xSDK Community Package Policies

The xSDK4ECP Team

Version 1.0.0, February 17, 2023
https://xsdk.info/policies

Background: A key aspect of work in the IDEAS Scientific Software Productivity Project is developing an Extreme-scale Scientific Software Development Kit (xSDK) — a collection of related and complementary software elements that provide the building blocks, tools, models, processes, and related artifacts for rapid and efficient development of high-quality applications. As an initial step in creating the xSDK, we have written the following draft xSDK package community policies to help address challenges in interoperability and sustainability of software developed by diverse groups at different institutions.

Goal: Develop a set of **xSDK community policies** that a software library/framework (henceforth referred to as package)[1] must satisfy in order to be **xSDK compatible**. The designation of a package being xSDK compatible informs potential users that the package can be easily used with other xSDK libraries and components and thus helps to address issues in long-term sustainability[2] and interoperability among packages.

We consider two categories of xSDK packages: **xSDK compatible** packages and **xSDK member** packages. We also consider two levels of xSDK compatibility: **mandatory policies** and **recommended policies**.

- A package will be declared **xSDK compatible** once the xSDK team has determined that the **package satisfies the mandatory xSDK policies** listed below. In addition to the required policies, we specify **recommended xSDK policies** that further help to address issues in software interoperability.

[1] For the purpose of this document, the term *package* refers to a collection of source code (possibly containing C, Fortran, or C++) that can generate zero or more shared or static libraries, zero or more include files, zero or more Fortran modules, and possibly other auxiliary artifacts, including executables, and whose functionality can be used by other packages and by application codes. A software artifact that generates only an executable is, by this definition, not an xSDK package; that is, xSDK packages are libraries, frameworks, and domain components.
[2] See, for example, "Self-Sustaining Software" as outlined in http://trac.trilinos.org/wiki/TribitsLifecycleModelOverview#self_sustaining_software.

- Similarly, a package can become an **xSDK member package** if (1) it is an xSDK-compatible package, *and* (2) it uses or can be used by another package in the xSDK, and the connecting interface is regularly tested for regressions.

Initially the requirements and process are informally presented; over time, if needed, we can begin to formalize them. Currently the xSDK includes twenty-six popular numerical libraries (AMREX, ArborX, ButterflyPACK, DataTransferKit (DTK), deal.II, ExaGO, Ginkgo, heFFTe, HiOp, hypre, libEnsemble, MAGMA, MFEM, Omega_h, PETSc/TAO, PHIST, PLASMA, preCICE, PUMI, SLATE, SLEPc, STRUMPACK, SUNDIALS, SuperLU, TASMANIAN, and Trilinos) and two application packages (Alquimia and PFLOTRAN), which satisfy the required policies. Over the longer term, the xSDK may expand to incorporate additional packages, depending on community needs and contributions.

xSDK Mandatory Policies

M1. Each xSDK-compatible package must support portable installation through Spack. All configuration, build, and installation phases must be specified in a Spack package recipe. Packages should have a build system that is appropriate for the language (e.g., ~ CMake, Autoconf, setup.py), and the packages should attempt to follow the best practices and guidelines of the respective environment. For example, a proper configuration phase should be used where relevant (a list of platform-specific makefiles would not be acceptable). It is recommended that packages have a mechanism for 'smoke' testing (e.g. make test_install). The package should provide Spack variants compliant with the [xSDK Spack variant guidelines]. Packages must not override Spack's resolved dependencies (as determined by `spack spec`). For example, if a BLAS library is part of the concretized `spack spec`, a package cannot silently substitute a different BLAS implementation.The goal of this policy is to enable easy and customizable installation of the package in a way that is compatible with other xSDK packages on the same system.

M2. Each xSDK-compatible package must provide a **comprehensive test suite** that can be run by users and does not require the purchase of commercial software. It is recommended that at least a significant subset of the test suite will complete within a few hours on standard workstation-level hardware. It is also recommended that at least a significant subset of the tests be runnable in batch-only environments, that is, systems that require the use of PBS or other submission scripts.

