Hacking the fallbacks

IMPORTANT NOTE
We only provide support for the fallbacks delivered through the Abinit Website. If you modify them, we suppose that you know what you are doing and are a sufficiently skilled programmer to solve most issues by yourself. We may be able to help you to a limited extent, but you have to remember that we are developers of Abinit, NOT of its fallbacks. An infinitely better way to address your issues will be to collaborate directly with the developers of the relevant projects. Some of us can however send you a quote if working with Abinit developers is mandatory for you.

Prerequisites

In order to be able to modify the Abinit Fallbacks, you need at least the following skill set:

- a strong experience in developing Python scripts;
- a good experience in building and fixing packages that use the GNU Autotools;
- some experience in writing portable shell scripts;
- some experience in writing M4 macros;
- some experience in using Git.

Unless you are an Abinit Developer, we will not provide any support regarding these topics, which are widely documented on many websites. Wikipedia is a good starting point.

The best way to make the fallbacks evolve towards a direction useful to you is to join the Abinit Project.

By joining the team, you commit to:

- take full care of all the data related to the packages you add to the list, including long-term maintenance;
- provide support for all the packages you are responsible for and accept support requests from the users of Abinit;
- improve the visibility and availability of the packages you are responsible for, so that they can be used as fully independent packages instead of fallbacks;
- participate to on-line and in-person meetings to globally improve the quality of Abinit and its surrounding ecosystem;
- accept that orphan packages have to be removed to maintain the overall quality of the whole.

In exchange, you will receive help, training and collaboration from other Abinit Developers with all the issues you may encounter.

Please note that if you do not join the team, you will have to maintain your modified versions of the fallbacks on your own, including synchronization with the trunk.
Quality standards

In order to be integrated into the fallbacks, a package has to follow a minimum set of coding standards:

- be free software, in the GNU sense;
- work stand-alone and be downloadable from its own webpage;
- have an Autotools-based or Cmake-based build system;
- have a test suite with sufficient coverage;
- accept debug flags;
- be sufficiently documented to be installed and used by beginners;
- have its own active support team, fully independent of Abinit and its fallbacks;
- be actually developed and maintained.

In particular, at no time may the Abinit Fallbacks be considered as a substitute for the distribution of undocumented and untested low-quality code with support to be delegated to the Abinit Developers. And since the Abinit Test Farm will be used for the validation of its portability, adopting a design strategy oriented towards portability and interoperability will greatly ease the integration of your package with the already existing fallbacks.

Another important aspect is that a package integrated within the Abinit Fallbacks must always remain available as a fully autonomous and stand-alone package at the same time. In other words, the Abinit Fallbacks cannot substitute the normal distribution of a package, even if it comes out of Abinit. Many mature and reliable services such as Gitlab and GitHub make it extremely easy to distribute free software and provide much broader services than a ad-hoc simplified embedding system could ever do.

Last but not least, the package must follow some conventions when installing files:

- allow for program suffixes;
- be able to install executables in `PREFIX/bin`;
- be able to install C headers and Fortran modules in `PREFIX/include`;
- be able to install libraries in `PREFIX/lib`;
- allow for staged installs, i.e. `$(DESTDIR)/PREFIX`;

where `PREFIX` is the top install directory of the package. These conditions are usually fulfilled when using standard build systems.

Procedure

There is a standard procedure to improve the Abinit Fallbacks. It involves different levels of validation and may take a significant amount of time before the new fallbacks are fully deployed. Please expect significant delays and be prepared to contribute with the Test Farm Team all along the process. Once a new package has been successfully installed, providing newer versions is relatively faster.

Please note that you have first to request an Abinit Fallbacks repository access to the Test Farm Team before starting to contribute to the fallbacks.
Here are the different steps:

1. Clone the Abinit Fallbacks repository.
2. Submit an issue to describe your work and discuss it if necessary.
3. Create a new branch.
4. Implement the desired modifications.
5. Run `./quicktest.sh` from the top source directory and correct the errors until it succeeds.
6. Install the new fallbacks on your computer and test Abinit with them.
7. Go back to step 5 until step 6 succeeds.
8. Submit a merge request to the master branch, in order to trigger additional tests.
9. Once merged, set your Abinit branch to use the new fallbacks and push it.

