of the naming of the command line argument. If at some stage the command line option for the Jules revision needs to be changed, the actual extract step would not need to be changed.

### Adding a new phase into the build process

A new phase can be inserted in the build process by overwriting one of the existing steps, before or after which the new phase should be executed. Here an example that adds PSyclone processing for LFRic build script:

```
def preprocess_x90_step(self) -> None:
    """
    Invokes the Fab preprocess step for all X90 files.
    """
    # TODO: Fab does not support path-specific flags for X90 files.
    preprocess_x90(self.config,
                   common_flags=self.preprocess_flags_common)

def psyclone_step(self) -> None:
    """
    Call Fab's existing PSyclone step.
    """
    psyclone(...)

def analyse_step(self) -> None:
    '''
    The method overwrites the base class analyse_step.
    For LFRic, it first runs the preprocess_x90_step and then runs
    psyclone_step. Finally, it calls the original analyse step.
    '''
    self.preprocess_x90_step()
    self.psyclone_step()
    self.analyse_step()
```

A new step, i.e. one not already provided by Fab, is defined by using Fab's `step` fixture. For example, to define a new `remove_private_step`, the following code is used:

```
from fab.steps import step
```

(continues on next page)

(continued from previous page)

```
@step
def remove_private_step(self):
    ...

def psyclone_step(self):
  '''
  Overwriting the psyclone_step method added above
  '''
  self.remove_private_step()
  super().psyclone_step()
```

### Adding new compilation profiles

This can be done in site-specific configuration files. As shown in *Default configuration* it is recommended to use a `default` configuration, which will allow for consistency across sites. The following example shows how a site can then add its own compilation profile:

```
def get_valid_profiles(self) -> List[str]:
    '''
    Determines the list of all allowed compiler profiles. Here we
    add one additional profile `memory-debug`. Note that the default
    setup will automatically create that mode for any available compiler.

    :returns List[str]: list of all supported compiler profiles.
    '''
    return super().get_valid_profiles() + ["memory-debug"]
```

This code will add a new `memory-debug` option, which can be selected using the command line option `--profile memory-debug`. Of course, the site-specific config needs to then also set up this new mode. For example:

```
def update_toolbox(self, build_config: BuildConfig) -> None:
    '''
    Define additional profiling mode 'memory-debug'.

    :param build_config: the Fab build configuration instance
    :type build_config: :py:class:`fab.BuildConfig`
    '''

    # The base class needs to be called first to create all
    # profile modes - this will include the newly defined in
    # the above get_valid_profiles call:
    super().update_toolbox(build_config)

    tr = ToolRepository()

    # Define the new compilation profile `memory-debug`
    gfortran = tr.get_tool(Category.FORTRAN_COMPILER, "gfortran")
    gfortran.add_flags(["-fsanitize=address"], "memory-debug")

    linker = tr.get_tool(Category.LINKER, "linker-gfortran")
    linker.add_post_lib_flags(["-static-libasan"], "memory-debug")
```

## 4.11 API Documentation

This API documentation is generated from comments within the source code.

| | |
|---|---|
| *fab* | Flexible build system for scientific software. |

### 4.11.1 fab

Flexible build system for scientific software.

#### Exceptions

| | |
|---|---|
| *FabException* | |

**exception** fab.**FabException**

#### Modules

| | |
|---|---|
| *artefacts* | This module contains *Artefacts Getter* classes which return *Artefact Collections* from the *Artefact Store*. |
| *build_config* | Contains the *BuildConfig* and helper classes. |
| *cli* | Functions to run Fab from the command line. |
| *constants* | Some constants to help keep things tidy and manageable. |
| *dep_tree* | Classes and helper functions related to the dependency tree, as created by the analysis stage. |
| *fab_base* | A simple init file to make it shorter to import FabBase. |
| *logtools* | Logging tools for the fab framework. |
| *metrics* | A module for recording and summarising metrics, with the following concepts: |
| *mo* | A temporary place for some Met Office specific logic which, for now, needs to be integrated into Fab's internals. |
| *parse* | |
| *steps* | Predefined build steps with sensible defaults. |
| *tools* | A simple init file to make it shorter to import tools. |
| *util* | Various utility functions live here - until we give them a proper place to live! |

#### fab.artefacts

This module contains *Artefacts Getter* classes which return *Artefact Collections* from the *Artefact Store*.

These classes are used by the *run* method of `Step` classes to retrieve the artefacts which need to be processed. Most steps have sensible defaults and can be configured with user-defined getters.

### Classes

| | |
|---|---|
| *ArtefactSet*(value) | A simple enum with the artefact types used internally in Fab. |
| *ArtefactStore*() | This object stores sets of artefacts (which can be of any type). |
| *ArtefactsGetter*() | Abstract base class for artefact getters. |
| *CollectionConcat*(collections) | Returns a concatenated list from multiple *Artefact Collections* (each expected to be an iterable). |
| *CollectionGetter*(collection_name) | A simple artefact getter which returns one *Artefact Collection* from the artefact_store. |
| *FilterBuildTrees*(suffix) | Filter build trees by suffix. |
| *SuffixFilter*(collection_name, suffix) | Returns the file paths in a *Artefact Collection* (expected to be an iterable), filtered by suffix. |

**class** fab.artefacts.**ArtefactSet**(*value*)

A simple enum with the artefact types used internally in Fab.

**class** fab.artefacts.**ArtefactStore**

This object stores sets of artefacts (which can be of any type). Each artefact is indexed by either an ArtefactSet enum, or a string.

The constructor calls reset, which will mean all the internal artefact categories are created.

**reset**()

Clears the artefact store (but does not delete any files).

**add**(*collection*, *files*)

Adds the specified artefacts to a collection. The artefact can be specified as a simple string, a list of string or a set, in which case all individual entries of the list/set will be added. :type collection: Union[str, *ArtefactSet*] :param collection: the name of the collection to add this to. :type files: Union[Path, Iterable[Path]] :param files: the artefacts to add.

**update_dict**(*collection*, *values*, *key=None*)

Modifies data associated with artefact set.

**Parameters**

- **collection** (Union[str, *ArtefactSet*]) – Name or enumeration of set to modify.
- **values** (Union[Path, Iterable[Path]]) – New data for set.
- **key** (Optional[str]) – Executable name associated with data. Do not specify for libraries. (default: None)

**copy_artefacts**(*source*, *dest*, *suffixes=None*)

Copies all artefacts from *source* to *destination*. If a suffix_fiter is specified, only files with the given suffix will be copied.

**Parameters**

- **source** (Union[str, *ArtefactSet*]) – the source artefact set.
- **dest** (Union[str, *ArtefactSet*]) – the destination artefact set.
- **suffixes** (Union[str, List[str], None]) – a string or list of strings specifying the suffixes to copy. (default: None)

**replace**(*artefact*, *remove_files*, *add_files*)

Replaces artefacts in one artefact set with other artefacts. This can be used e.g to replace files that have been preprocessed and renamed. There is no requirement for these lists to have the same number of elements, nor is there any check if an artefact to be removed is actually in the artefact set.

**Parameters**

- **artefact** (Union[str, *ArtefactSet*]) – the artefact set to modify.
- **remove_files** (List[Union[str, Path]]) – files to remove from the artefact set.
- **add_files** (Union[List[Union[str, Path]], dict]) – files to add to the artefact set.

**class** fab.artefacts.**ArtefactsGetter**

Abstract base class for artefact getters.

**class** fab.artefacts.**CollectionGetter**(*collection_name*)

A simple artefact getter which returns one *Artefact Collection* from the artefact_store.

Example:

```
`CollectionGetter('preprocessed_fortran')`
```

**Parameters**

**collection_name** (Union[str, *ArtefactSet*]) – The name of the artefact collection to retrieve.

**class** fab.artefacts.**CollectionConcat**(*collections*)

Returns a concatenated list from multiple *Artefact Collections* (each expected to be an iterable).

An *ArtefactsGetter* can be provided instead of a collection_name.

Example:

```
# The default source code getter for the Analyse step might look
# like this.
DEFAULT_SOURCE_GETTER = CollectionConcat([
    'preprocessed_c',
    'preprocessed_fortran',
    SuffixFilter(ArtefactSet.INITIAL_SOURCE, '.f90'),
])
```

**Parameters**

**collections** (Iterable[Union[*ArtefactSet*, str, *ArtefactsGetter*]]) – An iterable containing collection names (strings) or other ArtefactsGetters.

**class** fab.artefacts.**SuffixFilter**(*collection_name*, *suffix*)

Returns the file paths in a *Artefact Collection* (expected to be an iterable), filtered by suffix.

Example:

```
# The default source getter for the FortranPreProcessor step.
DEFAULT_SOURCE = SuffixFilter(ArtefactSet.INITIAL_SOURCE, '.F90')
```

**Parameters**

- **collection_name** (Union[str, *ArtefactSet*]) – The name of the artefact collection.

- **suffix** (Union[str, List[str]]) – A suffix string including the dot, or iterable of.

class fab.artefacts.**FilterBuildTrees**(*suffix*)

Filter build trees by suffix.

Example:

```
# The default source getter for the CompileFortran step.
DEFAULT_SOURCE_GETTER = FilterBuildTrees(suffix='.f90')
```

**Returns**
one list of files to compile per build tree, of the form Dict[name, List[AnalysedDependent]]

**Parameters**
**suffix** (Union[str, List[str]]) – A suffix string, or iterable of, including the preceding dot.

### fab.build_config

Contains the *BuildConfig* and helper classes.

### Classes

| | |
|---|---|
| *AddFlags*(match, flags) | Add command-line flags when our path filter matches. |
| *BuildConfig*(project_label, tool_box[, mpi, ...]) | Contains and runs a list of build steps. |
| *FlagsConfig*([common_flags, path_flags]) | Return command-line flags for a given path. |

class fab.build_config.**BuildConfig**(*project_label*, *tool_box*, *mpi=False*, *openmp=False*, *profile=None*, *multiprocessing=True*, *n_procs=None*, *reuse_artefacts=False*, *fab_workspace=None*, *two_stage=False*, *verbose=False*)

Contains and runs a list of build steps.

The user is not expected to instantiate this class directly, but rather through the build_config() context manager.

**Parameters**

- **project_label** (str) – Name of the build project. The project workspace folder is created from this name, with spaces replaced by underscores.
- **tool_box** (*ToolBox*) – The ToolBox with all tools to use in the build.
- **mpi** (bool) – whether the project uses MPI or not. This is used to pick a default compiler (if none is explicitly set in the ToolBox), and controls PSyclone parameters. (default: False)
- **openmp** (bool) – as with *mpi*, this controls whether the project is using OpenMP or not. This is used to pick a default compiler (if none is explicitly set in the ToolBox). The compiler-specific flag to enable OpenMP will automatically be added when compiling and linking. (default: False)
- **profile** (Optional[str]) – the name of a compiler profile to use. (default: None)
- **multiprocessing** (bool) – An option to disable multiprocessing to aid debugging. (default: True)
- **n_procs** (Optional[int]) – The number of cores to use for multiprocessing operations. Defaults to the number of available cores. (default: None)

- **reuse_artefacts** (bool) – A flag to avoid reprocessing certain files on subsequent runs. WARNING: Currently unsophisticated, this flag should only be used by Fab developers. The logic behind flag will soon be improved, in a work package called "incremental build". (default: False)
- **fab_workspace** (Optional[Path]) – Overrides the FAB_WORKSPACE environment variable. If not set, and FAB_WORKSPACE is not set, the fab workspace defaults to *~/fab-workspace*. (default: None)
- **two_stage** (bool) – Compile .mod files first in a separate pass. Theoretically faster in some projects. (default: False)
- **verbose** (bool) – DEBUG level logging. (default: False)

property tool_box: *ToolBox*

**Returns**
the tool box to use.

property artefact_store: *ArtefactStore*

**Returns**
the Artefact instance for this configuration.

property project_workspace: Path

**Returns**
the project workspace path.

property build_output: Path

**Returns**
the build output path.

property mpi: bool

**Returns**
whether MPI is requested or not in this config.

property openmp: bool

**Returns**
whether OpenMP is requested or not in this config.

property profile: str

**Returns**
the name of the compiler profile to use.

add_current_prebuilds(*artefacts*)

Mark the given file paths as being current prebuilds, not to be cleaned during housekeeping.

class fab.build_config.**AddFlags**(*match*, *flags*)

Add command-line flags when our path filter matches. Generally used inside a *FlagsConfig*.

**Parameters**

- **match** (str) – The string to match against each file path.
- **flags** (List[str]) – The command-line flags to add for matching files.

Both the *match* and *flags* arguments can make use of templating:

- *$source* for *<project workspace>/source*

- *$output* for *<project workspace>/build_output*
- *$relative* for *<the source file's folder>*

For example:

```
# For source in the um folder, add an absolute include path
AddFlags(match="$source/um/*", flags=['-I$source/include']),

# For source in the um folder, add an include path relative to
# each source file.
AddFlags(match="$source/um/*", flags=['-I$relative/include']),
```

**run**(*fpath*, *input_flags*, *config*)

Check if our filter matches a given file. If it does, add our flags.

**Parameters**

- **fpath** (Path) – Filepath to check.
- **input_flags** (List[str]) – The list of command-line flags Fab is building for this file.
- **config** – Contains the folders for templating *$source* and *$output*.

**class** fab.build_config.**FlagsConfig**(*common_flags=None*, *path_flags=None*)

Return command-line flags for a given path.

Simply allows appending flags but may evolve to also replace and remove flags.

**Parameters**

- **common_flags** (Optional[List[str]]) – List of flags to apply to all files. E.g *['-O2']*. (default: None)
- **path_flags** (Optional[List[*AddFlags*]]) – List of *AddFlags* objects which apply flags to selected paths. (default: None)

**flags_for_path**(*path*, *config*)

Get all the flags for a given file, in a reproducible order.

**Parameters**

- **path** (Path) – The file path for which we want command-line flags.
- **config** – The config contains the source root and project workspace.

### fab.cli

Functions to run Fab from the command line.

### Functions

| | |
|---|---|
| *cli_fab*([folder, kwargs]) | Running Fab from the command line will attempt to build the project in the current or given folder. |

fab.cli.**cli_fab**(*folder=None*, *kwargs=None*)

Running Fab from the command line will attempt to build the project in the current or given folder. The following params are used for testing. When run normally any parameters will be caught by a common_arg_parser.

**Parameters**

- **folder** (Optional[Path]) – source folder (Testing Only) (default: None)
- **kwargs** (Optional[Dict]) – parameters ( Testing Only ) (default: None)

### fab.constants

Some constants to help keep things tidy and manageable.

### fab.dep_tree

Classes and helper functions related to the dependency tree, as created by the analysis stage.

### Functions

| | |
|---|---|
| *extract_sub_tree*(source_tree, root[, verbose]) | Extract the subtree required to build the target, from the full source tree of all analysed source files. |
| *filter_source_tree*(source_tree, suffixes) | Pull out files with the given extensions from a source tree. |
| *validate_dependencies*(source_tree) | If any dep is missing from the tree, then it's unknown code and we won't be able to compile. |

### Classes

| | |
|---|---|
| *AnalysedDependent*(fpath[, file_hash, ...]) | An *AnalysedFile* which can depend on others, and be a dependency. |

**class** fab.dep_tree.**AnalysedDependent**(*fpath*, *file_hash=None*, *symbol_defs=None*, *symbol_deps=None*, *file_deps=None*)

An *AnalysedFile* which can depend on others, and be a dependency. Instances of this class are nodes in a source dependency tree.

During parsing, the symbol definitions and dependencies are filled in. During dependency analysis, symbol dependencies are turned into file dependencies.

**Parameters**

- **fpath** (Union[str, Path]) – The source file that was analysed.
- **file_hash** (Optional[int]) – The hash of the source. If omitted, Fab will evaluate lazily. (default: None)
- **symbol_defs** (Optional[Iterable[str]]) – Set of symbol names defined by this source file. (default: None)
- **symbol_deps** (Optional[Iterable[str]]) – Set of symbol names used by this source file. Can include symbols in the same file. (default: None)
- **file_deps** (Optional[Iterable[Path]]) – Other files on which this source depends. Must not include itself. This attribute is calculated during symbol analysis, after everything has been parsed. (default: None)

**classmethod field_names**()

Defines the order in which we want fields to appear in str or repr strings.

Calling this helps to ensure any lazy attributes are evaluated before use, e.g when constructing a string representation of the instance, or generating a hash value.

`to_dict()`

Create a dict representing the object.

The dict may be written to json, so can't contain sets. Lists are sorted for reproducibility in testing.

**Return type**
`Dict[str, Any]`

`fab.dep_tree.extract_sub_tree(source_tree, root, verbose=False)`

Extract the subtree required to build the target, from the full source tree of all analysed source files.

**Parameters**

- **source_tree** (`Dict[Path, AnalysedDependent]`) – The source tree of analysed files.
- **root** (`Path`) – The root of the dependency tree, this is the filename containing the Fortran program.
- **verbose** – Log missing dependencies. (default: `False`)

**Return type**
`Dict[Path, AnalysedDependent]`

`fab.dep_tree.filter_source_tree(source_tree, suffixes)`

Pull out files with the given extensions from a source tree.

Returns a list of `AnalysedDependent`.

**Parameters**

- **source_tree** (`Dict[Path, AnalysedDependent]`) – The source tree of analysed files.
- **suffixes** (`Iterable[str]`) – The suffixes we want, including the dot.

**Return type**
`List[AnalysedDependent]`

`fab.dep_tree.validate_dependencies(source_tree)`

If any dep is missing from the tree, then it's unknown code and we won't be able to compile.

**Parameters**
**source_tree** – The source tree of analysed files.

### fab.fab_base

A simple init file to make it shorter to import FabBase.

`class fab.fab_base.FabBase(name, link_target='executable')`

This is a convenience base class for writing Fab scripts. It provides an extensive set of command line options that can be used to influence the build process, and can be extended by applications. It has already support for compilation modes, and allows site-specific configuration scripts to be written that can modify the initialisation and build process.

**Parameters**

- **name** (`str`) – the name to be used for the workspace. Note that the name of the compiler will be added to it.
- **link_target** (`str`) – what target should be created. Must be one of "executable" (default), "static-library", or "shared-library" (default: `'executable'`)

set_link_target(*link_target*)

Sets the link target.

**Parameters**
**link_target** (str) – what target should be created. Must be one of "executable"s, "static-library", or "shared-library".

**Raises**
ValueError – if the link_target is invalid

**Return type**
None

define_project_name(*name*)

This method defines the project name, i.e. the directory name to use in the Fab workspace. It defaults to *name-profile-compiler*.

**Parameters**
**name** (str) – the base name of the project as specified by the caller.

**Return type**
str

**Returns**
the project name

set_root_symbol(*root_symbol*)

Defines the root symbol. It defaults to the name given in the constructor.

**Parameters**
**name** – the root symbol to use when creating a binary (unused otherwise).

**Return type**
None

property root_symbol: List[str]

**Returns**
the list of root symbols.

property site: str | None

**Returns**
the site, or None if no site is specified.

property logger: Logger

**Returns**
the logging instance to use.

property platform: str | None

**Returns**
the platform, or None if not specified.

property target: str

**Returns**
the target (="site-platform"), or "default" if nothing was specified.

property config: *BuildConfig*

**Returns**
the FAB BuildConfig instance.

**Return type**
fab.BuildConfig

property args: Namespace

**Returns**
the arg parse objects containing the user's command line information.

property project_workspace: Path

**Returns**
the Fab workspace for this build.

property preprocess_flags_common: List[str]

**Returns**
the list of all common preprocessor flags.

property preprocess_flags_path: List[*AddFlags*]

**Returns**
the list of all path-specific flags.

property fortran_compiler_flags_commandline: List[str]

**Returns**
the list of flags specified through –fflags.

property c_compiler_flags_commandline: List[str]

**Returns**
the list of flags specified through –cflags.

property linker_flags_commandline: List[str]

**Returns**
the list of flags specified through –ldflags.

setup_site_specific_location()

This method adds the required directories for site-specific configurations to the Python search path. This implementation will search the call tree to find the first call that's not from Fab, i.e. the user script. It then adds `site_specific` and `site_specific/default` to the directory in which the user script is located. An application can overwrite this method to change this behaviour and point at site-specific directories elsewhere.

**Return type**
None

define_site_platform_target()

This method defines the attributes site, platform (and target=site-platform) based on the command line option –site and –platform (using $SITE and $PLATFORM as a default). If site or platform is missing and the corresponding environment variable is not set, 'default' will be used.

**Return type**
None

**site_specific_setup**()

Imports a site-specific config file. The location is based on the attribute `target` (which is set to be `{site}_{platform}" based on the command line options, and the path is specified in ``setup_site_specific_location`).

**Return type**
None

**define_command_line_options**(*parser=None*)

Defines command line options. Can be overwritten by a derived class which can provide its own instance (to easily allow for a different description).

**Parameters**
**parser** (Optional[ArgumentParser]) – optional a pre-defined argument parser. If not, a new instance will be created. (default: None)

**Return type**
ArgumentParser

**handle_command_line_options**(*parser*)

Analyse the actual command line options using the specified parser. The base implementation will handle the *–suite* parameter, and compiler/linker parameters (including the usage of environment variables). Needs to be overwritten to handle additional options specified by a derived script.

**Parameters**
**parser** (*argparse.ArgumentParser*) – the argument parser.

**Return type**
None

**define_preprocessor_flags_step**()

Top level function that sets preprocessor flags. The base implementation does nothing, should be overwritten.

**Return type**
None

**get_linker_flags**()

Base class for setting linker flags. This base implementation for now just returns an empty list.

**Return type**
List[str]

**Returns**
list of flags for the linker.

**add_preprocessor_flags**(*list_of_flags*)

This function appends a preprocessor flags to the internal list of all preprocessor flags, which will be passed to Fab's various preprocessing steps (for C, Fortran, and X90).

Each flag can be either a str, or a path-specific instance of Fab's AddFlags object. For the convenience of the user, this function also accepts a single flag or a list of flags.

No checking will be done if a flag is already in the list of flags.

**Parameters**
**list_of_flags** (Union[*AddFlags*, str, List[*AddFlags*], List[str]]) – the preprocessor flag(s) to add. This can be either a `str` or an `AddFlags`, and in each case either a single item or a list.

**Return type**
None

`grab_files_step()`

This should typically be overwritten by an application to get files e.g. from a repository.

**Return type**
None

`find_source_files_step(path_filters=None)`

This function calls Fab's find_source_files, to identify and add all source files to Fab's artefact store.

**Parameters**
**path_filters** (Optional[Iterable[Union[*Exclude*, *Include*]]]) – optional list of path filters to be passed to Fab find_source_files, default is None. (default: None)

**Return type**
None

`preprocess_c_step()`

Calls Fab's preprocessing of all C files. It passes the common and path-specific flags set using add_preprocessor_flags.

**Return type**
None

`preprocess_fortran_step()`

Calls Fab's preprocessing of all fortran files. It passes the common and path-specific flags set using add_preprocessor_flags.

**Return type**
None

`analyse_step(find_programs=False)`

Calls Fab's analyse. It passes the config and root symbol for Fab to analyze the source code dependencies.

**Find_programs**
if set and an executable is created (see link_target), the flag will be set in Fab's analyse step, which means it will identify all main programs automatically.