M3. Each xSDK-compatible package that utilizes MPI must restrict its MPI operations to MPI communicators that are provided to it and not use directly MPI_COMM_WORLD. The package should use configure tests or version tests to detect MPI 3 features that may not be available; it should not be

assumed that a full MPI 3 implementation is available. The package can change the MPI error-handling mode by default but should have an option to prevent it from changing the MPI error handling (which may have been set by another package or the application). The package should also behave appropriately regardless of the MPI error handling being used. There is no requirement that the package provide a sequential (non-MPI) version, although this functionality is welcome, too. If the package provides a sequential version, there is no requirement that it be compatible or usable with other xSDK-compatible packages running without MPI.

M4. Each package team must do a "best effort" at **portability to common platforms**, including standard Linux distributions, and common compiler toolchains such as GNU, Clang,[3] and vendor compilers. **Further portability requirements for xSDK subsets may be conditionally applied based on sponsor requirements**.[4] Support for Apple Mac OS and Microsoft Windows Visual Studio is recommended.

M5. Each package team must provide a documented, reliable way to **contact the development team**; the mode of contact may be by email or a website. The package teams should **not** require users to join a generic mailing list (and hence receive irrelevant email) in order to report bugs or request assistance.

M6. Each package should **respect the decisions made by other previously called packages** regarding system resources and settings**.** For example, each package may provide an API for changing the floating-point exception (FPE) and signal handlers, and even set them in a particular way by default, but there must be a way to prevent the change. Because it is impossible to determine the current state of the FPE and signal handlers and thus restore them to the current state after changing them, it is recommended that the xSDK packages adopt a common protocol for pushing and popping FPE and signal handlers.

M7. The xSDK collaboration has a strong preference for packages to use an OSI-approved, **permissive open-source license** (e.g., MIT or BSD 3-Clause). All new packages will be required to use such a license. Current packages using other licenses are encouraged to relicense, where possible. Required dependencies must use an OSI-approved license that is considered compatible with the preferred permissive licenses for distribution purposes (see

[3] This does not mean that xSDK packages and all their dependencies cannot have a dependency of Fortran, merely that the C and C++ portions of the packages and their dependencies should be compilable with the Clang compilers.
[4] For example, xSDK packages that receive funding from the mathematical libraries component of the U.S. Department of Energy (DOE) Exascale Computing Project (ECP) must support portability to target machines at the computing facilities ALCF, NERSC, and OLCF.

https://en.wikipedia.org/wiki/License_compatibility). Non-critical optional dependencies can use any OSI-approved license.
RATIONALE: The choice of a permissive open-source license is friendlier to use by commercial entities. Note that strong copyleft licenses (e.g., GPL) are not considered compatible with permissive licenses. Weaker copyleft licenses (e.g., LGPL or GPL v2 with runtime exception) can be considered compatible for xSDK distribution purposes. Licenses that restrict commercial use are not acceptable in the xSDK. The xSDK leadership reserves the right to define a build where packages with weaker copyleft licenses (e.g., LGPL or GPL v2 with runtime exception) are optional, so that users who wish to avoid these restrictions can do so.

M8. Each package must provide **a runtime API**, for example a function call, to return the **current version number** of the software and indicate what configure/CMAKE and compiler options were used to build the package. For development versions of the software, each package must provide the current commit ID in the repository. With this information users should be able to rebuild the package in the same state. We do not currently require that version information for all dependent packages be provided, so it may not be possible to rebuild the entire software stack in the same state.

M9. Each package should use a **limited and well-defined symbol, macro, library, and include file name space**. For example, there should be no publicly visible include files such as utils.h, or package named libutil.a or macros named YES or TRUE. Namespacing of include files can be handled either by prepending each include file with a package name, for example <XXXmat.h>, or by placing and referencing all include files in a subdirectory with a package name, for example <XXX/mat.h>. Note that using a -I/XXX/ and referencing via <mat.h> would not be acceptable namespacing.

M10. It is mandatory that each package have a **public repository,** for example at GitHub, GitLab or Bitbucket, where the development version of the package is available. It is recommended supporting **pull requests,** which enables xSDK library developers and users to provide improvements, bug fixes and interoperability fixes.