**IMPORTANT NOTE**

Please note that this full procedure has to be performed for any of the modifications described in the sections below.

---

### Updating the version of a package

To update the version of a package, the only required modification in the fallbacks source tree is to update the self-documented `~fallbacks/config/specs/fallbacks.conf` file:

- find the section where the package is described;
- update all fields depending on its version number;
- calculate the MD5 sum of the source tarball and update the corresponding field.

For more information, please read the instructions contained in the file itself.

The more important steps to take are:

- coordinating with the other members of the fallbacks developers team, in order to synchronize the possible update of other fallbacks;
- coordinating with the test farm maintainers for the deployment of the new fallbacks;
- coordinating with the gatekeeper of Abinit to set a schedule for the activation of the new version of the external package;
- informing the developers of Abinit through the Abinit Forums that a new version of the package will be deployed.

All these steps should of course be taken as early as possible, in order to make the changes as smooth as possible for everybody. Ideally, informing and coordinating should be done when starting the project or its next development cycle, with an early reminder as the release date approaches.

### Enhancing the portability of a package

The most valuable service the Abinit Fallbacks provide to the users of Abinit comes through **portability tricks**. Every fallback must have them. These tricks are located in M4 macros, one for
each package, in the following file:

- `~fallbacks/config/m4/conf-tricks.m4`

Let us have a detailed look at one of the historical macros:

```
# AFB_TRICKS_ETSF_IO(FC_VENDOR,FC_VERSION)
# ----------------------------------------
# Applies tricks and workarounds to have the ETSF I/O library correctly
# linked to the binaries.
#
AC_DEFUN([AFB_TRICKS_ETSF_IO],[
  dnl Do some sanity checking of the arguments
  m4_if([$1], , [AC_FATAL([$0: missing argument 1])])dnl
  m4_if([$2], , [AC_FATAL([$0: missing argument 2])])dnl

  dnl Init
  afb_etsf_io_tricks="no"
  afb_etsf_io_tricky_vars=""
  tmp_etsf_io_num_tricks=2
  tmp_etsf_io_cnt_tricks=0

  dnl Configure tricks
  if test "${afb_etsf_io_cfgflags_custom}" = "no"; then
    AC_MSG_NOTICE([applying ETSF_IO tricks (vendor: $1, version: $2, flags: config)])
    
    dnl NetCDF
    tmpflags_etsf_io='--with-netcdf-incs="$(afb_netcdf_incs)"
    CFGFLAGS_ETSF_IO="${CFGFLAGS_ETSF_IO} ${tmpflags_etsf_io}"
    tmpflags_etsf_io='--with-netcdf-libs="$(afb_netcdf_libs)"
    CFGFLAGS_ETSF_IO="${CFGFLAGS_ETSF_IO} ${tmpflags_etsf_io}"
    
    dnl Internal ETSF_IO parameters
    tmpflags_etsf_io='--with-moduledir="$(prefix)/include"
    CFGFLAGS_ETSF_IO="${CFGFLAGS_ETSF_IO} ${tmpflags_etsf_io}"

    dnl Finish
    tmp_etsf_io_cnt_tricks=`expr ${tmp_etsf_io_cnt_tricks} \+ 1`
    afb_etsf_io_tricky_vars="${afb_etsf_io_tricky_vars} CFGFLAGS"
    unset tmpflags_etsf_io
  else
    AC_MSG_NOTICE([CFGFLAGS_ETSF_IO set => skipping ETSF_IO config tricks])
  fi

  dnl Fortran tricks
  if test "${afb_etsf_io_fcflags_custom}" = "no"; then
    AC_MSG_NOTICE([applying ETSF_IO tricks (vendor: $1, version: $2, flags: Fortran)])
```
case "$1" in
  ibm)
    FCFLAGS_ETSF_IO="${FCFLAGS_ETSF_IO} -qsuffix=cpp=f90:f=f"
    ;;
esac

dnl Finish
  tmp_etsf_io_cnt_tricks=`expr ${tmp_etsf_io_cnt_tricks} \+ 1`
  afb_etsf_io_tricky_vars="${afb_etsf_io_tricky_vars} FCFLAGS"
else
  AC_MSG_NOTICE([FCFLAGS_ETSF_IO set => skipping ETSF_IO Fortran tricks])
fi

dnl Count applied tricks
  case "${tmp_etsf_io_cnt_tricks}" in
    0)
      afb_etsf_io_tricks="no"
      ;;
    *{tmp_etsf_io_num_tricks}
      afb_etsf_io_tricks="yes"
      ;;
    *)
      afb_etsf_io_tricks="partial"
      ;;
esac
  unset tmp_etsf_io_cnt_tricks
  unset tmp_etsf_io_num_tricks
) # AFB_TRICKS_ETSF_IO