**Return type**
None

`compile_c_step(common_flags=None, path_flags=None)`

Calls Fab's compile_c. It passes the config for Fab to compile all C files. Optionally, common flags, path-specific flags and alternative source can also be passed to Fab for compilation.

**Return type**
None

`compile_fortran_step(common_flags=None, path_flags=None)`

Calls Fab's compile_fortran. It passes the config for Fab to compile all Fortran files. Optionally, common flags, path-specific flags and alternative source can also be passed to Fab for compilation.

**Parameters**
**path_flags** (Optional[List[*AddFlags*]]) – optional list of path-specific flags to be passed to Fab compile_fortran, default is None. (default: None)

**Return type**
None

`link_step`()

Calls Fab's archive_objects for creating static libraries, or link_shared_object for creating shared libraries, or link_exe for creating executable binaries. The outputs will be placed in the Fab workspace, either using the name or root_symbol passed to the Fab build config.

**Return type**
None

`build`()

This function defines the build process for Fab. Generally, a build process involves grabbing and finding Fortran and C source files for proprocessing, dependency analysis, compilation and linking.

**Return type**
None

## Modules

| | |
|---|---|
| *fab_base* | This is an OO basic interface to FAB. |
| *site_specific* | |

## fab.fab_base.fab_base

This is an OO basic interface to FAB. It allows typical applications to only modify very few settings to have a working FAB build script.

## Classes

| | |
|---|---|
| *FabBase*(name[, link_target]) | This is a convenience base class for writing Fab scripts. |

**class** fab.fab_base.fab_base.**FabBase**(*name*, *link_target='executable'*)

This is a convenience base class for writing Fab scripts. It provides an extensive set of command line options that can be used to influence the build process, and can be extended by applications. It has already support for compilation modes, and allows site-specific configuration scripts to be written that can modify the initialisation and build process.

**Parameters**

- **name** (str) – the name to be used for the workspace. Note that the name of the compiler will be added to it.
- **link_target** (str) – what target should be created. Must be one of "executable" (default), "static-library", or "shared-library" (default: `'executable'`)

`set_link_target`(*link_target*)

Sets the link target.

**Parameters**
**link_target** (str) – what target should be created. Must be one of "executable"s, "static-library", or "shared-library".

**Raises**
**ValueError** – if the link_target is invalid

**Return type**
None

**define_project_name**(*name*)

This method defines the project name, i.e. the directory name to use in the Fab workspace. It defaults to *name-profile-compiler*.

**Parameters**
**name** (str) – the base name of the project as specified by the caller.

**Return type**
str

**Returns**
the project name

**set_root_symbol**(*root_symbol*)

Defines the root symbol. It defaults to the name given in the constructor.

**Parameters**
**name** – the root symbol to use when creating a binary (unused otherwise).

**Return type**
None

**property root_symbol: List[str]**

**Returns**
the list of root symbols.

**property site: str | None**

**Returns**
the site, or None if no site is specified.

**property logger: Logger**

**Returns**
the logging instance to use.

**property platform: str | None**

**Returns**
the platform, or None if not specified.

**property target: str**

**Returns**
the target (=”site-platform”), or “default” if nothing was specified.

**property config: *BuildConfig***

**Returns**
the FAB BuildConfig instance.

**Return type**
fab.BuildConfig

**property args: Namespace**

**Returns**
the arg parse objects containing the user’s command line information.

property project_workspace: Path

**Returns**
the Fab workspace for this build.

property preprocess_flags_common: List[str]

**Returns**
the list of all common preprocessor flags.

property preprocess_flags_path: List[*AddFlags*]

**Returns**
the list of all path-specific flags.

property fortran_compiler_flags_commandline: List[str]

**Returns**
the list of flags specified through –fflags.

property c_compiler_flags_commandline: List[str]

**Returns**
the list of flags specified through –cflags.

property linker_flags_commandline: List[str]

**Returns**
the list of flags specified through –ldflags.

setup_site_specific_location()

This method adds the required directories for site-specific configurations to the Python search path. This implementation will search the call tree to find the first call that's not from Fab, i.e. the user script. It then adds `site_specific` and `site_specific/default` to the directory in which the user script is located. An application can overwrite this method to change this behaviour and point at site-specific directories elsewhere.

**Return type**
None

define_site_platform_target()

This method defines the attributes site, platform (and target=site-platform) based on the command line option –site and –platform (using $SITE and $PLATFORM as a default). If site or platform is missing and the corresponding environment variable is not set, 'default' will be used.

**Return type**
None

site_specific_setup()

Imports a site-specific config file. The location is based on the attribute `target` (which is set to be `{site}_{platform}" based on the command line options, and the path is specified in ``setup_site_specific_location`).

**Return type**
None

define_command_line_options(*parser=None*)

Defines command line options. Can be overwritten by a derived class which can provide its own instance (to easily allow for a different description).

**Parameters**
**parser** (Optional[ArgumentParser]) – optional a pre-defined argument parser. If not, a new instance will be created. (default: None)

**Return type**
ArgumentParser

**handle_command_line_options**(*parser*)

Analyse the actual command line options using the specified parser. The base implementation will handle the *–suite* parameter, and compiler/linker parameters (including the usage of environment variables). Needs to be overwritten to handle additional options specified by a derived script.

**Parameters**
**parser** (*argparse.ArgumentParser*) – the argument parser.

**Return type**
None

**define_preprocessor_flags_step**()

Top level function that sets preprocessor flags. The base implementation does nothing, should be overwritten.

**Return type**
None

**get_linker_flags**()

Base class for setting linker flags. This base implementation for now just returns an empty list.

**Return type**
List[str]

**Returns**
list of flags for the linker.

**add_preprocessor_flags**(*list_of_flags*)

This function appends a preprocessor flags to the internal list of all preprocessor flags, which will be passed to Fab's various preprocessing steps (for C, Fortran, and X90).

Each flag can be either a str, or a path-specific instance of Fab's AddFlags object. For the convenience of the user, this function also accepts a single flag or a list of flags.

No checking will be done if a flag is already in the list of flags.

**Parameters**
**list_of_flags** (Union[*AddFlags*, str, List[*AddFlags*], List[str]]) – the preprocessor flag(s) to add. This can be either a str or an AddFlags, and in each case either a single item or a list.

**Return type**
None

**grab_files_step**()

This should typically be overwritten by an application to get files e.g. from a repository.

**Return type**
None

**find_source_files_step**(*path_filters=None*)

This function calls Fab's find_source_files, to identify and add all source files to Fab's artefact store.

**Parameters**
**path_filters** (Optional[Iterable[Union[*Exclude*, *Include*]]]) – optional list of path filters to be passed to Fab find_source_files, default is None. (default: None)

**Return type**
None

`preprocess_c_step()`

Calls Fab's preprocessing of all C files. It passes the common and path-specific flags set using add_preprocessor_flags.

**Return type**
None

`preprocess_fortran_step()`

Calls Fab's preprocessing of all fortran files. It passes the common and path-specific flags set using add_preprocessor_flags.

**Return type**
None

`analyse_step(find_programs=False)`

Calls Fab's analyse. It passes the config and root symbol for Fab to analyze the source code dependencies.

**Find_programs**
if set and an executable is created (see link_target), the flag will be set in Fab's analyse step, which means it will identify all main programs automatically.

**Return type**
None

`compile_c_step(common_flags=None, path_flags=None)`

Calls Fab's compile_c. It passes the config for Fab to compile all C files. Optionally, common flags, path-specific flags and alternative source can also be passed to Fab for compilation.

**Return type**
None

`compile_fortran_step(common_flags=None, path_flags=None)`

Calls Fab's compile_fortran. It passes the config for Fab to compile all Fortran files. Optionally, common flags, path-specific flags and alternative source can also be passed to Fab for compilation.

**Parameters**
**path_flags** (Optional[List[*AddFlags*]]) – optional list of path-specific flags to be passed to Fab compile_fortran, default is None. (default: None)

**Return type**
None

`link_step()`

Calls Fab's archive_objects for creating static libraries, or link_shared_object for creating shared libraries, or link_exe for creating executable binaries. The outputs will be placed in the Fab workspace, either using the name or root_symbol passed to the Fab build config.

**Return type**
None

`build()`

This function defines the build process for Fab. Generally, a build process involves grabbing and finding Fortran and C source files for proprocessing, dependency analysis, compilation and linking.

**Return type**
None

### fab.fab_base.site_specific

### Modules

| | |
|---|---|
| *default* | |

### fab.fab_base.site_specific.default

### Modules

| | |
|---|---|
| *config* | This module contains the default Baf configuration class. |
| *setup_cray* | This file contains a function that sets the default flags for the Cray compilers and linkers in the ToolRepository. |
| *setup_gnu* | This file contains a function that sets the default flags for all GNU based compilers and linkers in the ToolRepository. |
| *setup_intel_classic* | This file contains a function that sets the default flags for all Intel classic based compilers in the ToolRepository (ifort, icc). |
| *setup_intel_llvm* | This file contains a function that sets the default flags for all Intel llvm based compilers and linkers in the ToolRepository (ifx, icx). |
| *setup_nvidia* | This file contains a function that sets the default flags for the NVIDIA compilers and linkers in the ToolRepository. |

### fab.fab_base.site_specific.default.config

This module contains the default Baf configuration class.

### Classes

| | |
|---|---|
| *Config*() | This class is the default Configuration object for Baf builds. |

**class** fab.fab_base.site_specific.default.config.**Config**

This class is the default Configuration object for Baf builds. It provides several callbacks which will be called from the build scripts to allow site-specific customisations.

**property args:** Namespace

**Returns**
the command line options specified by the user.

**get_valid_profiles**()

Determines the list of all allowed compiler profiles. The first entry in this list is the default profile to be used. This method can be overwritten by site configs to add or modify the supported profiles.

**Return type**
List[str]

**Returns**
list of all supported compiler profiles.

**handle_command_line_options**(*args*)

Additional callback function executed once all command line options have been added. This is for example used to add Vernier profiling flags, which are site-specific.

**Parameters**
**args** (*argparse.Namespace*) – the command line options added in the site configs

**Return type**
None

**update_toolbox**(*build_config*)

Set the default compiler flags for the various compiler that are supported.

**Parameters**
**build_config** (*BuildConfig*) – the Fab build configuration instance

**Return type**
None

**get_path_flags**(*build_config*)

Returns the path-specific flags to be used. TODO #313: Ideally we have only one kind of flag, but as a quick work around we provide this method.

**Return type**
List[*AddFlags*]

**setup_cray**(*build_config*)

This method sets up the Cray compiler and linker flags. For now call an external function, since it is expected that this configuration can be very lengthy (once we support compiler modes).

**Parameters**
**build_config** (*BuildConfig*) – the Fab build configuration instance

**Return type**
None

**setup_gnu**(*build_config*)

This method sets up the Gnu compiler and linker flags. For now call an external function, since it is expected that this configuration can be very lengthy (once we support compiler modes).

**Parameters**
**build_config** (*BuildConfig*) – the Fab build configuration instance

**Return type**
None

**setup_intel_classic**(*build_config*)

This method sets up the Intel classic compiler and linker flags. For now call an external function, since it is expected that this configuration can be very lengthy (once we support compiler modes).

**Parameters**
**build_config** (*BuildConfig*) – the Fab build configuration instance

**Return type**
None

**setup_intel_llvm**(*build_config*)

This method sets up the Intel LLVM compiler and linker flags. For now call an external function, since it is expected that this configuration can be very lengthy (once we support compiler modes).

**Parameters**
**build_config** (*BuildConfig*) – the Fab build configuration instance

**Return type**
None

**setup_nvidia**(*build_config*)

This method sets up the Nvidia compiler and linker flags. For now call an external function, since it is expected that this configuration can be very lengthy (once we support compiler modes).

**Parameters**
**build_config** (*BuildConfig*) – the Fab build configuration instance

**Return type**
None

### fab.fab_base.site_specific.default.setup_cray

This file contains a function that sets the default flags for the Cray compilers and linkers in the ToolRepository.

This function gets called from the default site-specific config file

### Functions

| | |
|---|---|
| *setup_cray*(build_config, args) | Defines the default flags for ftn. |

fab.fab_base.site_specific.default.setup_cray.**setup_cray**(*build_config*, *args*)

Defines the default flags for ftn.

**Parameters**

- **build_config** (*BuildConfig*) – the Fab build config instance from which required parameters can be taken.
- **args** (Namespace) – all command line options

**Return type**
Dict[str, List[*AddFlags*]]

### fab.fab_base.site_specific.default.setup_gnu

This file contains a function that sets the default flags for all GNU based compilers and linkers in the ToolRepository.

This function gets called from the default site-specific config file

### Functions

| | |
|---|---|
| *setup_gnu*(build_config, args) | Defines the default flags for all GNU compilers and linkers. |

fab.fab_base.site_specific.default.setup_gnu.**setup_gnu**(*build_config*, *args*)

Defines the default flags for all GNU compilers and linkers.

**Parameters**

- **build_config** (*BuildConfig*) – the Fab build config instance from which required parameters can be taken.
- **args** (Namespace) – all command line options

**Return type**
Dict[str, List[*AddFlags*]]

### fab.fab_base.site_specific.default.setup_intel_classic

This file contains a function that sets the default flags for all Intel classic based compilers in the ToolRepository (ifort, icc).

This function gets called from the default site-specific config file

### Functions

| | |
|---|---|
| *setup_intel_classic*(build_config, args) | Defines the default flags for all Intel classic compilers and linkers. |

fab.fab_base.site_specific.default.setup_intel_classic.**setup_intel_classic**(*build_config*, *args*)

Defines the default flags for all Intel classic compilers and linkers.

**Parameters**

- **build_config** (*BuildConfig*) – the Fab build config instance from which required parameters can be taken.
- **args** (*argparse.Namespace*) – all command line options

**Return type**
Dict[str, List[*AddFlags*]]

### fab.fab_base.site_specific.default.setup_intel_llvm

This file contains a function that sets the default flags for all Intel llvm based compilers and linkers in the ToolRepository (ifx, icx).

This function gets called from the default site-specific config file

### Functions

| | |
|---|---|
| *setup_intel_llvm*(build_config, args) | Defines the default flags for all Intel llvm compilers. |

fab.fab_base.site_specific.default.setup_intel_llvm.**setup_intel_llvm**(*build_config*, *args*)

Defines the default flags for all Intel llvm compilers.

**Parameters**

- **build_config** (*BuildConfig*) – the Fab build config instance from which required parameters can be taken.

- **args** (*argparse.Namespace*) – all command line options

**Return type**
Dict[str, List[*AddFlags*]]

### fab.fab_base.site_specific.default.setup_nvidia

This file contains a function that sets the default flags for the NVIDIA compilers and linkers in the ToolRepository.

This function gets called from the default site-specific config file

#### Functions

| | |
|---|---|
| *setup_nvidia*(build_config, args) | Defines the default flags for nvfortran. |

fab.fab_base.site_specific.default.setup_nvidia.**setup_nvidia**(*build_config*, *args*)

Defines the default flags for nvfortran.

**Parameters**

- **build_config** (*BuildConfig*) – the Fab build config instance from which required parameters can be taken.
- **args** (Namespace) – all command line options

**Return type**
Dict[str, List[*AddFlags*]]

### fab.logtools

Logging tools for the fab framework.

#### Functions

| | |
|---|---|
| *make_logger*(feature[, offset]) | Create a hierarchical logger. |
| *make_loggers*() | Create a set of named loggers. |
| *setup_file_logging*(logfile[, name, create]) | Direct log messages to a named file. |
| *setup_logging*(build_level, system_level[, ...]) | Setup the fab logging framework. |

#### Classes

| | |
|---|---|
| *FabLogFilter*(build_level, system_level[, quiet]) | Class to filter log messages from the console stream. |

fab.logtools.**make_logger**(*feature*, *offset=1*)

Create a hierarchical logger.

The name of the logger is based on the module name of the caller and the calling function, but with the addition of the feature name at the second level of the hierarchy.

If the logger is created in a module, no function name is appended. If the logger is created from the __init__ method of a class, the name of the class is used in place of a function.

**Parameters**

- **feature** (str) – the name of the logging feature. Typically either build or system.

- **offset** (int) – the number of frames to skip. Defaults to 1. (default: 1)

**Returns**
a logging.Logger instance.

fab.logtools.**make_loggers**()

Create a set of named loggers.

A simple convenience function that creates a pair of named loggers with build and system at the second level of the logging hierarchy. This function is ignored when inspecting the calling tree to determine the name to use.

**Returns**
tuple containing two logger instances, the first associated with the build hierarchy and the second with the system hierarchy.

**class** fab.logtools.**FabLogFilter**(*build_level*, *system_level*, *quiet=False*)

Class to filter log messages from the console stream.

This filters out specific types of log message based on level criteria and the boolean quiet flag. This makes it possible to write all log events to a file whilst controlling the amount of output seen by the user.

Initialize a filter.

Initialize with the name of the logger which, together with its children, will have its events allowed through the filter. If no name is specified, allow every event.

**filter**(*record*)

Decide whether to filter a specific log message.

fab.logtools.**setup_logging**(*build_level*, *system_level*, *quiet=False*, *iostream=<_io.TextIOWrapper name='<stderr>' mode='w' encoding='utf-8'>*)

Setup the fab logging framework.

Set output levels for build log messages and system log messages.

Build messages are purely intended to be used to output information about the compile tasks. System messages should help to debug the fab library itself.

**Parameters**

- **build_level** (Optional[int]) – verbosity of output from the build logger hierarchy. Setting to 0 or None implies a low level of output.
- **system_level** (Optional[int]) – verbosity of output from the system logger hierarchy. Setting to 0 or None implies a low level of output.
- **quiet** – do not log anything lower than an error to the user output stream. (default: False)
- **iostream** – output stream used to log messages. Defaults to sys.stderr if not specified. This is particularly useful when testing. (default: <_io.TextIOWrapper name='<stderr>' mode='w' encoding='utf-8'>)

fab.logtools.**setup_file_logging**(*logfile*, *name='root'*, *create=True*)

Direct log messages to a named file.

**Parameters**

- **logfile** (Path) – path to the output log
- **name** – name of the logger hierarchy being added (default: 'root')
- **create** – create the parent directory. Defaults to True. (default: True)

### fab.metrics

A module for recording and summarising metrics, with the following concepts:

**init**
: create pipes create reading process - daemonic, exits with main process

**send**
: group, name, value -> reading process overwrites any previous value for group[name]

**reading process**
: creates and add to metrics dict finishes -> send whole lot down a summary pipe and close

**stop**
: closes pipes & process return metrics from summary pipe

### Functions

| | |
|---|---|
| *init_metrics*(metrics_folder) | Create the pipe for sending metrics, the process to read them, and another pipe to push the final collated data. |
| *metrics_summary*(metrics_folder) | Create various summary charts from the metrics json. |
| *send_metric*(group, name, value) | Pass a metric to the reader process. |
| *stop_metrics*() | Close the metrics pipe and reader process. |

fab.metrics.**init_metrics**(*metrics_folder*)

Create the pipe for sending metrics, the process to read them, and another pipe to push the final collated data.

Only one call to init_metrics can be called before calling stop_metrics.

**Parameters**
**metrics_folder** (Path) – The folder where we will write metrics.

fab.metrics.**send_metric**(*group*, *name*, *value*)

Pass a metric to the reader process.

Metrics will be written to a json file after build steps have run.

Example:

```
send_metric('my step', 'reading took', 123)
send_metric('my step', 'writing took', 456)
```

**Parameters**

- **group** (str) – Name of the metrics group.
- **name** (str) – Name of the metric.
- **value** – Value of the metric.

fab.metrics.**stop_metrics**()

Close the metrics pipe and reader process.

fab.metrics.**metrics_summary**(*metrics_folder*)

Create various summary charts from the metrics json.

### fab.mo

A temporary place for some Met Office specific logic which, for now, needs to be integrated into Fab's internals.

### Functions

| | |
|---|---|
| *add_mo_commented_file_deps*(source_tree) | Handle dependencies from Met Office "DEPENDS ON:" code comments which refer to a c file. |

fab.mo.**add_mo_commented_file_deps**(*source_tree*)

Handle dependencies from Met Office "DEPENDS ON:" code comments which refer to a c file. These are the comments which refer to a .o file and not those which just refer to symbols.

**Parameters**
**source_tree** (Dict[Path, *AnalysedDependent*]) – The source tree of analysed files.

### fab.parse

### Classes

| | |
|---|---|
| *AnalysedFile*(fpath[, file_hash]) | Analysis results for a single file. |
| *EmptySourceFile*(fpath) | An analysis result for a file which resulted in an empty parse tree. |

### Exceptions

| |
|---|
| *ParseException* |

**exception** fab.parse.**ParseException**

**class** fab.parse.**AnalysedFile**(*fpath*, *file_hash=None*)

Analysis results for a single file. Abstract base class.

**Parameters**

- **fpath** (Union[str, Path]) – The path of the file which was analysed.
- **file_hash** (Optional[int]) – The checksum of the file which was analysed. If omitted, Fab will evaluate lazily. (default: None)

If not provided, the *self.file_hash* property is lazily evaluated in case the file does not yet exist.

**to_dict**()

Create a dict representing the object.

The dict may be written to json, so can't contain sets. Lists are sorted for reproducibility in testing.

**Return type**
Dict[str, Any]

**classmethod field_names**()

Defines the order in which we want fields to appear in str or repr strings.

Calling this helps to ensure any lazy attributes are evaluated before use, e.g when constructing a string representation of the instance, or generating a hash value.

class fab.parse.**EmptySourceFile**(*fpath*)

An analysis result for a file which resulted in an empty parse tree.

**Parameters**

**fpath** (Union[str, Path]) – The path of the file which was analysed.

## Modules

| | |
|---|---|
| *c* | C language handling classes. |
| *fortran* | Fortran language handling classes. |
| *fortran_common* | Common functionality for both Fortran and (sanitised) X90 processing. |
| *x90* | |

### fab.parse.c

C language handling classes.

## Classes

| | |
|---|---|
| *AnalysedC*(fpath[, file_hash, symbol_defs, ...]) | An analysis result for a single C file, containing symbol definitions and dependencies. |
| *CAnalyser*(config) | Identify symbol definitions and dependencies in a C file. |

class fab.parse.c.**AnalysedC**(*fpath*, *file_hash=None*, *symbol_defs=None*, *symbol_deps=None*, *file_deps=None*)

An analysis result for a single C file, containing symbol definitions and dependencies.

**Note: We don't need to worry about compile order with pure C projects; we** can compile all in one go. However, with a *Fortran -> C -> Fortran* dependency chain, we do need to ensure that one Fortran file is compiled before another, so this class must be part of the dependency tree analysis.

**Parameters**

- **fpath** (Union[str, Path]) – The source file that was analysed.
- **file_hash** (Optional[int]) – The hash of the source. If omitted, Fab will evaluate lazily. (default: None)
- **symbol_defs** (Optional[Iterable[str]]) – Set of symbol names defined by this source file. (default: None)
- **symbol_deps** (Optional[Iterable[str]]) – Set of symbol names used by this source file. Can include symbols in the same file. (default: None)
- **file_deps** (Optional[Iterable[Path]]) – Other files on which this source depends. Must not include itself. This attribute is calculated during symbol analysis, after everything has been parsed. (default: None)

class fab.parse.c.**CAnalyser**(*config*)

Identify symbol definitions and dependencies in a C file.