M11. No package should have hardwired print or I/O statements that cannot be turned off through a programmatic interface[5]; output should never be hard-wired to stdout or stderr. It is recommended that packages provide a way for users to turn on output and allow them to direct where it goes.[6]Also,

[5] Packages should not exclusively use environmental variables as a programmatic interface since other packages that may be controlling the simulation process cannot set such environmental variables. There must be an API that can be called from with the source code. It is fine to also support using environmental variables, but that cannot be the only way.
[6] For example, allowing users to control output in a C++ package means that the package must accept an arbitrary std::ostream object and all output should go to that object. In C, the package should accept a FILE object to which it can direct its output.

packages may print to stdout by default but only on one process (i.e., root rank "0"). But packages may also be completely silent by default (and require that users turn on outputting in the appropriate way).

M12. If a package imports software that is externally developed and maintained,[7] then it must allow installing, building, and linking against an outside copy of that software. Acceptable ways to accomplish this include (1) forsaking the internal copied version and using an externally provided implementation or (2) changing the file names and namespaces of all global symbols to allow the internal copy and the external copy to coexist in the same downstream libraries and programs.

M13. When configured with a prefix, a package must install its headers and libraries under <install-prefix>/include/ and <install-prefix>/[lib,lib64]/, respectively. In addition, the libraries and header file names should not have the version number embedded in them (except for shared libraries that can have soname versions and symlinks like `lib<package>.so -> lib<package>.so.X -> lib<package>.so.X.Y.Z`).
The aim of this policy is to prevent packages from installing into a non-standard directory layout and for other xSDK packages to be able to detect the location of include files and libraries.

M14. All xSDK-compatible packages must be buildable using 64-bit pointers (this is commonly the default). It is not required that they be buildable with 32-bit pointers.

M15. All xSDK compatibility changes should be sustainable (i.e., they should go into the regular development and release versions of the package and should not be in a private release/branch that is provided only for xSDK releases)

M16. Any xSDK-compatible package that compiles code should have a configuration option to build in Debug mode. Debug mode must produce debugging symbols in compiled code and may contain further useful debugging information and additional error checking. In Spack, the debug build type may be enabled via the variants recommended in the [xSDK Spack variant guidelines]. The aim of this policy is to enable the end-user to produce a version of the code that is suitable for debugging.

M17. Each xSDK member package should have **sufficient documentation** to support use and further development. While it is difficult to formalize what constitutes sufficiency of documentation, the

[7] For example, some projects import source for some routines from BLAS and LAPACK. The Trilinos Teuchos package imported an early version of the boost::any class. Also, Trilinos has its own copy of an older version of SparseSuite. In the latter two cases, new file names and namespaces were created for the imported software to allow coexistence with the (updated) external versions.

requirement will be reviewed by the xSDK community based on documentation elements and **must** include:

- Instructions on installation and getting started,
- Documentation on functionality, for example class- and member-level documentation (in object-oriented languages) or more generally API-level documentation.

It is recommended that the documentation also include the following:

- Discussions of how individual components of the code base interact (for example, in object-oriented languages, how classes or groups of classes interact), ideally using "typical use case" code snippets,
- Presence of complete example codes that illustrate typical use cases,
- Links to external resources such as mailing lists and forums where users and developers can find help,
- Instructions for on-ramping new developers,
- Description of team processes.

xSDK Recommended Policies

In addition to the required xSDK policies listed above, the following capabilities are also recommended.

R1. Each package should have **at least one validation (smoke) test** that can be invoked through the Spack package. This will be a post-installation test that validates the usability of the package and must be executable via 'spack test run <package_name>'.
Policy compliance requires that the 'spack test run' interface works on a standalone system. For MPI tests, MPICH or OpenMPI may be used.
Some xSDK packages with existing validation tests include PETSc, Tasmanian, and libEnsemble.
This policy enables successful installation to be quickly tested before proceeding with more resource-intensive tests. Further information can be found in the Spack packaging guide.

R2. It is recommended that all packages make it possible to run their test suite under Valgrind in order to test for memory corruption issues.