Each macro is named “AFB_TRICKS_PACKAGE”, where PACKAGE is the upper-case name of the package, with possible spaces replaced by underscores. It expects 2 arguments, provided automatically by the build system of the fallbacks:

- the vendor name of the Fortran compiler (e.g. gnu, ibm, intel);
- the version number of the Fortran compiler, with the format x.y (e.g. 4.9, 14.1, 15.0).

These arguments are automatically provided to the macro and respectively available under the $1 and $2 variables.

After a short section dedicated to initialization, the macro applies portability tricks in a strictly categorized manner and counts them one by one. Global variables used by other M4 macros and the configure script are prefixed with afb_, while variables internal to the macro are prefixed with tmp_. Two important variables must be defined there: tmp_package_num_tricks, which tells how many tricks have to be applied for the tricks to be complete, and tmp_package_cnt_tricks, set to 0, which counts how many tricks have been applied so far. They are instrumental in letting the user know whether some custom flags have been provided and quite drastically reduce user support requests due to wrongly set environment variables. To complement the information, the afb_package_tricky_flags variable lists the flags that have been enhanced by the tricks.

The tricks are categorized using the different stages of a typical build. For each category, a specific set of build flags is set. The following table summarizes the categories of tricks and associated flags:
<table>
<thead>
<tr>
<th>Build stage</th>
<th>Flags to set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configure</td>
<td>CFGFLAGS_PACKAGE</td>
</tr>
<tr>
<td>C preprocessing</td>
<td>CPPFLAGS_PACKAGE</td>
</tr>
<tr>
<td>C compiling</td>
<td>CFLAGS_PACKAGE</td>
</tr>
<tr>
<td>Fortran preprocessing</td>
<td>FPPFLAGS_PACKAGE</td>
</tr>
<tr>
<td>Fortran compiling</td>
<td>FCFLAGS_PACKAGE</td>
</tr>
<tr>
<td>Link flags</td>
<td>LDFLAGS_PACKAGE</td>
</tr>
<tr>
<td>Library flags</td>
<td>LIBS_PACKAGE</td>
</tr>
</tbody>
</table>

where `PACKAGE` is replaced by the upper-case name of the package, just as for the name of the macro. The `tmp_package_num_tricks` variable must be set to the number of lines in this table used within the macro.

For each category, the tricks can be applied only if the user has not provided custom flags. This is why each sequence of instructions starts with:

```bash
if test "${afb_package_someflags_custom}" = "no"; then
```

where `someflags` is replaced by the lower-case version of the flag type referenced in the above table. The `afb_package_someflags_custom` variables are set and provided by other macros. Please always remember that the priority is to inform the user, which is why all sections start with a `AC_MSG_NOTICE` telling what they will do and have a `else` part telling the user that the tricks have not been applied. Before the `else`, the trick counter is incremented by one in a shell-portable manner.

**IMPORTANT NOTE**

Be careful to always close the `if` block with a `fi` when adding such a section and test modifications to the macro as incrementally as possible, since errors of this kind are usually difficult to diagnose when running the configure script.

If the package has dependencies among the other fallbacks, they will be referenced through the Makefile variables in the configure part of the tricks. Here, since ETSF_IO depends on NetCDF, it has to be linked against NetCDF libraries which can either be external or provided by the fallbacks. This is why this information is transmitted through variables the setting is taken care of by other macros (`afb_netcdf_incs` and `afb_netcdf_libs` here). Please note that, since we reference Makefile variables from a shell script, they have to be both double-quoted and single-quoted, in order not to be interpreted.

The recommended way to apply the Fortran tricks is to use a `case ... esac` block on the Fortran compiler vendor when necessary (first macro argument), with possibly another nested `case ... esac` on the Fortran compiler version (second macro argument).