### fab.parse.fortran

Fortran language handling classes.

### Classes

| | |
|---|---|
| *AnalysedFortran*(fpath[, file_hash, ...]) | An analysis result for a single file, containing module and symbol definitions and dependencies. |
| *FortranAnalyser*(config[, std, ...]) | A build step which analyses a fortran file using fparser2, creating an `AnalysedFortran`. |
| *FortranParserWorkaround*(fpath[, ...]) | Use this class to create a workaround when the third-party Fortran parser is unable to process a valid source file. |

**class** fab.parse.fortran.**AnalysedFortran**(*fpath*, *file_hash=None*, *program_defs=None*, *module_defs=None*, *symbol_defs=None*, *module_deps=None*, *symbol_deps=None*, *mo_commented_file_deps=None*, *file_deps=None*, *psyclone_kernels=None*)

An analysis result for a single file, containing module and symbol definitions and dependencies.

The user should be unlikely to encounter this class. If the third-party fortran parser is unable to process a source file, a `FortranParserWorkaround` object can be provided to the `Analyse` step, which will be converted at runtime into an instance of this class.

**Parameters**

- **fpath** (`Union[str, Path]`) – The source file that was analysed.
- **file_hash** (`Optional[int]`) – The hash of the source. If omitted, Fab will evaluate lazily. (default: `None`)
- **program_defs** (`Optional[Iterable[str]]`) – Set of program names defined by this source file. (default: `None`)
- **module_defs** (`Optional[Iterable[str]]`) – Set of module names defined by this source file. A subset of symbol_defs (default: `None`)
- **symbol_defs** (`Optional[Iterable[str]]`) – Set of symbol names defined by this source file. (default: `None`)
- **module_deps** (`Optional[Iterable[str]]`) – Set of module names used by this source file. (default: `None`)
- **symbol_deps** (`Optional[Iterable[str]]`) – Set of symbol names used by this source file. Can include symbols in the same file. (default: `None`)
- **mo_commented_file_deps** (`Optional[Iterable[str]]`) – A set of C file names, without paths, on which this file depends. Comes from "DEPENDS ON:" comments which end in ".o". (default: `None`)
- **file_deps** (`Optional[Iterable[Path]]`) – Other files on which this source depends. Must not include itself. This attribute is calculated during symbol analysis, after everything has been parsed. (default: `None`)
- **psyclone_kernels** (`Optional[Dict[str, int]]`) – The hash of any PSyclone kernel metadata found in this source file, by name. (default: `None`)

property mod_filenames

The mod_filenames property defines which module files are expected to be created (but not where).

classmethod field_names()

Defines the order in which we want fields to appear in str or repr strings.

Calling this helps to ensure any lazy attributes are evaluated before use, e.g when constructing a string representation of the instance, or generating a hash value.

to_dict()

Create a dict representing the object.

The dict may be written to json, so can't contain sets. Lists are sorted for reproducibility in testing.

**Return type**
Dict[str, Any]

class fab.parse.fortran.**FortranAnalyser**(*config*, *std=None*, *ignore_dependencies=None*)

A build step which analyses a fortran file using fparser2, creating an AnalysedFortran.

**Parameters**

- **config** (*BuildConfig*) – The BuildConfig to use.
- **std** (Optional[str]) – The Fortran standard. (default: None)
- **ignore_dependencies** (Optional[Iterable[str]]) – Module names to ignore in use statements or 'DEPENDS ON' files to ignore or 'DEPENDS ON' modules to ignore. (default: None)

walk_nodes(*fpath*, *file_hash*, *node_tree*)

Examine the nodes in the parse tree, recording things we're interested in.

Return type depends on our subclass, and will be a subclass of AnalysedDependent.

**Return type**
*AnalysedFortran*

class fab.parse.fortran.**FortranParserWorkaround**(*fpath*, *module_defs=None*, *symbol_defs=None*, *module_deps=None*, *symbol_deps=None*, *mo_commented_file_deps=None*)

Use this class to create a workaround when the third-party Fortran parser is unable to process a valid source file.

**You must manually examine the source file and list:**

- module definitions
- module dependencies
- symbols defined outside a module
- symbols used without a use statement

Params are as for AnalysedFortranBase.

This class is intended to be passed to the Analyse step.

**Parameters**

- **fpath** (Union[str, Path]) – The source file that was analysed.
- **module_defs** (Optional[Iterable[str]]) – Set of module names defined by this source file. A subset of symbol_defs (default: None)

- **symbol_defs** (Optional[Iterable[str]]) – Set of symbol names defined by this source file. (default: None)
- **module_deps** (Optional[Iterable[str]]) – Set of module names used by this source file. (default: None)
- **symbol_deps** (Optional[Iterable[str]]) – Set of symbol names used by this source file. Can include symbols in the same file. (default: None)
- **mo_commented_file_deps** (Optional[Iterable[str]]) – A set of C file names, without paths, on which this file depends. Comes from "DEPENDS ON:" comments which end in ".o". (default: None)

### fab.parse.fortran_common

Common functionality for both Fortran and (sanitised) X90 processing.

### Classes

| | |
|---|---|
| *FortranAnalyserBase*(config, result_class[, std]) | Base class for Fortran parse-tree analysers, e.g FortranAnalyser and X90Analyser. |

**class** fab.parse.fortran_common.**FortranAnalyserBase**(*config*, *result_class*, *std=None*)

Base class for Fortran parse-tree analysers, e.g FortranAnalyser and X90Analyser.

**Parameters**

- **config** (*BuildConfig*) – The BuildConfig object.
- **result_class** – The type (class) of the analysis result. Defined by the subclass.
- **std** (Optional[str]) – The Fortran standard. (default: None)

**property config:** *BuildConfig*

Returns the BuildConfig to use.

**run**(*fpath*)

Parse the source file and record what we're interested in (subclass specific).

Reloads previous analysis results if available.

Returns the analysis data and the result file where it was stored/loaded.

**Return type**
Union[Tuple[*AnalysedDependent*, Path], Tuple[*EmptySourceFile*, None], Tuple[Exception, None]]

**abstract walk_nodes**(*fpath*, *file_hash*, *node_tree*)

Examine the nodes in the parse tree, recording things we're interested in.

Return type depends on our subclass, and will be a subclass of AnalysedDependent.

**Return type**
*AnalysedDependent*

### fab.parse.x90

#### Classes

| | |
|---|---|
| *AnalysedX90*(fpath, file_hash[, kernel_deps]) | Analysis results for an x90 file. |
| *X90Analyser*(config) | |

**class** fab.parse.x90.**AnalysedX90**(*fpath*, *file_hash*, *kernel_deps=None*)

Analysis results for an x90 file.

**Parameters**

- **fpath** (Union[str, Path]) – The path of the x90 file.
- **file_hash** (int) – The checksum of the x90 file.
- **kernel_deps** (Optional[Iterable[str]]) – Kernel symbols used by the x90 file. (default: None)

**to_dict**()

Create a dict representing the object.

The dict may be written to json, so can't contain sets. Lists are sorted for reproducibility in testing.

**Return type**
Dict[str, Any]

**classmethod field_names**()

Defines the order in which we want fields to appear in str or repr strings.

Calling this helps to ensure any lazy attributes are evaluated before use, e.g when constructing a string representation of the instance, or generating a hash value.

**class** fab.parse.x90.**X90Analyser**(*config*)

**Parameters**

- **config** (*BuildConfig*) – The BuildConfig object.
- **result_class** – The type (class) of the analysis result. Defined by the subclass.
- **std** – The Fortran standard.

**walk_nodes**(*fpath*, *file_hash*, *node_tree*)

Examine the nodes in the parse tree, recording things we're interested in.

Return type depends on our subclass, and will be a subclass of AnalysedDependent.

**Return type**
*AnalysedX90*

### fab.steps

Predefined build steps with sensible defaults.

### Functions

| | |
|---|---|
| *check_for_errors*(results[, caller_label]) | Check an iterable of results for any exceptions and handle them gracefully. |
| *run_mp*(config, items, func[, no_multiprocessing]) | Called from Step.run() to process multiple items in parallel. |
| *run_mp_imap*(config, items, func, result_handler) | Like run_mp, but uses imap instead of map so that we can process each result as it happens. |
| *step*(func) | Function decorator for steps. |

fab.steps.**step**(*func*)

Function decorator for steps.

fab.steps.**run_mp**(*config*, *items*, *func*, *no_multiprocessing=False*)

Called from Step.run() to process multiple items in parallel.

For example, a compile step would, in its run() method, find a list of source files in the artefact store. It could then pass those paths to this method, along with a function to compile a *single* file. The whole set of results are returned in a list-like, with undefined order.

**Parameters**

- **items** – An iterable of items to process in parallel.
- **func** – A function to process a single item. Must accept a single argument.
- **no_multiprocessing** (bool) – Overrides the config's multiprocessing flag, disabling multiprocessing for this call. (default: False)

fab.steps.**run_mp_imap**(*config*, *items*, *func*, *result_handler*)

Like run_mp, but uses imap instead of map so that we can process each result as it happens.

This is useful for a slow operation where we want to save our progress as we go instead of waiting for everything to finish, allowing us to pick up where we left off if the program is halted.

**Parameters**

- **items** – An iterable of items to process in parallel.
- **func** – A function to process a single item. Must accept a single argument.
- **result_handler** – A function to handle a single result. Must accept a single argument.

fab.steps.**check_for_errors**(*results*, *caller_label=None*)

Check an iterable of results for any exceptions and handle them gracefully.

This is a helper function for steps which use multiprocessing, getting multiple results back from run_mp() all in one go.

**Parameters**

- **results** (Iterable[Union[str, Exception]]) – An iterable of results.
- **caller_label** (Optional[str]) – Optional human-friendly name of the caller for logging. (default: None)

**Return type**

None

### Modules

| | |
|---|---|
| *analyse* | Fab parses each C and Fortran file into an `AnalysedDependent` object which contains the symbol definitions and dependencies for that file. |
| *archive_objects* | Object archive creation from a list of object files for use in static linking. |
| *c_pragma_injector* | Add custom pragmas to C code which identify user and system include regions. |
| *cleanup_prebuilds* | Pruning of old files from the incremental/prebuild folder. |
| *compile_c* | C file compilation. |
| *compile_fortran* | Fortran file compilation. |
| *find_source_files* | Gather files from a source folder. |
| *grab* | Build steps for pulling source code from remote repos and local folders. |
| *link* | Link an executable. |
| *preprocess* | Fortran and C Preprocessing. |
| *psyclone* | A preprocessor and code generation step for PSyclone. |
| *root_inc_files* | A helper step to copy .inc files to the root of the build source folder, for easy include by the preprocessor. |

### fab.steps.analyse

Fab parses each C and Fortran file into an `AnalysedDependent` object which contains the symbol definitions and dependencies for that file.

From this set of analysed files, Fab builds a symbol table mapping symbols to their containing files.

Fab uses the symbol table to turn symbol dependencies into file dependencies (stored in the AnalysedDependent objects). This gives us a file dependency tree for the entire project source. The data structure is simple, just a dict of *<source path>: <analysed file>*, where the analysed files' dependencies are other dict keys.

If we're building a library, that's the end of the analysis process as we'll compile the entire project source. If we're building one or more executables, which happens when we use the *root_symbol* argument, Fab will extract a subtree from the entire dependency tree for each root symbol we specify.

Finally, the resulting artefact collection is a dict of these subtrees (*"build trees"*), mapping *<root symbol>: <build tree>*. When building a library, there will be a single tree with a root symbol of *None*.

Addendum: The language parsers Fab uses are unable to detect some kinds of dependency. For example, fparser can't currently identify a call statement in a one-line if statement. We can tell Fab that certain symbols *should have been included* in the build tree using the *unreferenced_deps* argument. For every symbol we provide, its source file *and dependencies* will be added to the build trees.

Sometimes a language parser will crash while parsing a *valid* source file, even though the compiler can compile the file perfectly well. In this case we can give Fab the analysis results it should have made by passing FortranParserWorkaround objects into the *special_measure_analysis_results* argument. You'll have to manually read the file to determine which symbol definitions and dependencies it contains.

### Functions

| | |
|---|---|
| *analyse*(config[, source, root_symbol, ...]) | Produce one or more build trees by analysing source code dependencies. |

fab.steps.analyse.**analyse**(*config*, *source=None*, *root_symbol=None*, *find_programs=False*, *std='f2008'*, *special_measure_analysis_results=None*, *unreferenced_deps=None*, *ignore_dependencies=None*)

Produce one or more build trees by analysing source code dependencies.

The resulting artefact collection is a mapping from root symbol to build tree. The name of this artefact collection is taken from `fab.artefacts.ArtefactSet.BUILD_TREES`.

If no artefact getter is specified in *source*, a default is used which provides input files from multiple artefact collections, including the default C and Fortran preprocessor outputs and any source files with a 'little' *.f90* extension.

A build tree is produced for every root symbol specified in *root_symbol*, which can be a string or list of. This is how we create executable files. If no root symbol is specified, a single tree of the entire source is produced (with a root symbol of *None*). This is how we create shared and static libraries.

**Parameters**

- **config** – The *fab.build_config.BuildConfig* object where we can read settings such as the project workspace folder or the multiprocessing flag.
- **source** (Optional[*ArtefactsGetter*]) – An `ArtefactsGetter` to get the source files. (default: `None`)
- **find_programs** (bool) – Instructs the analyser to automatically identify program definitions in the source. Alternatively, the required programs can be specified with the root_symbol argument. (default: `False`)
- **root_symbol** (Union[str, List[str], None]) – When building an executable, provide the Fortran Program name(s), or 'main' for C. If None, build tree extraction will not be performed and the entire source will be used as the build tree - for building a shared or static library. (default: `None`)
- **std** (str) – The fortran standard, passed through to fparser2. Defaults to 'f2008'. (default: `'f2008'`)
- **special_measure_analysis_results** (Optional[Iterable[*FortranParserWorkaround*]]) – When a language parser cannot parse a valid source file, we can manually provide the expected analysis results with this argument. (default: `None`)
- **unreferenced_deps** (Optional[Iterable[str]]) – A list of symbols which are needed for the build, but which cannot be automatically determined by Fab. For example, functions that are called in a one-line if statement. Assuming the files containing these symbols are present and analysed, those files and all their dependencies will be added to the build tree(s). (default: `None`)
- **ignore_dependencies** (Optional[Iterable[str]]) – Third party Fortran module names in USE statements, 'DEPENDS ON' files and modules to be ignored. (default: `None`)
- **name** – Human friendly name for logger output, with sensible default.

### fab.steps.archive_objects

Object archive creation from a list of object files for use in static linking.

### Functions

| | |
|---|---|
| archive_objects(config[, source, ...]) | Create an object archive for every build target, from their object files. |

fab.steps.archive_objects.**archive_objects**(*config*, *source=None*, *output_fpath=None*, *output_collection=ArtefactSet.OBJECT_ARCHIVES*)

Create an object archive for every build target, from their object files.

An object archive is a set of object (*.o*) files bundled into a single file, typically with a *.a* extension.

Expects one or more build targets from its artefact getter, of the form Dict[name, object_files]. By default, it finds the build targets and their object files in the artefact collection named by `fab.constants.COMPILED_FILES`.

This step has three use cases:

- The **object archive** is the end goal of the build.
- The object archive is a convenience step before linking a **shared object**.
- One or more object archives as convenience steps before linking **executables**.

The benefit of creating an object archive before linking is simply to reduce the size of the linker command, which might otherwise include thousands of .o files, making any error output difficult to read. You don't have to use this step before linking. The linker step has a default artefact getter which will work with or without this preceding step.

**Creating a Static or Shared Library:**

When building a library there is expected to be a single build target with a *None* name. This typically happens when configuring the `Analyser` step *without* a root symbol. We can assume the list of object files is the entire project source, compiled.

In this case you must specify an *output_fpath*.

**Creating Executables:**

When creating executables, there is expected to be one or more build targets, each with a name. This typically happens when configuring the `Analyser` step *with* a root symbol(s). We can assume each list of object files is sufficient to build each <*root_symbol*> executable.

In this case you cannot specify an *output_fpath* path because they are automatically created from the target name.

**Parameters**

- **config** (*BuildConfig*) – The *fab.build_config.BuildConfig* object where we can read settings such as the project workspace folder or the multiprocessing flag.
- **source** (Optional[*ArtefactsGetter*]) – An *ArtefactsGetter* which give us our lists of objects to archive. The artefacts are expected to be of the form *Dict[root_symbol_name, list_of_object_files]*. (default: `None`)
- **output_fpath** (Optional[Path]) – The file path of the archive file to create. This string can include templating, where "$output" is replaced with the output folder.
  - Must be specified when building a library file (no build target name).
  - Must not be specified when building linker input (one or more build target names). (default: `None`)
- **output_collection** – The name of the artefact collection to create. Defaults to the name in `fab.artefacts.ArtefactSet.OBJECT_ARCHIVES`. (default: `<ArtefactSet.OBJECT_ARCHIVES: 11>`)

### fab.steps.c_pragma_injector

Add custom pragmas to C code which identify user and system include regions.

#### Functions

| | |
|---|---|
| *c_pragma_injector*(config[, source, output_name]) | A build step to inject custom pragmas to mark blocks of user and system include statements. |
| *inject_pragmas*(fpath) | Reads a C source file but when encountering an #include preprocessor directive injects a special Fab-specific #pragma which can be picked up later by the Analyser after the preprocessing |

fab.steps.c_pragma_injector.**c_pragma_injector**(*config*, *source=None*, *output_name=None*)

A build step to inject custom pragmas to mark blocks of user and system include statements.

By default, reads .c files from the *INITIAL_SOURCE* artefact and creates the *pragmad_c* artefact.

This step does not write to the build output folder, it creates the pragmad c in the same folder as the c file. This is because a subsequent preprocessing step needs to look in the source folder for header files, including in paths relative to the c file.

**Parameters**

- **config** – The *fab.build_config.BuildConfig* object where we can read settings such as the project workspace folder or the multiprocessing flag.
- **source** (Optional[*ArtefactsGetter*]) – An *ArtefactsGetter* which give us our c files to process. (default: None)
- **output_name** – The name of the artefact collection to create in the artefact store, with a sensible default (default: None)

fab.steps.c_pragma_injector.**inject_pragmas**(*fpath*)

Reads a C source file but when encountering an #include preprocessor directive injects a special Fab-specific #pragma which can be picked up later by the Analyser after the preprocessing

**Return type**
Generator

### fab.steps.cleanup_prebuilds

Pruning of old files from the incremental/prebuild folder.

#### Functions

| | |
|---|---|
| by_age(older_than, prebuilds_ts, current_files) | |
| by_version_age(n_versions, prebuilds_ts, ...) | |
| *cleanup_prebuilds*(config[, older_than, ...]) | A step to delete old files from the local incremental/prebuild folder. |
| *get_access_time*(fpath) | Return the access time of the given file. |
| remove_all_unused(found_files, current_files) | |

fab.steps.cleanup_prebuilds.**cleanup_prebuilds**(*config*, *older_than=None*, *n_versions=0*, *all_unused=None*)

A step to delete old files from the local incremental/prebuild folder.

Assumes prebuild filenames follow the pattern: *<stem>.<hash>.<suffix>*.

**Parameters**

- **config** – The *fab.build_config.BuildConfig* object where we can read settings such as the project workspace folder or the multiprocessing flag.
- **older_than** (Optional[timedelta]) – Delete prebuild artefacts which are *n seconds* older than the *last prebuild access time*. (default: None)
- **n_versions** (int) – Only keep the most recent n versions of each artefact *<stem>.*.<suffix>* (default: 0)
- **all_unused** (Optional[bool]) – Delete everything which was not part of the current build. (default: None)

If no parameters are specified then *all_unused* will default to *True*.

fab.steps.cleanup_prebuilds.**get_access_time**(*fpath*)

Return the access time of the given file.

Depends on the file system's ability to report a file's last access time via the *os.stat_result.st_atime* returned by *Path.stat*.

**Return type**
datetime

### fab.steps.compile_c

C file compilation.

### Functions

| | |
|---|---|
| *compile_c*(config[, common_flags, ...]) | Compiles all C files in all build trees, creating or extending a set of compiled files for each target. |
| *store_artefacts*(compiled_files, build_lists, ...) | Create our artefact collection; object files for each compiled file, per root symbol. |

### Classes

| | |
|---|---|
| *MpCommonArgs*(config, flags) | A simple class to pass arguments to subprocesses. |

**class** fab.steps.compile_c.**MpCommonArgs**(*config*, *flags*)

A simple class to pass arguments to subprocesses.

fab.steps.compile_c.**compile_c**(*config*, *common_flags=None*, *path_flags=None*, *source=None*)

Compiles all C files in all build trees, creating or extending a set of compiled files for each target.

This step uses multiprocessing. All C files are compiled in a single pass.

Uses multiprocessing, unless disabled in the *config*.

**Parameters**

- **config** – The *fab.build_config.BuildConfig* object where we can read settings such as the project workspace folder or the multiprocessing flag.
- **common_flags** (Optional[List[str]]) – A list of strings to be included in the command line call, for all files. (default: None)
- **path_flags** (Optional[List]) – A list of *AddFlags*, defining flags to be included in the command line call for selected files. (default: None)
- **source** (Optional[*ArtefactsGetter*]) – An *ArtefactsGetter* which give us our c files to process. (default: None)

fab.steps.compile_c.**store_artefacts**(*compiled_files*, *build_lists*, *artefact_store*)

Create our artefact collection; object files for each compiled file, per root symbol.

### fab.steps.compile_fortran

Fortran file compilation.

### Functions

| | |
|---|---|
| *compile_file*(analysed_file, flags, ...) | Call the compiler. |
| *compile_fortran*(config[, common_flags, ...]) | Compiles all Fortran files in all build trees, creating/extending a set of compiled files for each build target. |
| compile_pass(config, compiled, uncompiled, ...) | |
| *get_compile_next*(compiled, uncompiled) | Find what to compile next. |
| *get_mod_hashes*(analysed_files, config) | Get the hash of every module file defined in the list of analysed files. |
| handle_compiler_args(config[, common_flags, ...]) | |
| *process_file*(arg) | Prepare to compile a fortran file, and compile it if anything has changed since it was last compiled. |
| *store_artefacts*(compiled_files, build_lists, ...) | Create our artefact collection; object files for each compiled file, per root symbol. |

### Classes

| | |
|---|---|
| *MpCommonArgs*(config, flags, mod_hashes, ...) | Arguments to be passed into the multiprocessing function, alongside the filenames. |

**class** fab.steps.compile_fortran.**MpCommonArgs**(*config*, *flags*, *mod_hashes*, *syntax_only*)

Arguments to be passed into the multiprocessing function, alongside the filenames.

fab.steps.compile_fortran.**compile_fortran**(*config*, *common_flags=None*, *path_flags=None*, *source=None*)

Compiles all Fortran files in all build trees, creating/extending a set of compiled files for each build target.

Files are compiled in multiple passes, with each pass enabling further files to be compiled in the next pass.

Uses multiprocessing, unless disabled in the config.