R3. It is recommended that each package **adopt and document a consistent system for propagating/returning error conditions/exceptions** and provide an API for changing the behavior. For example, all routines may, by default, return an error code with the option of changing it to

generate an abort on the error (for running in the debugger). No package should have hardwired calls to abort, exit, or MPI_Abort(). Also, no package should have hardwired print statements for error conditions. Each package should document which error codes/exceptions are recoverable, which may result in lost resources (for example, unfreed memory), and which indicate that the process may be in an undefined or totally broken state (for example, after a segmentation violation). It is the responsibility of the calling routine not to simply continue the computation when a “hard” error is returned; the calling routine will likely, by default, call an abort, but again that should be possible to override.

R4. It is recommended that each package **free all system resources it has acquired** as soon as they are no longer needed. This recommendation includes closing open files, freeing memory, freeing MPI communicators, and freeing MPI data types created by the package. In particular, it is crucial that xSDK compatible code not have any growing memory leaks (such as leaking memory during every timestep). Any resources created by the package that should be freed by the user, rather than by the package, must be fully documented. Valgrind can be used to locate when these resources are mistakenly not released.

R5. It is recommended that each package provide a **mechanism to export its ordered list of library dependencies** so that any other package or executable linking to the package knows to include these dependencies when linking.
RATIONALE: When using static libraries, some compilers require the libraries in the link command to be listed in the correct dependency order - to avoid unresolved symbol errors.
One way this information can be provided to the users is via *pkg-config* file.

R6: Each package should document the **versions of packages with which it can work or upon which it depends,** including software external to the xSDK, preferably in a machine-readable form. The developers of xSDK member packages will coordinate the needed versions of various packages for each xSDK release.

R7. It is recommended that each package should have a README or README.md, a SUPPORT or SUPPORT.md, a LICENSE or LICENSE.md, and a CHANGELOG or CHANGELOG.md file in its top directory. The README file should contain at least the following information:

- a **brief** description of the package,
- information on how to install the package or a link to a file (e.g., INSTALL, INSTALL.md or similar) or a website with the installation information,
- a link to the package webpage (if there is one)

The LICENSE file should contain the full text of the license. If the full text can be found in a different directory or via a link to a website it is sufficient to provide a link to the full text in the file.
The SUPPORT file should contain the contact information that is required by community policy M5 to be able to get help.
The CHANGELOG file should contain important changes or a link to a file or webpage with this information, but not each individual commit message.
If providing additional information, e.g. on how to contribute, authorship, code of conduct, etc, it is recommended to consider suggestions in https://github.com/kmindi/special-files-in-repositoryroot/blob/master/README.md .

R8. Each package should provide pre-processor macros that allow for version comparison (for languages that support it - like C/C++/Fortran). Using these macros, dependent software can potentially be compatible with multiple versions of the package.

Example implementations for a given `package version 1.2.3`

- Example 1: provide version macros only, via public include files (or can be done easily via CMake)

```
#define <PACKAGE>_VERSION_MAJOR 1
#define <PACKAGE>_VERSION_MINOR 2
#define <PACKAGE>_VERSION_PATCH 3
```

- Example 2: provide a single integer version macro to encode major, minor, patch version

```
#define <PACKAGE>_VERSION 1002003
```

Additional comparison macros can help usage

- With Example 1 (via public include files)

```
#define <PACKAGE>_VERSION_GE(Major, Minor, Patch) \
(((Major == <PACKAGE>_VERSION_MAJOR) && (Minor == <PACKAGE>_VERSION_MINOR) && (Patch <=
<PACKAGE>_VERSION_PATCH)) || \
    ((Major == <PACKAGE>_VERSION_MAJOR) && (Minor < <PACKAGE>_VERSION_MINOR)) ||
(Major < <PACKAGE>_VERSION_MAJOR))

#define <PACKAGE>_VERSION_GT(Major, Minor, Patch) \
```

```
(((Major == <PACKAGE>_VERSION_MAJOR) && (Minor == <PACKAGE>_VERSION_MINOR) && (Patch < <PACKAGE>_VERSION_PATCH)) || \
    ((Major == <PACKAGE>_VERSION_MAJOR) && (Minor < <PACKAGE>_VERSION_MINOR)) || (Major < <PACKAGE>_VERSION_MAJOR))

#define <PACKAGE>_VERSION_EQ(Major, Minor, Patch) \
((Major == <PACKAGE>_VERSION_MAJOR) && (Minor == <PACKAGE>_VERSION_MINOR) && (Patch == <PACKAGE>_VERSION_PATCH))

#define <PACKAGE>_VERSION_LE(Major, Minor, Patch) \
!<PACKAGE>_VERSION_GT(Major, Minor, Patch)

#define <PACKAGE>_VERSION_LT(Major, Minor, Patch) \
!<PACKAGE>_VERSION_GE(Major, Minor, Patch)
```