The best practice is to unset all internal variables of the macro just before returning.

If you wish to add a tricks macro for a new package, we recommend that you copy/paste the closest existing macro, automatically replace the package name in the new macro (both upper-case and lower-case), and modify the code to match the needs of the new package. Please place the new macro in the file following alphabetical order.
Removing a package

Removing a package from the Abinit Fallbacks is always good news, since it means that the package has reached a sufficient maturity and popularity to be distributed with common operating systems, or that dead code has been removed from Abinit.

The operation is very simple:

1. Check that no other fallbacks depend on the package, or remove them at the same time.
2. Remove the package section in ~fallbacks/config/specs/fallbacks.conf.
3. Remove the package macro in ~fallbacks/config/m4/conf-tricks.m4.
4. Run ./quicktest.sh to validate the change.
5. Inform the fallbacks maintainers that the package has been successfully removed.

In principle, the extremely automated build system of the fallbacks should take care of updating all the relevant information when running ./quicktests.sh. If you encounter issues unrelated to the deleted package, please contact the fallbacks developers team as soon as possible.

Adding a new package

**VERY IMPORTANT**
The Abinit Fallbacks have never been designed to replace the work of a skilled system administrator and will completely lose their stability, portability, and genericity, if the dependencies expand over more than 2 levels. This is why all enhancements introducing 3 levels of dependencies between some packages will systematically be refused. Thank you in advance for your understanding.

Adding a new package to the Abinit Fallbacks is never recommended, since they should ideally only be used temporarily for the integration into Abinit of new features depending on fully stand-alone external packages, removing the fallback once the solution is deployed and working. Focusing directly on the availability and usability of the external package will always repay infinitely better than spending time tuning the parameters of a transient crappy embedder.

If you however still wish to add a new package to the fallbacks, the following information is of extreme importance. Since adding a new package will always bring a lot of ripples and side-effects to various components of Abinit, it must always be prepared with a lot of care and as early as possible. In particular, please always check that the following persons are duly informed and given the opportunity to provide feedback before starting:

- the maintainers of the Abinit Fallbacks;
- the gatekeeper(s) of Abinit;
- the maintainers of the Abinit Test Farm;
- the maintainers of the Abinit Build System;
- the maintainers of the Abinit Test Suite;
- the maintainers of the Abinit Input Variables.

They will provide you with useful guidance, as well as helpful information about the delays you can
expect. Please always remember that there are permanently many ongoing projects within Abinit, above all when a new major release of Abinit is scheduled. Patience and dialogue will always bring many more positive outcomes than you could ever expect. In the process, you may be asked to take over a certain number of tasks that were previously performed by other persons. Please make sure that you (or someone else) will be able to reliably take care of them.

Regarding the fallbacks themselves, the only changes you need to perform are very similar to those described in the “Updating the version of a package” and “Improving the portability of a package” sections, except that, this time, you will respectively add a section and a macro in the files instead of changing them. Please follow the alphabetical order to select the positions of the new blocks in the files, in order to facilitate the work of the other developers and maintainers.

Once you have provided the required enhancements, the extremely automated build system of the fallbacks should take care of the rest by itself when running `./quicktests.sh` from the top source directory. If you encounter issues unrelated with the new package, please contact the fallbacks maintainers as soon as possible.

**Patching a package**

Patching the source code of a package should be avoided at all costs. It is always infinitely better to report bugs and improve the package upstream than to produce special versions, since the latter prevents the users of Abinit from using the stand-alone pristine version of the package, depriving them at the same time of any means to correctly validate the package. Worse: if the pristine version is installed on the system, the probability of conflicts and/or inconsistencies is greatly amplified.

If, for some very good reasons (please note the plural), you really need to temporarily patch a package, just copy the patch file to `~/fallbacks/patches/package-version-index.patch`, where you replace *package* by the package name, *version* by its version, and *index* by a four-digit number starting at 0001. The build system of the fallbacks will automatically take care of registering the patches and applying them sequentially to the package source just before configuring it.

*In any case, if you feel the need for a more sophisticated patching mechanism, then it just means that you forgot that what you actually want is to collaborate with the upstream developers of the package to fix its bugs and improve its design.*