**Parameters**

- **config** (*BuildConfig*) – The *fab.build_config.BuildConfig* object where we can read settings such as the project workspace folder or the multiprocessing flag.

- **common_flags** (Optional[List[str]]) – A list of strings to be included in the command line call, for all files. (default: None)
- **path_flags** (Optional[List]) – A list of *AddFlags*, defining flags to be included in the command line call for selected files. (default: None)
- **source** (Optional[*ArtefactsGetter*]) – An *ArtefactsGetter* which gives us our Fortran files to process. (default: None)

fab.steps.compile_fortran.**get_compile_next**(*compiled*, *uncompiled*)

Find what to compile next. :type compiled: Dict[Path, *CompiledFile*] :param compiled: A dictionary with already compiled files. :type uncompiled: Set[*AnalysedFortran*] :param uncompiled: The set of still to be compiled files. :rtype: Set[*AnalysedFortran*] :returns: A set with all files that can now be compiled.

fab.steps.compile_fortran.**store_artefacts**(*compiled_files*, *build_lists*, *artefact_store*)

Create our artefact collection; object files for each compiled file, per root symbol.

fab.steps.compile_fortran.**process_file**(*arg*)

Prepare to compile a fortran file, and compile it if anything has changed since it was last compiled.

Object files are created directly as artefacts in the prebuild folder. Mod files are created in the module folder and copied as artefacts into the prebuild folder. If nothing has changed, prebuilt mod files are copied *from* the prebuild folder into the module folder. :rtype: Union[Tuple[*CompiledFile*, List[Path]], Tuple[Exception, None]]

> **Note**
>
> Prebuild filenames include a "combo-hash" of everything that, if changed, must trigger a recompile. For mod and object files, this includes a checksum of: *source code, compiler*. For object files, this also includes a checksum of: *compiler flags, modules on which we depend*.
>
> Before compiling a file, we calculate the combo hashes and see if the output files already exists.

Returns a compilation result, regardless of whether it was compiled or prebuilt.

fab.steps.compile_fortran.**compile_file**(*analysed_file*, *flags*, *output_fpath*, *mp_common_args*)

Call the compiler.

The current working folder for the command is set to the folder where the source file lives when compile_file is called. This is done to stop the compiler inserting folder information into the mod files, which would cause them to have different checksums depending on where they live.

fab.steps.compile_fortran.**get_mod_hashes**(*analysed_files*, *config*)

Get the hash of every module file defined in the list of analysed files.

**Return type**
Dict[str, int]

### fab.steps.find_source_files

Gather files from a source folder.

### Functions

| | |
|---|---|
| *find_source_files*(config[, source_root, ...]) | Find the files in the source folder, with filtering. |

### Classes

| | |
|---|---|
| *Exclude*(*filter_strings) | A path filter which excludes matching paths, this convenience class improves config readability. |
| *Include*(*filter_strings) | A path filter which includes matching paths, this convenience class improves config readability. |

**class** fab.steps.find_source_files.**Include**(**filter_strings*)

A path filter which includes matching paths, this convenience class improves config readability.

**Parameters**
**filter_strings** – One or more strings to be used as pattern matches.

**class** fab.steps.find_source_files.**Exclude**(**filter_strings*)

A path filter which excludes matching paths, this convenience class improves config readability.

**Parameters**
**filter_strings** – One or more strings to be used as pattern matches.

fab.steps.find_source_files.**find_source_files**(*config*, *source_root=None*, *output_collection=ArtefactSet.INITIAL_SOURCE*, *path_filters=None*)

Find the files in the source folder, with filtering.

Files can be included or excluded with simple pattern matching. Every file is included by default, unless the filters say otherwise.

Path filters are expected to be provided by the user in an *ordered* collection. The two convenience subclasses, `Include` and `Exclude`, improve readability.

Order matters. For example:

```
path_filters = [
    Exclude('my_folder'),
    Include('my_folder/my_file.F90'),
]
```

In the above example, swapping the order would stop the file being included in the build.

A path matches a filter string simply if it *contains* it, so the path *my_folder/my_file.F90* would match filters "my_folder", "my_file" and "er/my".

**Parameters**

- **config** – The *fab.build_config.BuildConfig* object where we can read settings such as the project workspace folder or the multiprocessing flag.
- **source_root** – Optional path to source folder, with a sensible default. (default: `None`)
- **output_collection** – Name of artefact collection to create, with a sensible default. (default: `<ArtefactSet.INITIAL_SOURCE: 1>`)
- **path_filters** (`Optional[Iterable[_PathFilter]]`) – Iterable of Include and/or Exclude objects, to be processed in order. (default: `None`)
- **name** – Human friendly name for logger output, with sensible default.

### fab.steps.grab

Build steps for pulling source code from remote repos and local folders.

### Modules

| | |
|---|---|
| *archive* | |
| *fcm* | This file contains the various fcm steps. |
| *folder* | |
| *git* | This module contains the git related steps. |
| *prebuild* | |
| *svn* | This file contains the steps related to SVN. |

### fab.steps.grab.archive

### Functions

| | |
|---|---|
| *grab_archive*(config, src[, dst_label]) | Copy source from an archive into the project folder. |

fab.steps.grab.archive.**grab_archive**(*config*, *src*, *dst_label=''*)

Copy source from an archive into the project folder.

**Parameters**

- **src** (Union[Path, str]) – The source archive to grab from.
- **dst_label** (str) – The name of a sub folder, in the project workspace, in which to put the source. If not specified, the code is copied into the root of the source folder. (default: '')
- **name** – Human friendly name for logger output, with sensible default.

### fab.steps.grab.fcm

This file contains the various fcm steps. They are not decorated with @steps since all functions here just call the corresponding svn steps.

### Functions

| | |
|---|---|
| *fcm_checkout*(config, src[, dst_label, revision]) | Params as per *svn_checkout()*. |
| *fcm_export*(config, src[, dst_label, revision]) | Params as per *svn_export()*. |
| *fcm_merge*(config, src[, dst_label, revision]) | Params as per *svn_merge()*. |

fab.steps.grab.fcm.**fcm_export**(*config*, *src*, *dst_label=None*, *revision=None*)

Params as per *svn_export()*.

fab.steps.grab.fcm.**fcm_checkout**(*config*, *src*, *dst_label=None*, *revision=None*)

Params as per *svn_checkout()*.

fab.steps.grab.fcm.**fcm_merge**(*config*, *src*, *dst_label=None*, *revision=None*)

Params as per *svn_merge()*.

### fab.steps.grab.folder

#### Functions

| | |
|---|---|
| *grab_folder*(config, src[, dst_label]) | Copy a source folder to the project workspace. |

fab.steps.grab.folder.**grab_folder**(*config*, *src*, *dst_label=''*)

Copy a source folder to the project workspace.

**Parameters**

- **config** – The *fab.build_config.BuildConfig* object where we can read settings such as the project workspace folder or the multiprocessing flag.
- **src** (Union[Path, str]) – The source location to grab. The nature of this parameter is depends on the subclass.
- **dst_label** (str) – The name of a sub folder, in the project workspace, in which to put the source. If not specified, the code is copied into the root of the source folder. (default: '')

### fab.steps.grab.git

This module contains the git related steps.

#### Functions

| | |
|---|---|
| *git_checkout*(config, src[, dst_label, revision]) | Checkout or update a Git repo. |
| *git_merge*(config, src[, dst_label, revision]) | Merge a git repo into a local working copy. |

fab.steps.grab.git.**git_checkout**(*config*, *src*, *dst_label=''*, *revision=None*)

Checkout or update a Git repo.

fab.steps.grab.git.**git_merge**(*config*, *src*, *dst_label=''*, *revision=None*)

Merge a git repo into a local working copy.

### fab.steps.grab.prebuild

#### Functions

| | |
|---|---|
| *grab_pre_build*(config, path[, allow_fail]) | Copy the contents of another project's prebuild folder into our local prebuild folder. |

fab.steps.grab.prebuild.**grab_pre_build**(*config*, *path*, *allow_fail=False*)

Copy the contents of another project's prebuild folder into our local prebuild folder.

### fab.steps.grab.svn

This file contains the steps related to SVN. It is also used by the various fcm steps, which call the functions here with just a different category (FCM) from the tool box.

### Functions

| | |
|---|---|
| check_conflict(tool, dst) | Check if there's a conflict |
| svn_checkout(config, src[, dst_label, ...]) | Checkout or update an FCM repo. |
| svn_export(config, src[, dst_label, ...]) | Export an FCM repo folder to the project workspace. |
| svn_merge(config, src[, dst_label, ...]) | Merge an FCM repo into a local working copy. |

fab.steps.grab.svn.**svn_export**(*config*, *src*, *dst_label=None*, *revision=None*, *category=Category.SUBVERSION*)

Export an FCM repo folder to the project workspace.

fab.steps.grab.svn.**svn_checkout**(*config*, *src*, *dst_label=None*, *revision=None*, *category=Category.SUBVERSION*)

Checkout or update an FCM repo.

> **Note**
>
> If the destination is a working copy, it will be updated to the given revision, **ignoring the source url**. As such, the revision should be provided via the argument, not as part of the url.

fab.steps.grab.svn.**svn_merge**(*config*, *src*, *dst_label=None*, *revision=None*, *category=Category.SUBVERSION*)

Merge an FCM repo into a local working copy.

fab.steps.grab.svn.**check_conflict**(*tool*, *dst*)

Check if there's a conflict

## fab.steps.link

Link an executable.

### Functions

| | |
|---|---|
| link_exe(config[, libs, flags, source]) | Link object files into an executable for every build target. |
| link_shared_object(config, output_fpath[, ...]) | Produce a shared object (*.so*) file from the given build target. |

### Classes

| | |
|---|---|
| DefaultLinkerSource() | A source getter specifically for linking. |

**class** fab.steps.link.**DefaultLinkerSource**

A source getter specifically for linking. Looks for the default output from archiving objects, falls back to default compiler output. This allows a link step to work with or without a preceding object archive step.

fab.steps.link.**link_exe**(*config*, *libs=None*, *flags=None*, *source=None*)

Link object files into an executable for every build target.

Expects one or more build targets from its artefact getter, of the form Dict[name, object_files].

The default artefact getter, `DefaultLinkerSource`, looks for any output from an `ArchiveObjects` step, and falls back to using output from compiler steps.

**Parameters**

- **config** – The *fab.build_config.BuildConfig* object where we can read settings such as the project workspace folder or the multiprocessing flag.
- **libs** (Optional[List[str]]) – A list of required library names to pass to the linker. (default: None)
- **flags** (Optional[List[str]]) – A list of additional flags to pass to the linker. (default: None)
- **source** (Optional[*ArtefactsGetter*]) – An optional *ArtefactsGetter*. It defaults to the output from compiler steps, which typically is the expected behaviour. (default: None)

fab.steps.link.**link_shared_object**(*config*, *output_fpath*, *flags=None*, *source=None*)

Produce a shared object (*.so*) file from the given build target.

Expects a *single build target* from its artefact getter, of the form Dict[None, object_files]. We can assume the list of object files is the entire project source, compiled.

Params are as for `LinkerBase`, with the addition of:

**Parameters**

- **config** – The *fab.build_config.BuildConfig* object where we can read settings such as the project workspace folder or the multiprocessing flag.
- **output_fpath** (str) – File path of the shared object to create.
- **flags** (Optional[List[str]]) – A list of flags to pass to the linker. (default: None)
- **source** (Optional[*ArtefactsGetter*]) – An optional *ArtefactsGetter*. Typically not required, as there is a sensible default. (default: None)

### fab.steps.preprocess

Fortran and C Preprocessing.

### Functions

| | |
|---|---|
| *pre_processor*(config, preprocessor, files, ...) | Preprocess Fortran or C files. |
| *preprocess_c*(config[, source]) | Wrapper to pre_processor for C files. |
| *preprocess_fortran*(config[, source]) | Wrapper to pre_processor for Fortran files. |
| *process_artefact*(arg) | Expects an input file in the source folder. |

### Classes

| | |
|---|---|
| *DefaultCPreprocessorSource*() | A source getter specifically for c preprocessing. |
| *MpCommonArgs*(config, output_suffix, ...) | Common args for calling process_artefact() using multiprocessing. |

**class** fab.steps.preprocess.**MpCommonArgs**(*config*, *output_suffix*, *preprocessor*, *flags*, *name*)

Common args for calling process_artefact() using multiprocessing.

fab.steps.preprocess.**pre_processor**(*config*, *preprocessor*, *files*, *output_collection*, *output_suffix*, *common_flags=None*, *path_flags=None*, *name='preprocess'*)

Preprocess Fortran or C files.

Uses multiprocessing, unless disabled in the config.

**Parameters**

- **config** (*BuildConfig*) – The *fab.build_config.BuildConfig* object where we can read settings such as the project workspace folder or the multiprocessing flag.
- **preprocessor** (*Preprocessor*) – The preprocessor executable.
- **files** (Collection[Path]) – The files to preprocess.
- **output_collection** (Union[str, *ArtefactSet*]) – The name of the output artefact collection.
- **output_suffix** – Suffix for output files.
- **common_flags** (Optional[List[str]]) – Used to construct a FlagsConfig object. (default: None)
- **path_flags** (Optional[List]) – Used to construct a *FlagsConfig* object. (default: None)
- **name** – Human friendly name for logger output, with sensible default. (default: 'preprocess')

fab.steps.preprocess.**process_artefact**(*arg*)

Expects an input file in the source folder. Writes the output file to the output folder, with a lower case extension.

fab.steps.preprocess.**preprocess_fortran**(*config*, *source=None*, ***kwargs*)

Wrapper to pre_processor for Fortran files.

Ensures all preprocessed files are in the build output. This means *copying* already preprocessed files from source to build output.

Params as per `_pre_processor()`.

If source is not provided, it defaults to *SuffixFilter(ArtefactStore.FORTRAN_BUILD_FILES, '.F90')*.

**class** fab.steps.preprocess.**DefaultCPreprocessorSource**

A source getter specifically for c preprocessing. Looks for the default output from pragma injection, falls back to default source finder. This allows the step to work with or without a preceding pragma step.

fab.steps.preprocess.**preprocess_c**(*config*, *source=None*, ***kwargs*)

Wrapper to pre_processor for C files.

Params as per `_pre_processor()`. If source is not provided, it defaults to *DefaultCPreprocessorSource*.

### fab.steps.psyclone

A preprocessor and code generation step for PSyclone. https://github.com/stfc/PSyclone

### Functions

| | |
|---|---|
| do_one_file(arg) | |
| *make_parsable_x90*(x90_path) | Take out the leading name keyword in calls to invoke(), making temporary, parsable fortran from x90s. |
| preprocess_x90(config[, common_flags]) | |
| *psyclone*(config[, kernel_roots, ...]) | Psyclone runner step. |

### Classes

| | |
|---|---|
| *MpCommonArgs*(config, analysed_x90, ...) | Runtime data for child processes to read. |

**class** fab.steps.psyclone.**MpCommonArgs**(*config*, *analysed_x90*, *kernel_roots*, *transformation_script*, *cli_args*, *api*, *all_kernel_hashes*, *overrides_folder*, *override_files*)

Runtime data for child processes to read.

Contains data used to calculate the prebuild hash.

fab.steps.psyclone.**psyclone**(*config*, *kernel_roots=None*, *transformation_script=None*, *cli_args=None*, *source_getter=None*, *overrides_folder=None*, *api=None*)

Psyclone runner step.

> **Note**
>
> This step produces Fortran, so it must be run before the `Analyse` step.

This step stores prebuilt results to speed up subsequent builds. To generate the prebuild hashes, it analyses the X90 and kernel files, storing prebuilt results for these also.

Kernel files are just normal Fortran, and the standard Fortran analyser is used to analyse them

**Parameters**

- **config** – The *fab.build_config.BuildConfig* object where we can read settings such as the project workspace folder or the multiprocessing flag.
- **kernel_roots** (Optional[List[Path]]) – Folders containing kernel files. Must be part of the analysed source code. (default: None)
- **transformation_script** (Optional[Callable[[Path, *BuildConfig*], Path]]) – The function to get Python transformation script. It takes in a file path and the config object, and returns the path of the transformation script or None. If no function is given or the function returns None, no script will be applied and PSyclone still runs. (default: None)
- **cli_args** (Optional[List[str]]) – Passed through to the psyclone cli tool. (default: None)
- **source_getter** (Optional[*ArtefactsGetter*]) – Optional override for getting input files from the artefact store. (default: None)
- **overrides_folder** (Optional[Path]) – Optional folder containing hand-crafted override files. Must be part of the subsequently analysed source code. Any file produced by psyclone will be deleted if there is a corresponding file in this folder. (default: None)

fab.steps.psyclone.**make_parsable_x90**(*x90_path*)

Take out the leading name keyword in calls to invoke(), making temporary, parsable fortran from x90s.

If present it looks like this:

```
call invoke( name = "compute_dry_mass", ...
```

Returns the path of the parsable file.

This function is not slow so we're not creating prebuilds for this work.

**Return type**
Path

### fab.steps.root_inc_files

A helper step to copy .inc files to the root of the build source folder, for easy include by the preprocessor.

Currently only used for building JULES, .inc files are due to be removed from dev practices, at which point this step should be deprecated.

### Functions

| | |
|---|---|
| *root_inc_files*(config) | Copy inc files into the workspace output root. |

fab.steps.root_inc_files.**root_inc_files**(*config*)

Copy inc files into the workspace output root.

Checks for name clash. This step does not create any artefacts. It's up to the user to configure other tools to find these files.

**Parameters**

- **artefact_store** – Artefacts created by previous Steps. This is where we find the artefacts to process.
- **config** (*BuildConfig*) – The *fab.build_config.BuildConfig* object where we can read settings such as the project workspace folder or the multiprocessing flag.

### fab.tools

A simple init file to make it shorter to import tools.

**class** fab.tools.**Ar**

This is the base class for *ar*.

**create**(*output_fpath*, *members*)

Create the archive with the specified name, containing the listed members.

**Parameters**

- **output_fpath** (Path) – the output path.
- **members** (List[Union[str, Path]]) – the list of objects to be added to the archive.

**class** fab.tools.**Category**(*value*)

This class defines the allowed tool categories.

property is_compiler

Returns if the category is either a C or a Fortran compiler.

class fab.tools.CCompiler(*name*, *exec_name*, *suite*, *version_regex*, *mpi=False*, *compile_flag=None*, *output_flag=None*, *openmp_flag=None*, *version_argument=None*, *availability_option=None*)

This is the base class for a C compiler. It just sets the category of the compiler as convenience.

**Parameters**

- **name** (str) – name of the compiler.
- **exec_name** (str) – name of the executable to start.
- **suite** (str) – name of the compiler suite.
- **version_regex** (str) – A regular expression that allows extraction of the version number from the version output of the compiler.
- **mpi** (bool) – whether the compiler or linker support MPI. (default: False)
- **compile_flag** (Optional[str]) – the compilation flag to use when only requesting compilation (not linking). (default: None)
- **output_flag** (Optional[str]) – the compilation flag to use to indicate the name of the output file (default: None)
- **openmp_flag** (Optional[str]) – the flag to use to enable OpenMP (default: None)

class fab.tools.Compiler(*name*, *exec_name*, *suite*, *version_regex*, *category*, *mpi=False*, *compile_flag=None*, *output_flag=None*, *openmp_flag=None*, *version_argument=None*, *availability_option=None*)

This is the base class for any compiler. It provides flags for

- compilation only (-c),
- naming the output file (-o),
- OpenMP

**Parameters**

- **name** (str) – name of the compiler.
- **exec_name** (Union[str, Path]) – name of the executable to start.
- **suite** (str) – name of the compiler suite this tool belongs to.
- **version_regex** (str) – A regular expression that allows extraction of the version number from the version output of the compiler. The version is taken from the first group of a match.
- **category** (*Category*) – the Category (C_COMPILER or FORTRAN_COMPILER).
- **mpi** (bool) – whether the compiler or linker support MPI. (default: False)
- **compile_flag** (Optional[str]) – the compilation flag to use when only requesting compilation (not linking). (default: None)
- **output_flag** (Optional[str]) – the compilation flag to use to indicate the name of the output file (default: None)
- **openmp_flag** (Optional[str]) – the flag to use to enable OpenMP. If no flag is specified, it is assumed that the compiler does not support OpenMP. (default: None)

- **availability_option** (Union[str, List[str], None]) – a command line option for the tool to test if the tool is available on the current system. Defaults to *–version*. (default: None)

property mpi: bool

**Returns**
whether this compiler supports MPI or not.

property openmp: bool

**Returns**
compiler's OpenMP support.

property openmp_flag: str

**Returns**
compiler argument to enable OpenMP.

property compile_flag: str

**Returns**
the flag to indicate compilation only (not linking).

property output_flag: str

**Returns**
compiler argument for output file.

get_hash(*profile=None*)

**Return type**
int

**Returns**
hash of compiler name and version.

get_flags(*profile=None*)

Determines the flags to be used.

**Return type**
List[str]

**Returns**
the flags to be used with this tool.

get_all_commandline_options(*config*, *input_file*, *output_file*, *add_flags=None*)

This function returns all command line options for a compiler (but not the executable name). It is used by a compiler wrapper to pass the right flags to the wrapper. This base implementation adds the input and output filename (including the -o flag), the flag to only compile (and not link), and if required openmp.

**Parameters**

- **input_file** (Path) – the name of the input file.
- **output_file** (Path) – the name of the output file.
- **config** (*BuildConfig*) – The BuildConfig, from which compiler profile and OpenMP status are taken.
- **add_flags** (Optional[List[str]]) – additional flags for the compiler. (default: None)

**Return type**
List[str]

**Returns**
all command line options for compilation.

**compile_file**(*input_file*, *output_file*, *config*, *add_flags=None*)

Compiles a file. It will add the flag for compilation-only automatically, as well as the output directives. The current working directory for the command is set to the folder where the source file lives when compile_file is called. This is done to stop the compiler inserting folder information into the mod files, which would cause them to have different checksums depending on where they live.

**Parameters**

- **input_file** (Path) – the path of the input file.
- **output_file** (Path) – the path of the output file.
- **config** (*BuildConfig*) – The BuildConfig, from which compiler profile and OpenMP status are taken.
- **add_flags** (Optional[List[str]]) – additional compiler flags. (default: None)

**check_available**()

Checks if the compiler is available. While the method in the Tools base class would be sufficient (when using –version), in case of a compiler we also want to store the compiler version. So, re-implement check_available in a way that will automatically store the compiler version for later usage.

**Return type**
bool

**Returns**
whether the compiler is available or not. We do this by requesting the compiler version.

**get_version**()

Try to get the version of the given compiler.

Expects a version in a certain part of the –version output, which must adhere to the n.n.n format, with at least 2 parts.

**Return type**
Tuple[int, ...]

**Returns**
a tuple of at least 2 integers, representing the version e.g. (6, 10, 1) for version '6.10.1'.

**Raises**
RuntimeError – if the compiler was not found, or if it returned an unrecognised output from the version command.

**run_version_command**(*version_command='--version'*)

Run the compiler's command to get its version.

**Parameters**
**version_command** (Optional[str]) – The compiler argument used to get version info. (default: '--version')

**Return type**
str

**Returns**
The output from the version command.

**Raises**
RuntimeError – if the compiler was not found, or raised an error.

**get_version_string**()

Get a string representing the version of the given compiler.