- With Example 2 (via public include files)

```
#define <PACKAGE>_VERSION_GE(Major, Minor, Patch) \
(Major * 10000 + Minor * 100 + Patch <= <PACKAGE>_VERSION)

#define <PACKAGE>_VERSION_GT(Major, Minor, Patch) \
(Major * 10000 + Minor * 100 + Patch < <PACKAGE>_VERSION)

#define <PACKAGE>_VERSION_LE(Major, Minor, Patch) \
!<PACKAGE>_VERSION_GT(Major, Minor, Patch)

#define <PACKAGE>_VERSION_LT(Major, Minor, Patch) \
!<PACKAGE>_VERSION_GE(Major, Minor, Patch)
```

Implementations that are *NOT* acceptable include:

- providing a string `#define <PACKAGE>_VERSION "1.2.3"` (as string version comparisons are not easy)
- expecting the user to "know" the version or expecting the user to manually edit source-code
- requiring the use of external tools such as CMake or git (on the user application or library side) to generate and access package version macros
- not having "namespaced" macros (as they could conflict with macros defined in other packages, using the package name as a prefix can avoid this issue)

History of xSDK community policies. The original version of this document was prepared in 2015 by Barry Smith with key input from Roscoe Bartlett and feedback from members of the IDEAS project. Over time, revisions have been introduced based on discussions with the broader computational science community and developers of an expanding collection of xSDK member packages. We thank all xSDK package developers, the IDEAS team, and the scientific computing community for insightful discussion about issues and approaches.

- Changes in version 1.0.0, February 17, 2023:
  - Made R1 (must have a public repository) mandatory, merging it into M10.
  - Converted R8 (documentation) to a new mandatory policy (M17).
  - Added new recommended policy requiring a validation test, replacing R1.
  - Added new recommended policy (provide version comparison preprocessor macros), replacing R8.
  - Added wording to M13, clarifying the requirement and providing motivation for the policy.
  - Added clarification to R5, giving rationale for the policy and an example of how to meet the requirement.
  - Minor wording/grammar changes were made to M7 and R2, without changing meaning.
- Changes in version 0.6.0, October 12, 2020:
  - Added new policy R8 on documentation quality
  - Merged policies M1 and M16 with emphasis on use of Spack as xSDK installer
  - Eliminated installation policies which were included in previous M1, provided a document with xSDK Spack installation guidelines
  - Added new policy M16, which requires an xSDK package to have a configuration option to be built in debug mode, a requirement previously included in the eliminated installation policies
- Changes in version 0.5.0, June 27, 2019:
  - Added new policy R7, which recommends the inclusion of various information files in the top directory
  - Dropped the requirement to detect MPI 2 features in M3
  - Made various editorial changes in R2, M5, M13 and M15 for clarification or to fix typo
- Changes in version 0.4.0, July 27, 2018:
  - Split policy M4 into 2 parts: M4 (portability to common platforms) and new policy R6 (package should document the versions of packages with which it can work on on which it depends). See https://github.com/xsdk-project/xsdk-issues/issues/55

  - Revision to M7: language about open source licensing requirements. See https://github.com/xsdk-project/xsdk-issues/issues/56
  - New section on history of policies and summary of changes, misc minor edits
- Changes in version 0.3.0, November 6, 2017: added 2 new policies (M15 and M16), changed naming convention to follow xSDK release number, minor typo edits
- Changes in version 0.3, December 2, 2016: clear definition of xSDK member packages, misc minor edits
- Changes in version 0.2, January 28, 2016: minor edits
- Version 0.1, November 10, 2015: original version

Frequently Asked Questions about the xSDK: See the xSDK FAQ list.

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research and Biological and Environmental Research programs.