**Return type**
str

**Returns**
a string of at least 2 numeric version components, i.e. major.minor[.patch, ...]

**Raises**
**RuntimeError** – if the compiler was not found, or if it returned an unrecognised output from the version command.

class fab.tools.**CompilerSuiteTool**(*name*, *exec_name*, *suite*, *category*, *availability_option=None*)

A tool that is part of a compiler suite (typically compiler and linker).

**Parameters**

- **name** (str) – name of the tool.
- **exec_name** (Union[str, Path]) – name of the executable to start.
- **suite** (str) – name of the compiler suite.
- **category** (*Category*) – the Category to which this tool belongs.
- **availability_option** (Union[str, List[str], None]) – a command line option for the tool to test if the tool is available on the current system. Defaults to *–version*. (default: None)

**property suite:** str

**Returns**
the compiler suite of this tool.

class fab.tools.**CompilerWrapper**(*name*, *exec_name*, *compiler*, *mpi=False*)

A decorator-based compiler wrapper. It basically uses a different executable name when compiling, but otherwise behaves like the wrapped compiler. An example of a compiler wrapper is *mpif90* (which can internally call e.g. gfortran, icc, ...)

**Parameters**

- **name** (str) – name of the wrapper.
- **exec_name** (str) – name of the executable to call.
- **compiler** (*Compiler*) – the compiler that is decorated.
- **mpi** (bool) – whether MPI is supported by this compiler or not. (default: False)

**property compiler:** *Compiler*

**Returns**
the compiler that is wrapped by this CompilerWrapper.

**property suite:** str

**Returns**
the compiler suite of this tool.

**property openmp_flag:** str

Returns the flag to enable OpenMP.

property has_syntax_only: bool

**Returns**
whether this compiler supports a syntax-only feature.

**Raises**
RuntimeError – if this function is called for a non-Fortran wrapped compiler.

get_flags(*profile=None*)

**Return type**
List[str]

**Returns**
the ProfileFlags for the given profile, combined from the wrapped compiler and this wrapper.

**Parameters**
**profile** (Optional[str]) – the profile to use. (default: None)

set_module_output_path(*path*)

Sets the output path for modules.

**Params path**
the path to the output directory.

**Raises**
RuntimeError – if this function is called for a non-Fortran wrapped compiler.

get_all_commandline_options(*config*, *input_file*, *output_file*, *add_flags=None*, *syntax_only=False*)

This function returns all command line options for a compiler wrapper. The syntax_only flag is only accepted, if the wrapped compiler is a Fortran compiler. Otherwise, an exception will be raised.

**Parameters**

- **input_file** (Path) – the name of the input file.
- **output_file** (Path) – the name of the output file.
- **config** (*BuildConfig*) – The BuildConfig, from which compiler profile and OpenMP status are taken.
- **add_flags** (Optional[List[str]]) – additional flags for the compiler. (default: None)
- **syntax_only** (Optional[bool]) – if set, the compiler will only do a syntax check (default: False)

**Return type**
List[str]

**Returns**
command line flags for compiler wrapper.

**Raises**
RuntimeError – if syntax_only is requested for a non-Fortran compiler.

compile_file(*input_file*, *output_file*, *config*, *add_flags=None*, *syntax_only=None*)

Compiles a file using the wrapper compiler.

**Parameters**

- **input_file** (Path) – the name of the input file.
- **output_file** (Path) – the name of the output file.

- **config** (*BuildConfig*) – The BuildConfig, from which compiler profile and OpenMP status are taken.
- **add_flags** (Optional[List[str]]) – additional flags for the compiler. (default: None)
- **syntax_only** (Optional[bool]) – if set, the compiler will only do a syntax check (default: None)

class fab.tools.**Cpp**

Class for cpp.

class fab.tools.**CppFortran**

Class for cpp when used as a Fortran preprocessor

class fab.tools.**Craycc**(*name='craycc-cc'*, *exec_name='cc'*)

Class for the native Cray C compiler. Since cc is actually a compiler wrapper, follow the naming scheme of a compiler wrapper and call it: craycc-cc.

**Cray has two different compilers. Older ones have as version number:**
Cray C : Version 8.7.0 Tue Jul 23, 2024 07:39:46

**Newer compiler (several lines, the important one):**
Cray clang version 15.0.1 (66f7391d6a03cf932f321b9f6b1d8612ef5f362c)

We use the beginning ("cray c") to identify the compiler, which works for both cray c and cray clang. Then we ignore non-numbers, to reach the version number which is then extracted.

**Parameters**

- **name** (str) – name of this compiler. (default: 'craycc-cc')
- **exec_name** (str) – name of the executable. (default: 'cc')

class fab.tools.**CrayCcWrapper**(*compiler*)

Class for the Cray C compiler wrapper. We add 'wrapper' to the class name to make this class distinct from the Craycc compiler class

**Parameters**
**compiler** (*Compiler*) – the compiler that the mpicc wrapper will use.

class fab.tools.**Crayftn**(*name='crayftn-ftn'*, *exec_name='ftn'*)

Class for the native Cray Fortran compiler. Since ftn is actually a compiler wrapper, follow the naming scheme of Cray compiler wrapper and call it crayftn-ftn.

**Parameters**

- **name** (str) – name of this compiler. (default: 'crayftn-ftn')
- **exec_name** (str) – name of the executable. (default: 'ftn')

class fab.tools.**CrayFtnWrapper**(*compiler*)

Class for the Cray Fortran compiler wrapper. We add 'wrapper' to the class name to make this class distinct from the Crayftn compiler class.

**Parameters**
**compiler** (*Compiler*) – the compiler that the ftn wrapper will use.

class fab.tools.**Fcm**

This is the base class for FCM. All commands will be mapped back to the corresponding subversion commands.

Constructor.

This is class is extended by the FCM interface which is why name and executable are mutable.

**Parameters**

- **name** – Tool name, defaults to "subversion."
- **exec_name** – Tool executable, defaults to "svn."
- **category** – Tool category, defaults to SUBVERSION.

class fab.tools.**Flags**(*list_of_flags=None*)

This class represents a list of parameters for a tool. It is a list with some additional functionality.

TODO #22: This class and build_config.FlagsConfig should be combined.

**Parameters**

**list_of_flags** (Optional[List[str]]) – List of parameters to initialise this object with. (default: None)

**checksum**()

**Return type**

str

**Returns**

a checksum of the flags.

**add_flags**(*new_flags*)

Adds the specified flags to the list of flags.

**Parameters**

**new_flags** (Union[str, List[str]]) – A single string or list of strings which are the flags to be added.

**remove_flag**(*remove_flag*, *has_parameter=False*)

Removes all occurrences of *remove_flag* in flags. If has_parameter is defined, the next entry in flags will also be removed, and if this object contains this flag+parameter without space (e.g. *-J/tmp*), it will be correctly removed. Note that only the flag itself must be specified, you cannot remove a flag only if a specific parameter is given (i.e. *remove_flag="-J/tmp"* will not work if this object contains *[…,"-J", "/tmp"]*).

**Parameters**

- **remove_flag** (str) – the flag to remove
- **has_parameter** (bool) – if the flag to remove takes a parameter (default: False)

class fab.tools.**FortranCompiler**(*name*, *exec_name*, *suite*, *version_regex*, *mpi=False*, *compile_flag=None*, *output_flag=None*, *openmp_flag=None*, *version_argument=None*, *module_folder_flag=None*, *syntax_only_flag=None*)

This is the base class for a Fortran compiler. It is a compiler that needs to support a module output path and support for syntax-only compilation (which will only generate the .mod files).

**Parameters**

- **name** (str) – name of the compiler.
- **exec_name** (str) – name of the executable to start.
- **suite** (str) – name of the compiler suite.
- **version_regex** (str) – A regular expression that allows extraction of the version number from the version output of the compiler.
- **mpi** (bool) – whether MPI is supported by this compiler or not. (default: False)

- **compile_flag** (Optional[str]) – the compilation flag to use when only requesting compilation (not linking). (default: None)
- **output_flag** (Optional[str]) – the compilation flag to use to indicate the name of the output file (default: None)
- **openmp_flag** (Optional[str]) – the flag to use to enable OpenMP (default: None)
- **module_folder_flag** (Optional[str]) – the compiler flag to indicate where to store created module files. (default: None)
- **syntax_only_flag** (Optional[str]) – flag to indicate to only do a syntax check. The side effect is that the module files are created. (default: None)

property **has_syntax_only**: bool

**Returns**
whether this compiler supports a syntax-only feature.

**set_module_output_path**(*path*)

Sets the output path for modules.

**Params path**
the path to the output directory.

**get_all_commandline_options**(*config*, *input_file*, *output_file*, *add_flags=None*, *syntax_only=False*)

This function returns all command line options for a Fortran compiler (but not the executable name). It is used by a compiler wrapper to pass the right flags to the wrapper. This Fortran-specific implementation adds the module- and syntax-only flags (as required) to the standard compiler flags.

**Parameters**

- **input_file** (Path) – the name of the input file.
- **output_file** (Path) – the name of the output file.
- **config** (*BuildConfig*) – The BuildConfig, from which compiler profile and OpenMP status are taken.
- **add_flags** (Optional[List[str]]) – additional flags for the compiler. (default: None)
- **syntax_only** (Optional[bool]) – if set, the compiler will only do a syntax check (default: False)

**Return type**
List[str]

**Returns**
all command line options for Fortran compilation.

**compile_file**(*input_file*, *output_file*, *config*, *add_flags=None*, *syntax_only=False*)

Compiles a file. This basically re-implements *compile_file* of the base class, but passes the syntax_only flag in

**Parameters**

- **input_file** (Path) – the name of the input file.
- **output_file** (Path) – the name of the output file.
- **config** (*BuildConfig*) – The BuildConfig, from which compiler profile and OpenMP status are taken.
- **add_flags** (Optional[List[str]]) – additional flags for the compiler. (default: None)

- **syntax_only** (Optional[bool]) – if set, the compiler will only do a syntax check (default: False)

class fab.tools.**Fpp**

Class for Intel's Fortran-specific preprocessor.

class fab.tools.**Gcc**(*name='gcc'*, *exec_name='gcc'*, *mpi=False*)

Class for GNU's gcc compiler.

**Parameters**

- **name** (str) – name of this compiler. (default: 'gcc')
- **exec_name** (str) – name of the executable. (default: 'gcc')
- **mpi** (bool) – whether the compiler supports MPI. (default: False)

class fab.tools.**Gfortran**(*name='gfortran'*, *exec_name='gfortran'*)

Class for GNU's gfortran compiler.

**Parameters**

- **name** (str) – name of this compiler. (default: 'gfortran')
- **exec_name** (str) – name of the executable. (default: 'gfortran')
- **mpi** – whether the compiler supports MPI.

class fab.tools.**Git**

Interface to Git version control system.

Constructor.

**Parameters**

- **name** – Display name of this tool.
- **exec_name** – Executable for this tool.
- **category** – Tool belongs to this category.

**current_commit**(*folder=None*)

**Return type**
str

**Returns**
the hash of the current commit.

**Parameters**
**folder** (Union[Path, str, None]) – the folder for which to determine the current commitf (defaults to .). (default: None)

**init**(*folder*)

Initialises a directory.

**Parameters**
**folder** (Union[Path, str]) – the directory to initialise.

**clean**(*folder*)

Removes all non versioned files in a directory.

**Parameters**
**folder** (Union[Path, str]) – the directory to clean.

**fetch**(*src*, *dst*, *revision*)

Runs *git fetch* in the specified directory

**Parameters**

- **src** (Union[str, Path]) – the source directory from which to fetch
- **revision** (Optional[str]) – the revision to fetch (can be "" for latest revision)
- **dst** (Union[str, Path]) – the directory in which to run fetch.

**checkout**(*src*, *dst=''*, *revision=None*)

Checkout or update a Git repo.

**Parameters**

- **src** (str) – the source directory from which to checkout.
- **dst** (str) – the directory in which to run checkout. (default: '')
- **revision** (Optional[str]) – the revision to check out (can be "" for latest revision). (default: None)

**merge**(*dst*, *revision=None*)

Merge a git repo into a local working copy. If the merge fails, it will run *git merge –abort* to clean the directory.

**Parameters**

- **dst** (Union[str, Path]) – the directory to merge in.
- **revision** (Optional[str]) – the revision number (only used for error message, it relies on git fetch running previously). (default: None)

**class** fab.tools.**Icc**(*name='icc'*, *exec_name='icc'*)

Class for the Intel's icc compiler.

**Parameters**

- **name** (str) – name of this compiler. (default: 'icc')
- **exec_name** (str) – name of the executable. (default: 'icc')
- **mpi** – whether the compiler supports MPI.

**class** fab.tools.**Icx**(*name='icx'*, *exec_name='icx'*)

Class for the Intel's new llvm based icx compiler.

**Parameters**

- **name** (str) – name of this compiler. (default: 'icx')
- **exec_name** (str) – name of the executable. (default: 'icx')

**class** fab.tools.**Ifort**(*name='ifort'*, *exec_name='ifort'*)

Class for Intel's ifort compiler.

**Parameters**

- **name** (str) – name of this compiler. (default: 'ifort')
- **exec_name** (str) – name of the executable. (default: 'ifort')
- **mpi** – whether the compiler supports MPI.

class fab.tools.**Ifx**(*name='ifx'*, *exec_name='ifx'*)

Class for Intel's new ifx compiler.

**Parameters**

- **name** (str) – name of this compiler. (default: 'ifx')
- **exec_name** (str) – name of the executable. (default: 'ifx')

class fab.tools.**Linker**(*compiler*, *linker=None*, *name=None*)

This is the base class for any Linker. It takes an existing compiler instance as parameter, and optional another linker. The latter is used to get linker settings - for example, linker-mpif90-gfortran will use mpif90-gfortran as compiler (i.e. to test if it is available and get compilation flags), and linker-gfortran as linker. This way a user only has to specify linker flags in the most basic class (gfortran), all other linker wrapper will inherit the settings.

**Parameters**

- **compiler** (*Compiler*) – a compiler instance
- **linker** (Optional[*Linker*]) – an optional linker instance (default: None)
- **name** (Optional[str]) – name of the linker (default: None)

**Raises**

- **RuntimeError** – if both compiler and linker are specified.
- **RuntimeError** – if neither compiler nor linker is specified.

**check_available**()

**Return type**
bool

**Returns**
whether this linker is available by asking the wrapped linker or compiler.

**property suite:** str

**Returns**
the suite this linker belongs to by getting it from the wrapped compiler.

**property mpi:** bool

**Returns**
whether this linker supports MPI or not by checking with the wrapped compiler.

**property openmp:** bool

**Returns**
whether this linker supports OpenMP or not by checking with the wrapped compiler.

**property output_flag:** str

**Returns**
the flag that is used to specify the output name.

**define_profile**(*name*, *inherit_from=None*)

Defines a new profile name, and allows to specify if this new profile inherit settings from an existing profile.

**Parameters**

- **name** (str) – Name of the profile to define.

- **inherit_from** (Optional[str]) – Optional name of a profile to inherit settings from. (default: None)

get_profile_flags(*profile*)

**Return type**
List[str]

**Returns**
the ProfileFlags for the given profile, combined from the wrapped compiler and this wrapper.

**Parameters**
**profile** (str) – the profile to use.

get_lib_flags(*lib*)

Gets the standard flags for a standard library

**Parameters**
**lib** (str) – the library name

**Return type**
List[str]

**Returns**
a list of flags

**Raises**
RuntimeError – if lib is not recognised

add_lib_flags(*lib*, *flags*, *silent_replace=False*)

Add a set of flags for a standard library

**Parameters**

- **lib** (str) – the library name
- **flags** (List[str]) – the flags to use with the library
- **silent_replace** (bool) – if set, no warning will be printed when an existing lib is overwritten. (default: False)

add_pre_lib_flags(*flags*, *profile=None*)

Add a set of flags to use before any library-specific flags

**Parameters**
**flags** (List[str]) – the flags to include

add_post_lib_flags(*flags*, *profile=None*)

Add a set of flags to use after any library-specific flags

**Parameters**
**flags** (List[str]) – the flags to include

get_pre_link_flags(*config*)

Returns the list of pre-link flags. It will concatenate the flags for this instance with all potentially wrapped linkers. This wrapper's flag will come first - the assumption is that the pre-link flags are likely paths, so we need a wrapper to be able to put a search path before the paths from a wrapped linker.

**Return type**
List[str]

**Returns**
List of pre-link flags of this linker and all wrapped linkers

**get_post_link_flags**(*config*)

Returns the list of post-link flags. It will concatenate the flags for this instance with all potentially wrapped linkers. This wrapper's flag will be added to the end.

**Return type**
List[str]

**Returns**
List of post-link flags of this linker and all wrapped linkers

**link**(*input_files*, *output_file*, *config*, *libs=None*, *add_flags=None*)

Executes the linker with the specified input files, creating *output_file*.

**Parameters**

- **input_files** (List[Path]) – list of input files to link.
- **output_file** (Path) – output file.
- **config** (*BuildConfig*) – The BuildConfig, from which compiler profile and OpenMP status are taken.
- **libs** (Optional[List[str]]) – additional libraries to link with. (default: None)

**Return type**
str

**Returns**
the stdout of the link command

**class** fab.tools.**Mpif90**(*compiler*)

Class for a simple wrapper for using a compiler driver (like mpif90) It will be using the name "mpif90-COMPILER_NAME" and calls *mpif90*. All flags from the original compiler will be used when using the wrapper as compiler.

**Parameters**
**compiler** (*Compiler*) – the compiler that the mpif90 wrapper will use.

**class** fab.tools.**Mpicc**(*compiler*)

Class for a simple wrapper for using a compiler driver (like mpicc) It will be using the name "mpicc-COMPILER_NAME" and calls *mpicc*. All flags from the original compiler will be used when using the wrapper as compiler.

**Parameters**
**compiler** (*Compiler*) – the compiler that the mpicc wrapper will use.

**class** fab.tools.**Nvc**(*name='nvc'*, *exec_name='nvc'*)

Class for Nvidia's nvc compiler. Note that the '-' in the Nvidia version number is ignored, e.g. 23.5-0 would return '23.5'.

**Parameters**

- **name** (str) – name of this compiler. (default: 'nvc')
- **exec_name** (str) – name of the executable. (default: 'nvc')

**class** fab.tools.**Nvfortran**(*name='nvfortran'*, *exec_name='nvfortran'*)

Class for Nvidia's nvfortran compiler. Note that the '-' in the Nvidia version number is ignored, e.g. 23.5-0 would return '23.5'.

**Parameters**

- **name** (str) – name of this compiler. (default: 'nvfortran')

- **exec_name** (str) – name of the executable. (default: 'nvfortran')

class fab.tools.**Preprocessor**(*name*, *exec_name*, *category*, *availability_option=None*)

This is the base class for any preprocessor.

**Parameters**

- **name** (str) – the name of the preprocessor.
- **exec_name** (Union[str, Path]) – the name of the executable.
- **category** (*Category*) – the category (C_PREPROCESSOR or FORTRAN_PREPROCESSOR)

**preprocess**(*input_file*, *output_file*, *add_flags=None*)

Calls the preprocessor to process the specified input file, creating the requested output file.

**Parameters**

- **input_file** (Path) – input file.
- **output_file** (Path) – the output filename.
- **add_flags** (Optional[List[Union[str, Path]]]) – List with additional flags to be used. (default: None)

class fab.tools.**ProfileFlags**

A list of flags that support a 'profile' to be used. If no profile is specified, it will use "" (empty string) as 'profile'. All functions take an optional profile parameter, so this class can also be used for tools that do not need a profile.

**define_profile**(*name*, *inherit_from=None*)

Defines a new profile name, and allows to specify if this new profile inherit settings from an existing profile. If inherit_from is specified, the newly defined profile will inherit from an existing profile (including the default profile "").

**Parameters**

- **name** (str) – Name of the profile to define.
- **inherit_from** (Optional[str]) – Optional name of a profile to inherit settings from. (default: None)

**add_flags**(*new_flags*, *profile=None*)

Adds the specified flags to the list of flags.

**Parameters**

**new_flags** (Union[str, List[str]]) – A single string or list of strings which are the flags to be added.

**remove_flag**(*remove_flag*, *profile=None*, *has_parameter=False*)

Removes all occurrences of *remove_flag* in flags. If *has_parameter* is defined, the next entry in flags will also be removed, and if this object contains this flag+parameter without space (e.g. *-J/tmp*), it will be correctly removed. Note that only the flag itself must be specified, you cannot remove a flag only if a specific parameter is given (i.e. *remove_flag="-J/tmp"* will not work if this object contains *[…,"-J", "/tmp"]*).

**Parameters**

- **remove_flag** (str) – the flag to remove
- **has_parameter** (bool) – if the flag to remove takes a parameter (default: False)

**checksum**(*profile=None*)

**Return type**
str

**Returns**
a checksum of the flags.

class fab.tools.**Psyclone**

This is the base class for *PSyclone*.

**check_available**()

This function determines if PSyclone is available. Additionally, it established the version, since command line option changes significantly from python 2.5.0 to the next release.

**Return type**
bool

**process**(*config*, *x90_file*, *psy_file=None*, *alg_file=None*, *transformed_file=None*, *transformation_script=None*, *additional_parameters=None*, *kernel_roots=None*, *api=None*)

Run PSyclone with the specified parameters. If PSyclone is used to transform existing Fortran files, *api* must be None, and the output file name is *transformed_file*. If PSyclone is using its DSL features, api must be a valid PSyclone API, and the two output filenames are *psy_file* and *alg_file*.

**Parameters**

- **api** (Optional[str]) – the PSyclone API. (default: None)
- **x90_file** (Path) – the input file for PSyclone
- **psy_file** (Optional[Path]) – the output PSy-layer file. (default: None)
- **alg_file** (Union[Path, str, None]) – the output modified algorithm file. (default: None)
- **transformed_file** (Optional[Path]) – the output filename if PSyclone is called as transformation tool. (default: None)
- **transformation_script** (Optional[Callable[[Path, *BuildConfig*], Path]]) – an optional transformation script (default: None)
- **additional_parameters** (Optional[List[str]]) – optional additional parameters for PSyclone (default: None)
- **kernel_roots** (Optional[List[Union[str, Path]]]) – optional directories with kernels. (default: None)

class fab.tools.**Rsync**

This is the base class for *rsync*.

**execute**(*src*, *dst*)

Execute an rsync command from src to dst. It supports ~ expansion for src, and makes sure that *src* end with a / so that rsync does not create a sub-directory.

**Parameters**

- **src** (Path) – the input path.
- **dst** (Path) – destination path.

class fab.tools.**Shell**(*name*)

A simple wrapper that runs a shell script. There seems to be no consistent way to simply check if a shell is working - not only support a version command (e.g. sh and dash don't). Instead, availability is tested by running a simple 'echo' command.

**Name**
the path to the script to run.

**exec**(*command*)

Executes the specified command.

**Parameters**
**command** (Union[str, List[Union[str, Path]]]) – the command and potential parameters to execute.

**Return type**
str

**Returns**
stdout of the result.

**class** fab.tools.**Subversion**(*name=None*, *exec_name=None*, *category=Category.SUBVERSION*)

Interface to the Subversion version control system.

Constructor.

This is class is extended by the FCM interface which is why name and executable are mutable.

**Parameters**

- **name** (Optional[str]) – Tool name, defaults to "subversion." (default: None)
- **exec_name** (Union[Path, str, None]) – Tool executable, defaults to "svn." (default: None)
- **category** (*Category*) – Tool category, defaults to SUBVERSION. (default: <Category.SUBVERSION: 9>)

**execute**(*pre_commands=None*, *revision=None*, *post_commands=None*, *env=None*, *cwd=None*, *capture_output=True*)

Executes a svn command.

**Parameters**

- **pre_commands** (Optional[List[str]]) – List of strings to be sent to subprocess.run() as the command. (default: None)
- **revision** (Union[int, str, None]) – optional revision number as argument (default: None)
- **post_commands** (Optional[List[str]]) – List of additional strings to be sent to subprocess.run() after the optional revision number. (default: None)
- **env** (Optional[Dict[str, str]]) – Optional env for the command. By default it will use the current session's environment. (default: None)
- **capture_output** – If True, capture and return stdout. If False, the command will print its output directly to the console. (default: True)

**Return type**
str

**export**(*src*, *dst*, *revision=None*)

Runs svn export.

**Parameters**

- **src** (Union[str, Path]) – from where to export.
- **dst** (Union[str, Path]) – destination path.

- **revision** (Optional[str]) – revision to export. (default: None)

**checkout**(*src*, *dst*, *revision=None*)

Runs svn checkout.

**Parameters**

- **src** (Union[str, Path]) – from where to check out.
- **dst** (Union[str, Path]) – destination path.
- **revision** (Optional[str]) – revision to check out. (default: None)

**update**(*dst*, *revision=None*)

Runs svn checkout.

**Parameters**

- **dst** (Union[str, Path]) – destination path.
- **revision** (Optional[str]) – revision to check out. (default: None)

**merge**(*src*, *dst*, *revision=None*)

Runs svn merge.

**Parameters**

- **src** (Union[str, Path]) – the src URI.
- **dst** (Union[str, Path]) – destination path.
- **revision** (Optional[str]) – revision to check out. (default: None)

**class** fab.tools.**Tool**(*name*, *exec_name*, *category=Category.MISC*, *availability_option=None*)

This is the base class for all tools. It stores the name of the tool, the name of the executable, and provides a *run* method.

**Parameters**

- **name** (str) – name of the tool.
- **exec_name** (Union[str, Path]) – name or full path of the executable to start.
- **category** (*Category*) – the Category to which this tool belongs. (default: <Category.MISC: 13>)
- **availability_option** (Union[str, List[str], None]) – a command line option for the tool to test if the tool is available on the current system. Defaults to *–version*. (default: None)

**check_available**()

Run a 'test' command to check if this tool is available in the system. :rtype: bool :returns: whether the tool is working (True) or not.

**set_full_path**(*full_path*)

This function adds the full path to a tool. This allows tools to be used that are not in the user's PATH. The ToolRepository will automatically update the path for a tool if the user specified a full path.

**Parameters**
**full_path** (Path) – the full path to the executable.

**property is_available:** bool

Checks if the tool is available or not. It will call a tool-specific function check_available to determine this, but will cache the results to avoid testing a tool more than once.

**Returns**
whether the tool is available (i.e. installed and working).

**property is_compiler: bool**

Returns whether this tool is a (Fortran or C) compiler or not.

**property exec_path: Path**

**Returns**
the path of the executable.

**property exec_name: str**

**Returns**
the name of the executable.

**property name: str**

**Returns**
the name of the tool.

**property availability_option: str | List[str]**

**Returns**
the option to use to check if the tool is available.

**property category: *Category***

**Returns**
the category of this tool.

**get_flags**(*profile=None*)

**Returns**
the flags to be used with this tool.

**add_flags**(*new_flags*, *profile=None*)

Adds the specified flags to the list of flags.

**Parameters**
**new_flags** (Union[str, List[str]]) – A single string or list of strings which are the flags to be added.

**define_profile**(*name*, *inherit_from=None*)

Defines a new profile name, and allows to specify if this new profile inherit settings from an existing profile.

**Parameters**

- **name** (str) – Name of the profile to define.
- **inherit_from** (Optional[str]) – Optional name of a profile to inherit settings from. (default: None)

**property logger: Logger**

**Returns**
a logger object for convenience.

**run**(*additional_parameters=None*, *profile=None*, *env=None*, *cwd=None*, *capture_output=True*)

Run the binary as a subprocess.

**Parameters**

- **additional_parameters** (Union[str, Sequence[Union[Path, str]], None]) – List of strings or paths to be sent to subprocess.run() as additional parameters for the command. Any path will be converted to a normal string. (default: None)
- **env** (Optional[Dict[str, str]]) – Optional env for the command. By default it will use the current session's environment. (default: None)
- **capture_output** – If True, capture and return stdout. If False, the command will print its output directly to the console. (default: True)

**Raises**

- **RuntimeError** – if the code is not available.
- **RuntimeError** – if the return code of the executable is not 0.

**Return type**
str

class fab.tools.**ToolBox**

This class implements the tool box. It stores one tool for each category to be used in a FAB build.

**add_tool**(*tool*, *silent_replace=False*)

Adds a tool for a given category.

**Parameters**

- **tool** (*Tool*) – the tool to add.
- **silent_replace** (bool) – if set, no warning will be printed if an existing tool is replaced. (default: False)

**Raises**
**RuntimeError** – if the tool to be added is not available.

**Return type**
None

**get_tool**(*category*, *mpi=None*, *openmp=None*)

Returns the tool for the specified category.

**Parameters**

- **category** (*Category*) – the name of the category in which to look for the tool.
- **mpi** (Optional[bool]) – if no compiler or linker is explicitly specified in this tool box, use the MPI and OpenMP setting to find an appropriate default from the tool repository. (default: None)
- **mpi** – if no compiler or linker is explicitly specified in this tool box, use the MPI and OpenMP setting to find an appropriate default from the tool repository.

**Raises**
**KeyError** – if the category is not known.

**Return type**
*Tool*

class fab.tools.**ToolRepository**

This class implements the tool repository. It stores a list of tools for various categories. For each compiler, it will automatically create a tool called "linker-{compiler-name}" which can be used for linking with the specified compiler.

Singleton access. Changes the value of _singleton so that the constructor can verify that it is indeed called from here.

**add_tool**(*tool*)

Creates an instance of the specified class and adds it to the tool repository. If the tool is a compiler, it automatically adds the compiler as a linker as well (named "linker-{tool.name}").

**Parameters**
**tool** (*Tool*) – the tool to add.

**get_tool**(*category*, *name*)

This functions returns a tool with a given name. The name can either be a Fab compiler name (including wrapper naming), e.g. mpif90-gfortran, or linker-mpif90-ifort, or just the name of the executable (mpif90). If a Fab name is specified, the corresponding tool will be returned, even if it should not be available (allowing default site scripts to setup any compiler, even if they are not available everywhere). If an exec name is specified, the tool must be available. This is required to make sure the user gets the right tool: by specifying just mpif90, it is not clear if the user wants mpif90-ifort, mpif90-gfortran, ... . But only one of these tools will actually be available (the wrapper checks the version number to detect the compiler-vendor).

The name can also be specified using an absolute path, in which case only the stem will be used to look up the name, but the returned tool will be updated to use the full path to the tool. This allows the usage of e.g. compilers that are not in $PATH of the user.

**Return type**
*Tool*

**Returns**
the tool with a given name in the specified category.

**Parameters**

- **category** (*Category*) – the name of the category in which to look for the tool.
- **name** (str) – the name of the tool to find. A full path can be used, in which case only the stem of the path is used, and the tool will be updated to use the absolute path specified.

**Raises**

- **KeyError** – if there is no tool in this category.
- **KeyError** – if no tool in the given category has the requested name.

**set_default_compiler_suite**(*suite*)

Sets the default for linker and compilers to be of the given compiler suite.

**Parameters**
**suite** (str) – the name of the compiler suite to make the default.

**get_default**(*category*, *mpi=None*, *openmp=None*)

Returns the default tool for a given category that is available. For most tools that will be the first entry in the list of tools. The exception are compilers and linker: in this case it must be specified if MPI support is required or not. And the default return will be the first tool that either supports MPI or not.

**Parameters**

- **category** (*Category*) – the category for which to return the default tool.
- **mpi** (Optional[bool]) – if a compiler or linker is required that supports MPI. (default: None)
- **openmp** (Optional[bool]) – if a compiler or linker is required that supports OpenMP. (default: None)

**Raises**

- **KeyError** – if the category does not exist.
- **RuntimeError** – if no tool in the requested category is available on the system.
- **RuntimeError** – if no compiler/linker is found with the requested level of MPI support (yes or no).

**class** fab.tools.**Versioning**(*name*, *exec_name*, *category*)

Base class for versioning tools like Git and Subversion.

Constructor.

**Parameters**

- **name** (str) – Display name of this tool.
- **exec_name** (Union[str, Path]) – Executable for this tool.
- **category** (*Category*) – Tool belongs to this category.

## Modules

| | |
|---|---|
| *ar* | This file contains the Ar class for archiving files. |
| *category* | This simple module defines an Enum for all allowed categories. |
| *compiler* | This file contains the base class for any compiler, and derived classes for gcc, gfortran, icc, ifort |
| *compiler_wrapper* | This file contains the base class for any compiler-wrapper, including the derived classes for mpif90, mpicc, and CrayFtnWrapper and CrayCcWrapper. |
| *flags* | This file contains a simple Flag class to manage tool flags. |
| *linker* | This file contains the base class for any Linker. |
| *preprocessor* | This file contains the base class for any preprocessor, and two derived classes for cpp and fpp. |
| *psyclone* | This file contains the tool class for PSyclone. |
| *rsync* | This file contains the Rsync class for synchronising file trees. |
| *shell* | This file contains a base class for shells. |
| *tool* | This file contains the base class for all tools, i.e. compiler, preprocessor, linker, archiver, Psyclone, rsync, versioning tools. |
| *tool_box* | This file contains the ToolBox class. |
| *tool_repository* | This file contains the ToolRepository class. |
| *versioning* | Versioning tools such as Subversion and Git. |

### fab.tools.ar

This file contains the Ar class for archiving files.

### Classes

| | |
|---|---|
| *Ar*() | This is the base class for *ar*. |

**class** fab.tools.ar.**Ar**

This is the base class for *ar*.

**create**(*output_fpath*, *members*)

Create the archive with the specified name, containing the listed members.

**Parameters**

- **output_fpath** (Path) – the output path.
- **members** (List[Union[str, Path]]) – the list of objects to be added to the archive.

## fab.tools.category

This simple module defines an Enum for all allowed categories.

### Classes

| | |
|---|---|
| *Category*(value) | This class defines the allowed tool categories. |

**class** fab.tools.category.**Category**(*value*)

This class defines the allowed tool categories.

**property is_compiler**

Returns if the category is either a C or a Fortran compiler.

## fab.tools.compiler

This file contains the base class for any compiler, and derived classes for gcc, gfortran, icc, ifort

### Classes

| | |
|---|---|
| *CCompiler*(name, exec_name, suite, version_regex) | This is the base class for a C compiler. |
| *Compiler*(name, exec_name, suite, ...[, mpi, ...]) | This is the base class for any compiler. |
| *Craycc*([name, exec_name]) | Class for the native Cray C compiler. |
| *Crayftn*([name, exec_name]) | Class for the native Cray Fortran compiler. |
| *FortranCompiler*(name, exec_name, suite, ...) | This is the base class for a Fortran compiler. |
| *Gcc*([name, exec_name, mpi]) | Class for GNU's gcc compiler. |
| *Gfortran*([name, exec_name]) | Class for GNU's gfortran compiler. |
| *Icc*([name, exec_name]) | Class for the Intel's icc compiler. |
| *Icx*([name, exec_name]) | Class for the Intel's new llvm based icx compiler. |
| *Ifort*([name, exec_name]) | Class for Intel's ifort compiler. |
| *Ifx*([name, exec_name]) | Class for Intel's new ifx compiler. |
| *Nvc*([name, exec_name]) | Class for Nvidia's nvc compiler. |
| *Nvfortran*([name, exec_name]) | Class for Nvidia's nvfortran compiler. |

class fab.tools.compiler.**Compiler**(*name*, *exec_name*, *suite*, *version_regex*, *category*, *mpi=False*, *compile_flag=None*, *output_flag=None*, *openmp_flag=None*, *version_argument=None*, *availability_option=None*)

This is the base class for any compiler. It provides flags for

- compilation only (-c),
- naming the output file (-o),
- OpenMP

**Parameters**

- **name** (str) – name of the compiler.
- **exec_name** (Union[str, Path]) – name of the executable to start.
- **suite** (str) – name of the compiler suite this tool belongs to.
- **version_regex** (str) – A regular expression that allows extraction of the version number from the version output of the compiler. The version is taken from the first group of a match.
- **category** (*Category*) – the Category (C_COMPILER or FORTRAN_COMPILER).
- **mpi** (bool) – whether the compiler or linker support MPI. (default: False)
- **compile_flag** (Optional[str]) – the compilation flag to use when only requesting compilation (not linking). (default: None)
- **output_flag** (Optional[str]) – the compilation flag to use to indicate the name of the output file (default: None)
- **openmp_flag** (Optional[str]) – the flag to use to enable OpenMP. If no flag is specified, it is assumed that the compiler does not support OpenMP. (default: None)
- **availability_option** (Union[str, List[str], None]) – a command line option for the tool to test if the tool is available on the current system. Defaults to *–version*. (default: None)

property **mpi**: bool

**Returns**
whether this compiler supports MPI or not.

property **openmp**: bool

**Returns**
compiler's OpenMP support.

property **openmp_flag**: str

**Returns**
compiler argument to enable OpenMP.

property **compile_flag**: str

**Returns**
the flag to indicate compilation only (not linking).

property **output_flag**: str

**Returns**
compiler argument for output file.

**get_hash**(*profile=None*)

**Return type**
int

**Returns**
hash of compiler name and version.

**get_flags**(*profile=None*)

Determines the flags to be used.

**Return type**
List[str]

**Returns**
the flags to be used with this tool.

**get_all_commandline_options**(*config*, *input_file*, *output_file*, *add_flags=None*)

This function returns all command line options for a compiler (but not the executable name). It is used by a compiler wrapper to pass the right flags to the wrapper. This base implementation adds the input and output filename (including the -o flag), the flag to only compile (and not link), and if required openmp.

**Parameters**

- **input_file** (Path) – the name of the input file.
- **output_file** (Path) – the name of the output file.
- **config** (*BuildConfig*) – The BuildConfig, from which compiler profile and OpenMP status are taken.
- **add_flags** (Optional[List[str]]) – additional flags for the compiler. (default: None)

**Return type**
List[str]

**Returns**
all command line options for compilation.

**compile_file**(*input_file*, *output_file*, *config*, *add_flags=None*)

Compiles a file. It will add the flag for compilation-only automatically, as well as the output directives. The current working directory for the command is set to the folder where the source file lives when compile_file is called. This is done to stop the compiler inserting folder information into the mod files, which would cause them to have different checksums depending on where they live.

**Parameters**

- **input_file** (Path) – the path of the input file.
- **output_file** (Path) – the path of the output file.
- **config** (*BuildConfig*) – The BuildConfig, from which compiler profile and OpenMP status are taken.
- **add_flags** (Optional[List[str]]) – additional compiler flags. (default: None)

**check_available**()

Checks if the compiler is available. While the method in the Tools base class would be sufficient (when using –version), in case of a compiler we also want to store the compiler version. So, re-implement check_available in a way that will automatically store the compiler version for later usage.

**Return type**
bool

**Returns**
whether the compiler is available or not. We do this by requesting the compiler version.

`get_version`()

Try to get the version of the given compiler.

Expects a version in a certain part of the –version output, which must adhere to the n.n.n format, with at least 2 parts.

**Return type**
`Tuple`[`int`, `...`]

**Returns**
a tuple of at least 2 integers, representing the version e.g. (6, 10, 1) for version '6.10.1'.

**Raises**
`RuntimeError` – if the compiler was not found, or if it returned an unrecognised output from the version command.

`run_version_command`(*version_command='--version'*)

Run the compiler's command to get its version.

**Parameters**
**version_command** (`Optional`[`str`]) – The compiler argument used to get version info. (default: `'--version'`)

**Return type**
`str`

**Returns**
The output from the version command.

**Raises**
`RuntimeError` – if the compiler was not found, or raised an error.

`get_version_string`()

Get a string representing the version of the given compiler.

**Return type**
`str`

**Returns**
a string of at least 2 numeric version components, i.e. major.minor[.patch, ...]

**Raises**
`RuntimeError` – if the compiler was not found, or if it returned an unrecognised output from the version command.

class fab.tools.compiler.**CCompiler**(*name*, *exec_name*, *suite*, *version_regex*, *mpi=False*, *compile_flag=None*, *output_flag=None*, *openmp_flag=None*, *version_argument=None*, *availability_option=None*)

This is the base class for a C compiler. It just sets the category of the compiler as convenience.

**Parameters**

- **name** (`str`) – name of the compiler.
- **exec_name** (`str`) – name of the executable to start.
- **suite** (`str`) – name of the compiler suite.
- **version_regex** (`str`) – A regular expression that allows extraction of the version number from the version output of the compiler.

- **mpi** (bool) – whether the compiler or linker support MPI. (default: False)
- **compile_flag** (Optional[str]) – the compilation flag to use when only requesting compilation (not linking). (default: None)
- **output_flag** (Optional[str]) – the compilation flag to use to indicate the name of the output file (default: None)
- **openmp_flag** (Optional[str]) – the flag to use to enable OpenMP (default: None)

class fab.tools.compiler.**FortranCompiler**(*name*, *exec_name*, *suite*, *version_regex*, *mpi=False*, *compile_flag=None*, *output_flag=None*, *openmp_flag=None*, *version_argument=None*, *module_folder_flag=None*, *syntax_only_flag=None*)

This is the base class for a Fortran compiler. It is a compiler that needs to support a module output path and support for syntax-only compilation (which will only generate the .mod files).

**Parameters**

- **name** (str) – name of the compiler.
- **exec_name** (str) – name of the executable to start.
- **suite** (str) – name of the compiler suite.
- **version_regex** (str) – A regular expression that allows extraction of the version number from the version output of the compiler.
- **mpi** (bool) – whether MPI is supported by this compiler or not. (default: False)
- **compile_flag** (Optional[str]) – the compilation flag to use when only requesting compilation (not linking). (default: None)
- **output_flag** (Optional[str]) – the compilation flag to use to indicate the name of the output file (default: None)
- **openmp_flag** (Optional[str]) – the flag to use to enable OpenMP (default: None)
- **module_folder_flag** (Optional[str]) – the compiler flag to indicate where to store created module files. (default: None)
- **syntax_only_flag** (Optional[str]) – flag to indicate to only do a syntax check. The side effect is that the module files are created. (default: None)

property **has_syntax_only**: bool

**Returns**
whether this compiler supports a syntax-only feature.

**set_module_output_path**(*path*)

Sets the output path for modules.

**Params path**
the path to the output directory.

**get_all_commandline_options**(*config*, *input_file*, *output_file*, *add_flags=None*, *syntax_only=False*)

This function returns all command line options for a Fortran compiler (but not the executable name). It is used by a compiler wrapper to pass the right flags to the wrapper. This Fortran-specific implementation adds the module- and syntax-only flags (as required) to the standard compiler flags.

**Parameters**

- **input_file** (Path) – the name of the input file.

- **output_file** (Path) – the name of the output file.
- **config** (*BuildConfig*) – The BuildConfig, from which compiler profile and OpenMP status are taken.
- **add_flags** (Optional[List[str]]) – additional flags for the compiler. (default: None)
- **syntax_only** (Optional[bool]) – if set, the compiler will only do a syntax check (default: False)

**Return type**
List[str]

**Returns**
all command line options for Fortran compilation.

**compile_file**(*input_file*, *output_file*, *config*, *add_flags=None*, *syntax_only=False*)

Compiles a file. This basically re-implements *compile_file* of the base class, but passes the syntax_only flag in

**Parameters**

- **input_file** (Path) – the name of the input file.
- **output_file** (Path) – the name of the output file.
- **config** (*BuildConfig*) – The BuildConfig, from which compiler profile and OpenMP status are taken.
- **add_flags** (Optional[List[str]]) – additional flags for the compiler. (default: None)
- **syntax_only** (Optional[bool]) – if set, the compiler will only do a syntax check (default: False)

**class** fab.tools.compiler.**Gcc**(*name='gcc'*, *exec_name='gcc'*, *mpi=False*)

Class for GNU's gcc compiler.

**Parameters**

- **name** (str) – name of this compiler. (default: 'gcc')
- **exec_name** (str) – name of the executable. (default: 'gcc')
- **mpi** (bool) – whether the compiler supports MPI. (default: False)

**class** fab.tools.compiler.**Gfortran**(*name='gfortran'*, *exec_name='gfortran'*)

Class for GNU's gfortran compiler.

**Parameters**

- **name** (str) – name of this compiler. (default: 'gfortran')
- **exec_name** (str) – name of the executable. (default: 'gfortran')
- **mpi** – whether the compiler supports MPI.

**class** fab.tools.compiler.**Icc**(*name='icc'*, *exec_name='icc'*)

Class for the Intel's icc compiler.

**Parameters**

- **name** (str) – name of this compiler. (default: 'icc')
- **exec_name** (str) – name of the executable. (default: 'icc')
- **mpi** – whether the compiler supports MPI.

class fab.tools.compiler.**Ifort**(*name='ifort'*, *exec_name='ifort'*)

Class for Intel's ifort compiler.

**Parameters**

- **name** (str) – name of this compiler. (default: 'ifort')
- **exec_name** (str) – name of the executable. (default: 'ifort')
- **mpi** – whether the compiler supports MPI.

class fab.tools.compiler.**Icx**(*name='icx'*, *exec_name='icx'*)

Class for the Intel's new llvm based icx compiler.

**Parameters**

- **name** (str) – name of this compiler. (default: 'icx')
- **exec_name** (str) – name of the executable. (default: 'icx')

class fab.tools.compiler.**Ifx**(*name='ifx'*, *exec_name='ifx'*)

Class for Intel's new ifx compiler.

**Parameters**

- **name** (str) – name of this compiler. (default: 'ifx')
- **exec_name** (str) – name of the executable. (default: 'ifx')

class fab.tools.compiler.**Nvc**(*name='nvc'*, *exec_name='nvc'*)

Class for Nvidia's nvc compiler. Note that the '-' in the Nvidia version number is ignored, e.g. 23.5-0 would return '23.5'.

**Parameters**

- **name** (str) – name of this compiler. (default: 'nvc')
- **exec_name** (str) – name of the executable. (default: 'nvc')

class fab.tools.compiler.**Nvfortran**(*name='nvfortran'*, *exec_name='nvfortran'*)

Class for Nvidia's nvfortran compiler. Note that the '-' in the Nvidia version number is ignored, e.g. 23.5-0 would return '23.5'.

**Parameters**

- **name** (str) – name of this compiler. (default: 'nvfortran')
- **exec_name** (str) – name of the executable. (default: 'nvfortran')

class fab.tools.compiler.**Craycc**(*name='craycc-cc'*, *exec_name='cc'*)

Class for the native Cray C compiler. Since cc is actually a compiler wrapper, follow the naming scheme of a compiler wrapper and call it: craycc-cc.

**Cray has two different compilers. Older ones have as version number:**
Cray C : Version 8.7.0 Tue Jul 23, 2024 07:39:46

**Newer compiler (several lines, the important one):**
Cray clang version 15.0.1 (66f7391d6a03cf932f321b9f6b1d8612ef5f362c)

We use the beginning ("cray c") to identify the compiler, which works for both cray c and cray clang. Then we ignore non-numbers, to reach the version number which is then extracted.

**Parameters**

- **name** (str) – name of this compiler. (default: 'craycc-cc')

- **exec_name** (str) – name of the executable. (default: 'cc')

class fab.tools.compiler.**Crayftn**(*name='crayftn-ftn'*, *exec_name='ftn'*)

Class for the native Cray Fortran compiler. Since ftn is actually a compiler wrapper, follow the naming scheme of Cray compiler wrapper and call it crayftn-ftn.

**Parameters**

- **name** (str) – name of this compiler. (default: 'crayftn-ftn')
- **exec_name** (str) – name of the executable. (default: 'ftn')

### fab.tools.compiler_wrapper

This file contains the base class for any compiler-wrapper, including the derived classes for mpif90, mpicc, and CrayFtnWrapper and CrayCcWrapper.

### Classes

| | |
|---|---|
| *CompilerWrapper*(name, exec_name, compiler[, mpi]) | A decorator-based compiler wrapper. |
| *CrayCcWrapper*(compiler) | Class for the Cray C compiler wrapper. |
| *CrayFtnWrapper*(compiler) | Class for the Cray Fortran compiler wrapper. |
| *Mpicc*(compiler) | Class for a simple wrapper for using a compiler driver (like mpicc) It will be using the name "mpicc-COMPILER_NAME" and calls *mpicc*. |
| *Mpif90*(compiler) | Class for a simple wrapper for using a compiler driver (like mpif90) It will be using the name "mpif90-COMPILER_NAME" and calls *mpif90*. |

class fab.tools.compiler_wrapper.**CompilerWrapper**(*name*, *exec_name*, *compiler*, *mpi=False*)

A decorator-based compiler wrapper. It basically uses a different executable name when compiling, but otherwise behaves like the wrapped compiler. An example of a compiler wrapper is *mpif90* (which can internally call e.g. gfortran, icc, ...)

**Parameters**

- **name** (str) – name of the wrapper.
- **exec_name** (str) – name of the executable to call.
- **compiler** (*Compiler*) – the compiler that is decorated.
- **mpi** (bool) – whether MPI is supported by this compiler or not. (default: False)

property **compiler**: *Compiler*

**Returns**
the compiler that is wrapped by this CompilerWrapper.

property **suite**: str

**Returns**
the compiler suite of this tool.

property **openmp_flag**: str

Returns the flag to enable OpenMP.

property has_syntax_only: bool

**Returns**
whether this compiler supports a syntax-only feature.

**Raises**
RuntimeError – if this function is called for a non-Fortran wrapped compiler.

get_flags(*profile=None*)

**Return type**
List[str]

**Returns**
the ProfileFlags for the given profile, combined from the wrapped compiler and this wrapper.

**Parameters**
**profile** (Optional[str]) – the profile to use. (default: None)

set_module_output_path(*path*)

Sets the output path for modules.

**Params path**
the path to the output directory.

**Raises**
RuntimeError – if this function is called for a non-Fortran wrapped compiler.

get_all_commandline_options(*config*, *input_file*, *output_file*, *add_flags=None*, *syntax_only=False*)

This function returns all command line options for a compiler wrapper. The syntax_only flag is only accepted, if the wrapped compiler is a Fortran compiler. Otherwise, an exception will be raised.

**Parameters**

- **input_file** (Path) – the name of the input file.
- **output_file** (Path) – the name of the output file.
- **config** (*BuildConfig*) – The BuildConfig, from which compiler profile and OpenMP status are taken.
- **add_flags** (Optional[List[str]]) – additional flags for the compiler. (default: None)
- **syntax_only** (Optional[bool]) – if set, the compiler will only do a syntax check (default: False)

**Return type**
List[str]

**Returns**
command line flags for compiler wrapper.

**Raises**
RuntimeError – if syntax_only is requested for a non-Fortran compiler.

compile_file(*input_file*, *output_file*, *config*, *add_flags=None*, *syntax_only=None*)

Compiles a file using the wrapper compiler.

**Parameters**

- **input_file** (Path) – the name of the input file.
- **output_file** (Path) – the name of the output file.

- **config** (*BuildConfig*) – The BuildConfig, from which compiler profile and OpenMP status are taken.
- **add_flags** (Optional[List[str]]) – additional flags for the compiler. (default: None)
- **syntax_only** (Optional[bool]) – if set, the compiler will only do a syntax check (default: None)

class fab.tools.compiler_wrapper.**Mpif90**(*compiler*)

Class for a simple wrapper for using a compiler driver (like mpif90) It will be using the name "mpif90-COMPILER_NAME" and calls *mpif90*. All flags from the original compiler will be used when using the wrapper as compiler.

**Parameters**
**compiler** (*Compiler*) – the compiler that the mpif90 wrapper will use.

class fab.tools.compiler_wrapper.**Mpicc**(*compiler*)

Class for a simple wrapper for using a compiler driver (like mpicc) It will be using the name "mpicc-COMPILER_NAME" and calls *mpicc*. All flags from the original compiler will be used when using the wrapper as compiler.

**Parameters**
**compiler** (*Compiler*) – the compiler that the mpicc wrapper will use.

class fab.tools.compiler_wrapper.**CrayFtnWrapper**(*compiler*)

Class for the Cray Fortran compiler wrapper. We add 'wrapper' to the class name to make this class distinct from the Crayftn compiler class.

**Parameters**
**compiler** (*Compiler*) – the compiler that the ftn wrapper will use.

class fab.tools.compiler_wrapper.**CrayCcWrapper**(*compiler*)

Class for the Cray C compiler wrapper. We add 'wrapper' to the class name to make this class distinct from the Craycc compiler class

**Parameters**
**compiler** (*Compiler*) – the compiler that the mpicc wrapper will use.

### fab.tools.flags

This file contains a simple Flag class to manage tool flags. It will need to be combined with build_config.FlagsConfig in a follow up PR.

### Classes

| | |
|---|---|
| *Flags*([list_of_flags]) | This class represents a list of parameters for a tool. |
| *ProfileFlags*() | A list of flags that support a 'profile' to be used. |

class fab.tools.flags.**Flags**(*list_of_flags=None*)

This class represents a list of parameters for a tool. It is a list with some additional functionality.

TODO #22: This class and build_config.FlagsConfig should be combined.

**Parameters**
**list_of_flags** (Optional[List[str]]) – List of parameters to initialise this object with. (default: None)

**checksum**()

**Return type**
str

**Returns**
a checksum of the flags.

**add_flags**(*new_flags*)

Adds the specified flags to the list of flags.

**Parameters**
**new_flags** (Union[str, List[str]]) – A single string or list of strings which are the flags to be added.

**remove_flag**(*remove_flag*, *has_parameter=False*)

Removes all occurrences of *remove_flag* in flags. If has_parameter is defined, the next entry in flags will also be removed, and if this object contains this flag+parameter without space (e.g. *-J/tmp*), it will be correctly removed. Note that only the flag itself must be specified, you cannot remove a flag only if a specific parameter is given (i.e. *remove_flag="-J/tmp"* will not work if this object contains *[...,"-J", "/tmp"]*).

**Parameters**

- **remove_flag** (str) – the flag to remove
- **has_parameter** (bool) – if the flag to remove takes a parameter (default: False)

**class** fab.tools.flags.**ProfileFlags**

A list of flags that support a 'profile' to be used. If no profile is specified, it will use "" (empty string) as 'profile'. All functions take an optional profile parameter, so this class can also be used for tools that do not need a profile.

**define_profile**(*name*, *inherit_from=None*)

Defines a new profile name, and allows to specify if this new profile inherit settings from an existing profile. If inherit_from is specified, the newly defined profile will inherit from an existing profile (including the default profile "").

**Parameters**

- **name** (str) – Name of the profile to define.
- **inherit_from** (Optional[str]) – Optional name of a profile to inherit settings from. (default: None)

**add_flags**(*new_flags*, *profile=None*)

Adds the specified flags to the list of flags.

**Parameters**
**new_flags** (Union[str, List[str]]) – A single string or list of strings which are the flags to be added.

**remove_flag**(*remove_flag*, *profile=None*, *has_parameter=False*)

Removes all occurrences of *remove_flag* in flags. If *has_parameter* is defined, the next entry in flags will also be removed, and if this object contains this flag+parameter without space (e.g. *-J/tmp*), it will be correctly removed. Note that only the flag itself must be specified, you cannot remove a flag only if a specific parameter is given (i.e. *remove_flag="-J/tmp"* will not work if this object contains *[...,"-J", "/tmp"]*).

**Parameters**

- **remove_flag** (str) – the flag to remove
- **has_parameter** (bool) – if the flag to remove takes a parameter (default: False)

**checksum**(*profile=None*)

**Return type**
str

**Returns**
a checksum of the flags.

### fab.tools.linker

This file contains the base class for any Linker.

### Classes

| | |
|---|---|
| *Linker*(compiler[, linker, name]) | This is the base class for any Linker. |

**class** fab.tools.linker.**Linker**(*compiler*, *linker=None*, *name=None*)

This is the base class for any Linker. It takes an existing compiler instance as parameter, and optional another linker. The latter is used to get linker settings - for example, linker-mpif90-gfortran will use mpif90-gfortran as compiler (i.e. to test if it is available and get compilation flags), and linker-gfortran as linker. This way a user only has to specify linker flags in the most basic class (gfortran), all other linker wrapper will inherit the settings.

**Parameters**

- **compiler** (*Compiler*) – a compiler instance
- **linker** (Optional[*Linker*]) – an optional linker instance (default: None)
- **name** (Optional[str]) – name of the linker (default: None)

**Raises**

- **RuntimeError** – if both compiler and linker are specified.
- **RuntimeError** – if neither compiler nor linker is specified.

**check_available**()

**Return type**
bool

**Returns**
whether this linker is available by asking the wrapped linker or compiler.

**property suite:** str

**Returns**
the suite this linker belongs to by getting it from the wrapped compiler.

**property mpi:** bool

**Returns**
whether this linker supports MPI or not by checking with the wrapped compiler.

property openmp: bool

**Returns**
whether this linker supports OpenMP or not by checking with the wrapped compiler.

property output_flag: str

**Returns**
the flag that is used to specify the output name.

define_profile(*name*, *inherit_from=None*)

Defines a new profile name, and allows to specify if this new profile inherit settings from an existing profile.

**Parameters**

- **name** (str) – Name of the profile to define.
- **inherit_from** (Optional[str]) – Optional name of a profile to inherit settings from. (default: None)

get_profile_flags(*profile*)

**Return type**
List[str]

**Returns**
the ProfileFlags for the given profile, combined from the wrapped compiler and this wrapper.

**Parameters**
**profile** (str) – the profile to use.

get_lib_flags(*lib*)

Gets the standard flags for a standard library

**Parameters**
**lib** (str) – the library name

**Return type**
List[str]

**Returns**
a list of flags

**Raises**
RuntimeError – if lib is not recognised

add_lib_flags(*lib*, *flags*, *silent_replace=False*)

Add a set of flags for a standard library

**Parameters**

- **lib** (str) – the library name
- **flags** (List[str]) – the flags to use with the library
- **silent_replace** (bool) – if set, no warning will be printed when an existing lib is overwritten. (default: False)

add_pre_lib_flags(*flags*, *profile=None*)

Add a set of flags to use before any library-specific flags

**Parameters**
**flags** (List[str]) – the flags to include

**add_post_lib_flags**(*flags*, *profile=None*)

Add a set of flags to use after any library-specific flags

**Parameters**
**flags** (List[str]) – the flags to include

**get_pre_link_flags**(*config*)

Returns the list of pre-link flags. It will concatenate the flags for this instance with all potentially wrapped linkers. This wrapper's flag will come first - the assumption is that the pre-link flags are likely paths, so we need a wrapper to be able to put a search path before the paths from a wrapped linker.

**Return type**
List[str]

**Returns**
List of pre-link flags of this linker and all wrapped linkers

**get_post_link_flags**(*config*)

Returns the list of post-link flags. It will concatenate the flags for this instance with all potentially wrapped linkers. This wrapper's flag will be added to the end.

**Return type**
List[str]

**Returns**
List of post-link flags of this linker and all wrapped linkers

**link**(*input_files*, *output_file*, *config*, *libs=None*, *add_flags=None*)

Executes the linker with the specified input files, creating *output_file*.

**Parameters**

- **input_files** (List[Path]) – list of input files to link.
- **output_file** (Path) – output file.
- **config** (*BuildConfig*) – The BuildConfig, from which compiler profile and OpenMP status are taken.
- **libs** (Optional[List[str]]) – additional libraries to link with. (default: None)

**Return type**
str

**Returns**
the stdout of the link command

### fab.tools.preprocessor

This file contains the base class for any preprocessor, and two derived classes for cpp and fpp.

### Classes

| | |
|---|---|
| *Cpp*() | Class for cpp. |
| *CppFortran*() | Class for cpp when used as a Fortran preprocessor |
| *Fpp*() | Class for Intel's Fortran-specific preprocessor. |
| *Preprocessor*(name, exec_name, category[, ...]) | This is the base class for any preprocessor. |

**class** fab.tools.preprocessor.**Preprocessor**(*name*, *exec_name*, *category*, *availability_option=None*)

This is the base class for any preprocessor.

**Parameters**

- **name** (str) – the name of the preprocessor.
- **exec_name** (Union[str, Path]) – the name of the executable.
- **category** (*Category*) – the category (C_PREPROCESSOR or FORTRAN_PREPROCESSOR)

**preprocess**(*input_file*, *output_file*, *add_flags=None*)

Calls the preprocessor to process the specified input file, creating the requested output file.

**Parameters**

- **input_file** (Path) – input file.
- **output_file** (Path) – the output filename.
- **add_flags** (Optional[List[Union[str, Path]]]) – List with additional flags to be used. (default: None)

**class** fab.tools.preprocessor.**Cpp**

Class for cpp.

**class** fab.tools.preprocessor.**CppFortran**

Class for cpp when used as a Fortran preprocessor

**class** fab.tools.preprocessor.**Fpp**

Class for Intel's Fortran-specific preprocessor.

### fab.tools.psyclone

This file contains the tool class for PSyclone.

### Classes

| | |
|---|---|
| *Psyclone*() | This is the base class for *PSyclone*. |

**class** fab.tools.psyclone.**Psyclone**

This is the base class for *PSyclone*.

**check_available**()

This function determines if PSyclone is available. Additionally, it established the version, since command line option changes significantly from python 2.5.0 to the next release.

**Return type**
bool

**process**(*config*, *x90_file*, *psy_file=None*, *alg_file=None*, *transformed_file=None*, *transformation_script=None*, *additional_parameters=None*, *kernel_roots=None*, *api=None*)

Run PSyclone with the specified parameters. If PSyclone is used to transform existing Fortran files, *api* must be None, and the output file name is *transformed_file*. If PSyclone is using its DSL features, api must be a valid PSyclone API, and the two output filenames are *psy_file* and *alg_file*.

**Parameters**

- **api** (Optional[str]) – the PSyclone API. (default: None)
- **x90_file** (Path) – the input file for PSyclone
- **psy_file** (Optional[Path]) – the output PSy-layer file. (default: None)
- **alg_file** (Union[Path, str, None]) – the output modified algorithm file. (default: None)
- **transformed_file** (Optional[Path]) – the output filename if PSyclone is called as transformation tool. (default: None)
- **transformation_script** (Optional[Callable[[Path, *BuildConfig*], Path]]) – an optional transformation script (default: None)
- **additional_parameters** (Optional[List[str]]) – optional additional parameters for PSyclone (default: None)
- **kernel_roots** (Optional[List[Union[str, Path]]]) – optional directories with kernels. (default: None)

### fab.tools.rsync

This file contains the Rsync class for synchronising file trees.

### Classes

| | |
|---|---|
| *Rsync*() | This is the base class for *rsync*. |

**class** fab.tools.rsync.**Rsync**

This is the base class for *rsync*.

**execute**(*src*, *dst*)

Execute an rsync command from src to dst. It supports ~ expansion for src, and makes sure that *src* end with a / so that rsync does not create a sub-directory.

**Parameters**

- **src** (Path) – the input path.
- **dst** (Path) – destination path.

### fab.tools.shell

This file contains a base class for shells. This can be used to execute other scripts.

### Classes

| | |
|---|---|
| *Shell*(name) | A simple wrapper that runs a shell script. |

**class** fab.tools.shell.**Shell**(*name*)

A simple wrapper that runs a shell script. There seems to be no consistent way to simply check if a shell is working - not only support a version command (e.g. sh and dash don't). Instead, availability is tested by running a simple 'echo' command.

**Name**
the path to the script to run.

**exec**(*command*)

Executes the specified command.

**Parameters**
**command** (Union[str, List[Union[str, Path]]]) – the command and potential parameters to execute.

**Return type**
str

**Returns**
stdout of the result.

### fab.tools.tool

This file contains the base class for all tools, i.e. compiler, preprocessor, linker, archiver, Psyclone, rsync, versioning tools.

Each tool belongs to one category (e.g. FORTRAN_COMPILER). This category is used when adding a tool to a ToolRepository or ToolBox. It provides basic support for running a binary, and keeping track if a tool is actually available.

### Classes

| | |
|---|---|
| *CompilerSuiteTool*(name, exec_name, suite, ...) | A tool that is part of a compiler suite (typically compiler and linker). |
| *Tool*(name, exec_name[, category, ...]) | This is the base class for all tools. |

**class** fab.tools.tool.**Tool**(*name*, *exec_name*, *category=Category.MISC*, *availability_option=None*)

This is the base class for all tools. It stores the name of the tool, the name of the executable, and provides a *run* method.

**Parameters**

- **name** (str) – name of the tool.
- **exec_name** (Union[str, Path]) – name or full path of the executable to start.
- **category** (*Category*) – the Category to which this tool belongs. (default: <Category.MISC: 13>)
- **availability_option** (Union[str, List[str], None]) – a command line option for the tool to test if the tool is available on the current system. Defaults to *–version*. (default: None)

**check_available**()

Run a 'test' command to check if this tool is available in the system. :rtype: bool :returns: whether the tool is working (True) or not.

**set_full_path**(*full_path*)

This function adds the full path to a tool. This allows tools to be used that are not in the user's PATH. The ToolRepository will automatically update the path for a tool if the user specified a full path.

**Parameters**
**full_path** (Path) – the full path to the executable.

**property is_available: bool**

Checks if the tool is available or not. It will call a tool-specific function check_available to determine this, but will cache the results to avoid testing a tool more than once.

**Returns**
whether the tool is available (i.e. installed and working).

**property is_compiler: bool**

Returns whether this tool is a (Fortran or C) compiler or not.

**property exec_path: Path**

**Returns**
the path of the executable.

**property exec_name: str**

**Returns**
the name of the executable.

**property name: str**

**Returns**
the name of the tool.

**property availability_option: str | List[str]**

**Returns**
the option to use to check if the tool is available.

**property category: *Category***

**Returns**
the category of this tool.

**get_flags**(*profile=None*)

**Returns**
the flags to be used with this tool.

**add_flags**(*new_flags*, *profile=None*)

Adds the specified flags to the list of flags.

**Parameters**
**new_flags** (Union[str, List[str]]) – A single string or list of strings which are the flags to be added.

**define_profile**(*name*, *inherit_from=None*)

Defines a new profile name, and allows to specify if this new profile inherit settings from an existing profile.

**Parameters**

- **name** (str) – Name of the profile to define.
- **inherit_from** (Optional[str]) – Optional name of a profile to inherit settings from. (default: None)

**property logger: Logger**

**Returns**
a logger object for convenience.

**run**(*additional_parameters=None*, *profile=None*, *env=None*, *cwd=None*, *capture_output=True*)

Run the binary as a subprocess.

**Parameters**

- **additional_parameters** (Union[str, Sequence[Union[Path, str]], None]) – List of strings or paths to be sent to subprocess.run() as additional parameters for the command. Any path will be converted to a normal string. (default: None)
- **env** (Optional[Dict[str, str]]) – Optional env for the command. By default it will use the current session's environment. (default: None)
- **capture_output** – If True, capture and return stdout. If False, the command will print its output directly to the console. (default: True)

**Raises**

- **RuntimeError** – if the code is not available.
- **RuntimeError** – if the return code of the executable is not 0.

**Return type**
str

class fab.tools.tool.**CompilerSuiteTool**(*name*, *exec_name*, *suite*, *category*, *availability_option=None*)

A tool that is part of a compiler suite (typically compiler and linker).

**Parameters**

- **name** (str) – name of the tool.
- **exec_name** (Union[str, Path]) – name of the executable to start.
- **suite** (str) – name of the compiler suite.
- **category** (*Category*) – the Category to which this tool belongs.
- **availability_option** (Union[str, List[str], None]) – a command line option for the tool to test if the tool is available on the current system. Defaults to *–version*. (default: None)

property **suite**: str

**Returns**
the compiler suite of this tool.

### fab.tools.tool_box

This file contains the ToolBox class.

### Classes

| | |
|---|---|
| *ToolBox*() | This class implements the tool box. |

class fab.tools.tool_box.**ToolBox**

This class implements the tool box. It stores one tool for each category to be used in a FAB build.

**add_tool**(*tool*, *silent_replace=False*)

Adds a tool for a given category.

**Parameters**

- **tool** (*Tool*) – the tool to add.
- **silent_replace** (bool) – if set, no warning will be printed if an existing tool is replaced. (default: False)

**Raises**
RuntimeError – if the tool to be added is not available.

**Return type**
None

get_tool(*category*, *mpi=None*, *openmp=None*)

Returns the tool for the specified category.

**Parameters**

- **category** (*Category*) – the name of the category in which to look for the tool.
- **mpi** (Optional[bool]) – if no compiler or linker is explicitly specified in this tool box, use the MPI and OpenMP setting to find an appropriate default from the tool repository. (default: None)
- **mpi** – if no compiler or linker is explicitly specified in this tool box, use the MPI and OpenMP setting to find an appropriate default from the tool repository.

**Raises**
KeyError – if the category is not known.

**Return type**
*Tool*

### fab.tools.tool_repository

This file contains the ToolRepository class.

### Classes

| | |
|---|---|
| *ToolRepository*() | This class implements the tool repository. |

class fab.tools.tool_repository.**ToolRepository**

This class implements the tool repository. It stores a list of tools for various categories. For each compiler, it will automatically create a tool called "linker-{compiler-name}" which can be used for linking with the specified compiler.

Singleton access. Changes the value of _singleton so that the constructor can verify that it is indeed called from here.

add_tool(*tool*)

Creates an instance of the specified class and adds it to the tool repository. If the tool is a compiler, it automatically adds the compiler as a linker as well (named "linker-{tool.name}").

**Parameters**
**tool** (*Tool*) – the tool to add.

get_tool(*category*, *name*)

This functions returns a tool with a given name. The name can either be a Fab compiler name (including wrapper naming), e.g. mpif90-gfortran, or linker-mpif90-ifort, or just the name of the executable (mpif90). If a Fab name is specified, the corresponding tool will be returned, even if it should not be available (allowing default site scripts to setup any compiler, even if they are not available everywhere). If an exec name is specified, the tool must be available. This is required to make sure the user gets the right tool: by specifying just mpif90, it is not clear if the user wants mpif90-ifort, mpif90-gfortran, … . But only one of these tools will actually be available (the wrapper checks the version number to detect the compiler-vendor).

The name can also be specified using an absolute path, in which case only the stem will be used to look up the name, but the returned tool will be updated to use the full path to the tool. This allows the usage of e.g. compilers that are not in $PATH of the user.

**Return type**
*Tool*

**Returns**
the tool with a given name in the specified category.

**Parameters**

- **category** (*Category*) – the name of the category in which to look for the tool.
- **name** (str) – the name of the tool to find. A full path can be used, in which case only the stem of the path is used, and the tool will be updated to use the absolute path specified.

**Raises**

- **KeyError** – if there is no tool in this category.
- **KeyError** – if no tool in the given category has the requested name.

`set_default_compiler_suite`(*suite*)

Sets the default for linker and compilers to be of the given compiler suite.

**Parameters**
**suite** (str) – the name of the compiler suite to make the default.

`get_default`(*category*, *mpi=None*, *openmp=None*)

Returns the default tool for a given category that is available. For most tools that will be the first entry in the list of tools. The exception are compilers and linker: in this case it must be specified if MPI support is required or not. And the default return will be the first tool that either supports MPI or not.

**Parameters**

- **category** (*Category*) – the category for which to return the default tool.
- **mpi** (Optional[bool]) – if a compiler or linker is required that supports MPI. (default: None)
- **openmp** (Optional[bool]) – if a compiler or linker is required that supports OpenMP. (default: None)

**Raises**

- **KeyError** – if the category does not exist.
- **RuntimeError** – if no tool in the requested category is available on the system.
- **RuntimeError** – if no compiler/linker is found with the requested level of MPI support (yes or no).

### fab.tools.versioning

Versioning tools such as Subversion and Git.

### Classes

| | |
|---|---|
| *Fcm*() | This is the base class for FCM. |
| *Git*() | Interface to Git version control system. |
| *Subversion*([name, exec_name, category]) | Interface to the Subversion version control system. |
| *Versioning*(name, exec_name, category) | Base class for versioning tools like Git and Subversion. |

**class** fab.tools.versioning.**Versioning**(*name*, *exec_name*, *category*)

Base class for versioning tools like Git and Subversion.

Constructor.

**Parameters**

- **name** (str) – Display name of this tool.
- **exec_name** (Union[str, Path]) – Executable for this tool.
- **category** (*Category*) – Tool belongs to this category.

**class** fab.tools.versioning.**Git**

Interface to Git version control system.

Constructor.

**Parameters**

- **name** – Display name of this tool.
- **exec_name** – Executable for this tool.
- **category** – Tool belongs to this category.

**current_commit**(*folder=None*)

**Return type**
str

**Returns**
the hash of the current commit.

**Parameters**
**folder** (Union[Path, str, None]) – the folder for which to determine the current commitf (defaults to .). (default: None)

**init**(*folder*)

Initialises a directory.

**Parameters**
**folder** (Union[Path, str]) – the directory to initialise.

**clean**(*folder*)

Removes all non versioned files in a directory.

**Parameters**
**folder** (Union[Path, str]) – the directory to clean.

**fetch**(*src*, *dst*, *revision*)

Runs *git fetch* in the specified directory

**Parameters**

- **src** (Union[str, Path]) – the source directory from which to fetch

- **revision** (Optional[str]) – the revision to fetch (can be "" for latest revision)
- **dst** (Union[str, Path]) – the directory in which to run fetch.

**checkout**(*src*, *dst=''*, *revision=None*)

Checkout or update a Git repo.

**Parameters**

- **src** (str) – the source directory from which to checkout.
- **dst** (str) – the directory in which to run checkout. (default: '')
- **revision** (Optional[str]) – the revision to check out (can be "" for latest revision). (default: None)

**merge**(*dst*, *revision=None*)

Merge a git repo into a local working copy. If the merge fails, it will run *git merge –abort* to clean the directory.

**Parameters**

- **dst** (Union[str, Path]) – the directory to merge in.
- **revision** (Optional[str]) – the revision number (only used for error message, it relies on git fetch running previously). (default: None)

class fab.tools.versioning.**Subversion**(*name=None*, *exec_name=None*, *category=Category.SUBVERSION*)

Interface to the Subversion version control system.

Constructor.

This is class is extended by the FCM interface which is why name and executable are mutable.

**Parameters**

- **name** (Optional[str]) – Tool name, defaults to "subversion." (default: None)
- **exec_name** (Union[Path, str, None]) – Tool executable, defaults to "svn." (default: None)
- **category** (*Category*) – Tool category, defaults to SUBVERSION. (default: <Category.SUBVERSION: 9>)

**execute**(*pre_commands=None*, *revision=None*, *post_commands=None*, *env=None*, *cwd=None*, *capture_output=True*)

Executes a svn command.

**Parameters**

- **pre_commands** (Optional[List[str]]) – List of strings to be sent to subprocess.run() as the command. (default: None)
- **revision** (Union[int, str, None]) – optional revision number as argument (default: None)
- **post_commands** (Optional[List[str]]) – List of additional strings to be sent to subprocess.run() after the optional revision number. (default: None)
- **env** (Optional[Dict[str, str]]) – Optional env for the command. By default it will use the current session's environment. (default: None)
- **capture_output** – If True, capture and return stdout. If False, the command will print its output directly to the console. (default: True)

**Return type**
str

**export**(*src*, *dst*, *revision=None*)

Runs svn export.

**Parameters**

- **src** (Union[str, Path]) – from where to export.
- **dst** (Union[str, Path]) – destination path.
- **revision** (Optional[str]) – revision to export. (default: None)

**checkout**(*src*, *dst*, *revision=None*)

Runs svn checkout.

**Parameters**

- **src** (Union[str, Path]) – from where to check out.
- **dst** (Union[str, Path]) – destination path.
- **revision** (Optional[str]) – revision to check out. (default: None)

**update**(*dst*, *revision=None*)

Runs svn checkout.

**Parameters**

- **dst** (Union[str, Path]) – destination path.
- **revision** (Optional[str]) – revision to check out. (default: None)

**merge**(*src*, *dst*, *revision=None*)

Runs svn merge.

**Parameters**

- **src** (Union[str, Path]) – the src URI.
- **dst** (Union[str, Path]) – destination path.
- **revision** (Optional[str]) – revision to check out. (default: None)

**class** fab.tools.versioning.**Fcm**

This is the base class for FCM. All commands will be mapped back to the corresponding subversion commands.

Constructor.

This is class is extended by the FCM interface which is why name and executable are mutable.

**Parameters**

- **name** – Tool name, defaults to "subversion."
- **exec_name** – Tool executable, defaults to "svn."
- **category** – Tool category, defaults to SUBVERSION.

### fab.util

Various utility functions live here - until we give them a proper place to live!

### Functions

| | |
|---|---|
| *by_type*(iterable, cls) | Find all the elements of an iterable which are of a given type. |
| *common_arg_parser*() | A helper function returning an argument parser with common, useful arguments controlling command line tools. |
| *file_checksum*(fpath) | Return a checksum of the given file. |
| *file_walk*(path[, ignore_folders]) | Return every file in *path* and its sub-folders. |
| *get_fab_workspace*() | Read the Fab workspace from the *FAB_WORKSPACE* environment variable, defaulting to ./fab-workspace. |
| *get_prebuild_file_groups*(prebuild_files) | Group prebuild filenames by originating artefact. |
| *input_to_output_fpath*(config, input_path) | Convert a path in the project's source folder to the equivalent path in the output folder. |
| *log_or_dot*(logger, msg) | Util function which prints a fullstop without a newline, except in debug logging where it logs a message. |
| *log_or_dot_finish*(logger) | Util function which completes the row of fullstops from *log_or_dot()*, by printing a newline when not in debug logging. |
| *string_checksum*(s) | Return a checksum of the given string. |
| *suffix_filter*(fpaths, suffixes) | Pull out all the paths with a given suffix from an iterable. |

### Classes

| | |
|---|---|
| *CompiledFile*(input_fpath, output_fpath) | A Fortran or C file which has been compiled. |
| *HashedFile*(fpath, file_hash) | Create new instance of HashedFile(fpath, file_hash) |
| *Timer*() | A simple timing context manager. |
| *TimerLogger*(label[, res]) | A labelled timing context manager which logs the label and the time taken. |

fab.util.**log_or_dot**(*logger*, *msg*)

Util function which prints a fullstop without a newline, except in debug logging where it logs a message.

fab.util.**log_or_dot_finish**(*logger*)

Util function which completes the row of fullstops from *log_or_dot()*, by printing a newline when not in debug logging.

**class** fab.util.**HashedFile**(*fpath*, *file_hash*)

Create new instance of HashedFile(fpath, file_hash)

**file_hash**

Alias for field number 1

**fpath**

Alias for field number 0

fab.util.**file_checksum**(*fpath*)

Return a checksum of the given file.

This function is deterministic, returning the same result across Python invocations.

We use crc32 for now because it's deterministic, unlike out-the-box hash. We could seed hash with a non-random or look into hashlib, if/when we want to improve this.

fab.util.**string_checksum**(*s*)

Return a checksum of the given string.

This function is deterministic, returning the same result across Python invocations.

We use crc32 for now because it's deterministic, unlike out-the-box hash. We could seed hash with a non-random or look into hashlib, if/when we want to improve this.

fab.util.**file_walk**(*path*, *ignore_folders=None*)

Return every file in *path* and its sub-folders.

**Parameters**

- **path** (Union[str, Path]) – Folder to iterate.
- **ignore_folders** (Optional[List[Path]]) – Pass in any folder if you don't want to traverse into. Please see explanation and intended use, below. (default: None)

**Return type**
Iterator[Path]

**Note**

The prebuild folder can contain multiple versions of a single, generated fortran file, created by multiple runs of the build config. The prebuild folder stores these copies for when they're next needed, when they are copied out and reused. We usually won't want to include this folder when searching for source code to analyse. To meet these needs, this function will not traverse into the given folders, if provided.

**class** fab.util.**Timer**

A simple timing context manager.

**class** fab.util.**TimerLogger**(*label*, *res=0.001*)

A labelled timing context manager which logs the label and the time taken.

**class** fab.util.**CompiledFile**(*input_fpath*, *output_fpath*)

A Fortran or C file which has been compiled.

**Parameters**

- **input_fpath** – The file that was compiled.
- **output_fpath** – The object file that was created.

fab.util.**input_to_output_fpath**(*config*, *input_path*)

Convert a path in the project's source folder to the equivalent path in the output folder.

Allows the given path to already be in the output folder.

**Parameters**

- **config** – The config object, which defines the source and output folders.
- **input_path** (Path) – The path to transform from input to output folders.

Note: This function can also handle paths which are not in the project workspace at all. This can happen when pointing the FindFiles step elsewhere, for example. In that case, the entire path will be made relative to the source folder instead of its anchor.

fab.util.**suffix_filter**(*fpaths*, *suffixes*)

Pull out all the paths with a given suffix from an iterable.

**Parameters**

- **fpaths** (Iterable[Path]) – Iterable of paths.
- **suffixes** (Iterable[str]) – Iterable of suffixes we want.

fab.util.**by_type**(*iterable*, *cls*)

Find all the elements of an iterable which are of a given type.

**Parameters**

- **iterable** – The iterable to search.
- **cls** – The type of the elements we want.

fab.util.**get_fab_workspace**()

Read the Fab workspace from the *FAB_WORKSPACE* environment variable, defaulting to ./fab-workspace.

**Return type**
Path

fab.util.**get_prebuild_file_groups**(*prebuild_files*)

Group prebuild filenames by originating artefact.

Prebuild filenames have the form <*stem*>.<*hash*>.<*suffix*>. This function creates a dict with wildcard key <*stem*>.*.<*suffix*> with each entry mapping to a set of all matching prebuild files.

Given the input files *my_mod.123.o* and *my_mod.456.o*, returns a dict {'my_mod.*.o': {'my_mod.123.o', 'my_mod.456.o'}}

**Return type**
Dict[str, Set]

fab.util.**common_arg_parser**()

A helper function returning an argument parser with common, useful arguments controlling command line tools.

More arguments can be added. The caller must call *parse_args* on the returned parser.

**Return type**
ArgumentParser

## 4.12 Developer's guide

Interested in developing Fab? Here are some resources to help you on your way.

### 4.12.1 Install from source

The following commands will checkout the latest version of the code and create an editable install.

This lets you edit the code without needing to reinstall fab after every change.

```
$ git clone https://github.com/MetOffice/fab.git <fab-folder>
$ pip install -e <fab-folder>
```

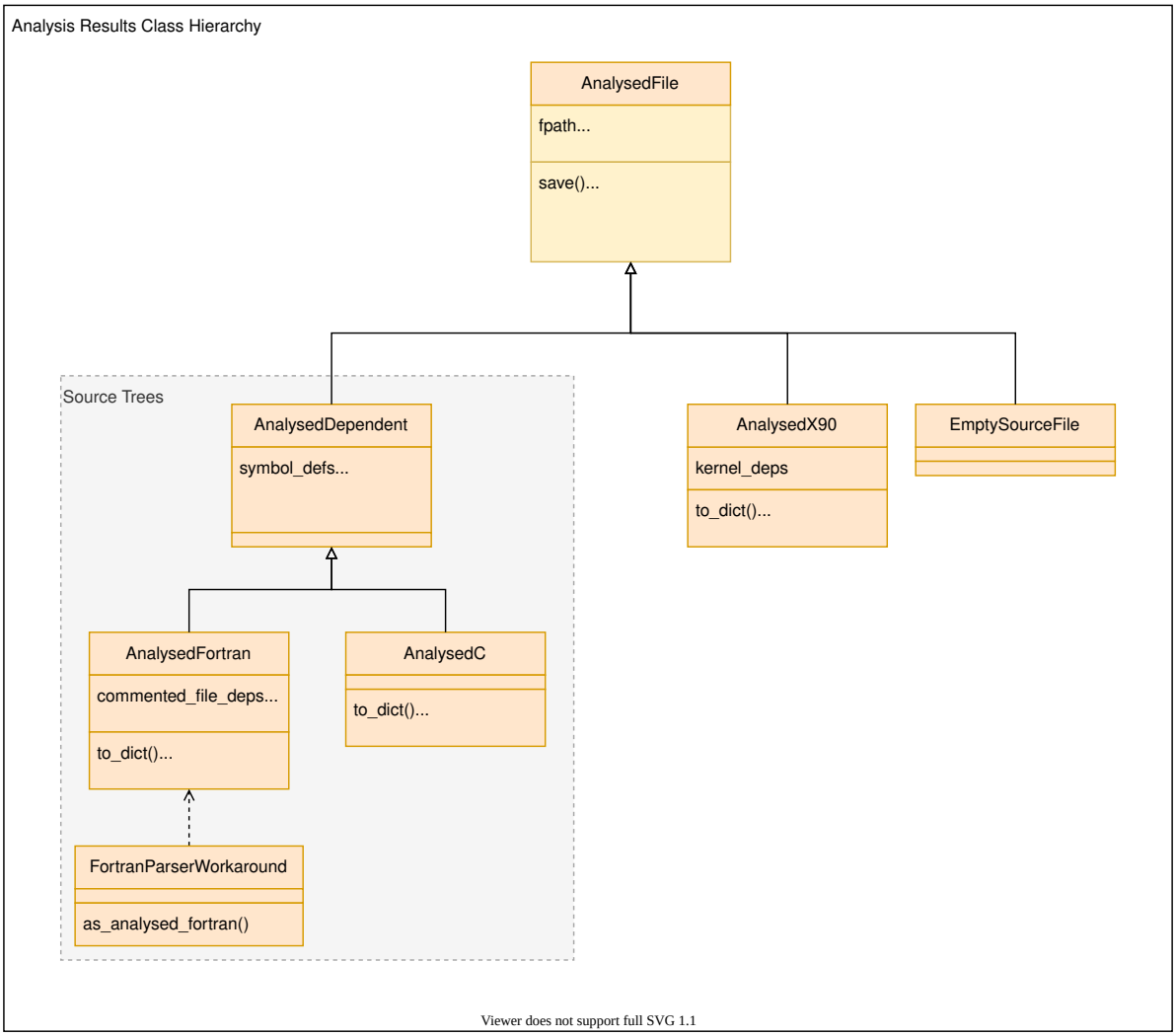
You can install extra features by using [test], [docs], [features] or [dev], as defined in setup.py.

```
$ pip install -e <fab-folder>[dev]
```

### 4.12.2 Source code Analysis

The class hierarchy for analysis results can be seen below. See `analyse` for a description of the analysis process.

Classes which are involved in source tree analysis contain symbol definitions and dependencies, and the file dependencies into which they are converted.




### 4.12.3 Incremental & Prebuilds

See *Incremental Build* and *Prebuild* for definitions.

Prebuilt artefacts are stored in a flat *_prebuild* folder underneath the *build_output* folder. They include a checksum in their filename to distinguish between different builds of the same artefact. All prebuild files are named: *<stem>.<hash>.<suffix>*, e.g: *my_mod.123.o*.

#### Checksums

Fab inserts a checksum in the names of prebuild files. This checksum is derived from everything which should trigger a rebuild if changed. Before an artefact is created, Fab will calculate the checksum and search for an existing artefact so it can avoid reprocessing the inputs.

#### Analysis results

Analysis results are stored in files with a *.an* suffix. The checksum in the filename is solely the hash of the analysed source file. Note: this can change with different preprocessor flags.

#### Fortran module files

When creating a module file from a Fortran source file, the prebuild checksum is created from hashes of:

- source file
- compiler
- compiler version

#### Fortran object files

When creating an object file from a Fortran source file, the prebuild checksum is created from hashes of:

- source file
- compiler
- compiler version
- compiler flags
- modules on which the source depends

### 4.12.4 Running the tests

You'll need to install from source, and a full *[dev] install* to get the testing dependencies.

#### Unit and system tests

From the fab folder, type:

```
$ pytest tests/unit_tests
$ pytest tests/system_tests
```

#### Flake8 and mypy

When making a PR, you might want to run all the checks which give us green ticks. You can see the commands we run in `.github/workflows/build.yml`.

To run flake8 and mypy, type:

```
$ flake8 .
$ mypy setup.py source tests
```

### 4.12.5 Github Actions

Various workflows are maintained by the project.

#### Testing a PR

The github action defined in `.github/workflows/build.yml` automatically runs the unit and system tests, plus flake8 and mypy, and adds green ticks to pull requests.

#### Build these docs

The github action to build the docs is defined in `.github/workflows/build_docs.yml`. It is manually triggered and can be run from any branch in the MetOffice repo.

You can also run it on your fork to produce a separate build, for viewing work in progress.

### 4.12.6 Version numbering

We use a PEP 440 compliant semantic versioning, of the form `{major}.{minor}.{patch}[{a|b|rc}N]`

- 0.9.5
- 1.0.0a1
- 1.0.0a2
- 1.0.0b1
- 1.0.0rc1
- 1.0.0
- 1.0.1
- 1.1.0a1

Dev versions are not for release and cover multiple commits. * 1.0.dev0 * … * 1.0.0 * 1.0.dev1 * … * 1.0.1

The version number is defined in `source/fab/__init_.py`.

### 4.12.7 Developing at the Met Office

There are special notes for *developers working at the Met Office*.

## 4.13 Glossary

**Artefact**
: *Artefact* is a term for an item, usually a file, created by a build step and added to the *Artefact Store*.

  **Some examples of artefacts are:**

  - a Fortran file
  - a compiled object file
  - an executable file

  These exist as files in the file system and paths in the *Artefact Store*.

**Artefact Collection**
: A collection of *Artefacts* in the *Artefact Store*.

  These entries are usually a list of file paths or objects, sometimes a *Source Tree*, but could be anything created by one step and consumed by another.

  As an example, a Fortran preprocessing step might create a list of output paths as `artefact_store['preprocessed fortran'] = my_results`. A subsequent step could read this list.

**Artefacts Getter**
: A helper class which a *Step* uses to find artefacts in the *Artefact Store*. Fab's built-in steps come with sensible defaults so the user doesn't have to write unnecessary config.

As an example, the Fortran preprocessor has a default artefact getter which reads *".F90"* files from the *Artefact Collection* called `"INITIAL_SOURCE"`.

Artefact getters are derived from `ArtefactsGetter`.

**Artefact Store**
The artefact store holds *collections* created and used by build steps.

Fab passes the growing store to each step in turn, where they typically read a collection and create a new one for the next step.

**Build Tree**
A mapping of filenames to Analysis results. This subset of the *Source Tree* contains only the files needed to build one target (*Root Symbol*). It's created in the analysis step and used by the Fortran compilation step. When building a library, there is no root symbol and the entire source tree is included in a single build tree.

**Fab Workspace**
The folder in which all Fab output is created, for all build projects. Defaults to *~/fab-workspace*, and can be overridden by the `$FAB_WORKSPACE` environment variable or the *fab_workspace* argument to the `BuildConfig` constructor. See also *Configure the Fab Workspace*

**Incremental Build**
This term refers to Fab's ability to avoid reprocessing an artefact if the output is already available. For example, if the user has previously built the project, there will likely be object files Fab can use to avoid recompilation.

**Prebuild**
This term has much overlap with the term *Incremental Build*. It refers to artefacts that were built by another user, which can be copied to avoid reprocessing artefacts. Technical details on how this works can be found in *Incremental & Prebuilds*.

**Project Workspace**
A folder inside the *Fab Workspace*, containing all source and output from a build config.

**Root Symbol**
The name of a Fortran PROGRAM, or `"main"` for C code. Fab builds an executable for every root symbol it's given.

**Source Tree**
The `analyse` step produces a dependency tree of the entire project source. This is represented as a mapping from Path to `AnalysedDependent`. The AnalysedDependent's file dependencies are Paths, which refer to other entries in the mapping, and which define the tree structure. This is called the source tree.

**Step**
A step performs a function in the build process.

Each step typically reads from, and adds to, an in-memory *Artefact* repository called the *Artefact Store*. Steps are derived from the `Step` base class.

## 4.14 Index

A hack to get the general index into the tox sidebar. https://stackoverflow.com/questions/36235578/how-can-i-include-the-genindex-in-a-sphinx-toc

## 4.15 Python Module Index

A hack to get the general index into the tox sidebar. https://stackoverflow.com/questions/36235578/how-can-i-include-the-genindex-in-a-sphinx-toc

# PYTHON MODULE INDEX

## f

# INDEX

## D

## E



## F

## G

## H



## I

## L



## M

## N



## O



## P



## R

## S

## T



## U



## V



## W



## X