xSDK Community Installation Policies:

GNU Autoconf and CMake Options

Version 0.5.0, June 27, 2019

https://xsdk.info/policies

Background: What is software configuration? and How to configure software.

Motivation: Combinations of multiple software packages developed by different groups have become essential for large-scale computational science, where the capabilities needed for modeling, simulation, and analysis are broader than any single team has resources to address. The often-tedious trial-and-error process of obtaining, configuring, and installing any single tool may arguably be manageable. From the perspective of an end-user application scientist, however, handling each tool's installation idiosyncrasies can easily become overwhelming when dealing with several packages in combination. Worse, such problems are compounded by the need for consistency among packages to be used within the same application in terms of compiler, compiler version, exotic compiler optimization options, and common third-party packages such as BLAS and HDF5.

Goal: A key aspect of work in the IDEAS software productivity project is developing an Extreme-scale Scientific Software Development Kit (xSDK). As an initial step in this work, our goal is to define and implement a standard subset[1] of configure and CMake[2] options for xSDK and other HPC packages in order to make the configuration and installation process as efficient as possible on standard Linux distributions and Mac OS, as well as on target machines at DOE computing facilities (ALCF, NERSC, OLCF). Note that we are not requiring that all packages use the same installation software, merely that they follow the same standard procedure with the same option names for installation. This approach provides maximum flexibility for each package to select the most suitable toolchain to use for its package.

Impact: Development of a standard xSDK package installation interface is a foundational step toward the seamless combined use of multiple xSDK libraries. The impact of this work is that all xSDK packages will have standard configuration and build instructions, as well as a tester to ensure that all functionality works properly. In addition, because new packages in the xSDK will follow the same standard, it is possible to make the installations "scriptable," that is, to write tools to install many packages automatically. This work is part of the xSDK Community Package Policies.

---

[1] Packages are free to support their own additional options, but using the standard options should be all that is needed to get correct builds.

[2] A subset of these standard behaviors is implemented in the XSDKDefaults.cmake module and is demonstrated and tested in the CMake project https://github.com/bartlettroscoe/XSDKCMakeProj .

xSDK Standard Configure and CMake Options[3]

1. Implement the default behavior described below. Each package can decide whether XSDK mode is the default mode.[4]
    a. --dis/enable-xsdk-defaults
    b. USE_XSDK_DEFAULTS=[YES,NO]
2. Identify location to install package. Multiple "versions" of packages, such as debug and release, can be installed by using different prefix directories.
    a. --prefix=directory
    b. CMAKE_INSTALL_PREFIX=directory
3. Select compilers[5] and compiler flags.
    a. If the compilers (and or flags) are explicitly set on input, use those:
        i. CC=<cc>, CXX=<cxx>, FC=<fc>, CPP, FFLAGS, FCFLAGS, CFLAGS, CXXFLAGS, CPPFLAGS, LDFLAGS
        ii. CMAKE_C_COMPILER=<cc>, CMAKE_CXX_COMPILER=<cxx>, CMAKE_Fortran_COMPILER=<fc>, CMAKE_C_FLAGS="<flag1> <flag2> ...", CMAKE_CXX_FLAGS="...", CMAKE_Fortran_FLAGS="..."
    b. If the compilers and/or flags are not explicitly set on input but are set in the environment variables FC, CC, CXX, CPP[6], FFLAGS, FCFLAGS, CFLAGS, CXXFLAGS, CPPFLAGS[7], LDFLAGS, then the compilers and flags must be set to these. If both FFLAGS and FCFLAGS are set, then they need to be the same or it is an error.
    c. If the compilers and/or compiler flags are not explicitly passed in or defined in the environment variables listed above, then the package is free to try to find compilers on the system and set the compiler flags consistent with the other settings defined below (e.g., shared libraries vs. static libraries, debug vs. non-debug).[8]

[3] This standard is related only to arguments to CMake and GNU Autoconf; there is no requirement regarding the make system used (for example, that it be GNU make) nor that the make system accepts any particular arguments, such as make LIBS+=-lz.
[4] For packages like Trilinos that need to maintain backward compatibility over consecutive releases, USE_XSDK_DEFAULTS may be FALSE by default.
[5] Packages must support using the MPI compiler wrappers for these arguments.
[6] The environment variable CPP is not supported by raw CMake.
[7] The environmental variable CPPFLAGS is not supported by raw CMake.
[8] All CMake projects should use the same built-in CMake algorithm to find the default compilers, so even when no explicit compilers or flags are set they should use the same compilers and flags. Also, raw CMake projects will append compiler flags based on the build type. See "Selecting compiler and linker options".

4. Create libraries with debugging information and possible additional error checking (default is debug in XSDK mode).
    a. --dis/enable-debug
    b. CMAKE_BUILD_TYPE=[Debug,Release]
5. Select option used for indicating whether to build shared libraries (default is shared in XSDK mode).
    a. --dis/enable-shared (configure)
    b. BUILD_SHARED_LIBS=[YES,NO]
6. Build interface for a particular additional language.
    a. --dis/enable-<language>
    b. XSDK_ENABLE_<language>=[YES,NO]
7. Determine precision for packages that build only for one precision (default is double). Packages that handle all precisions automatically are free to ignore this option.
    a. --with-precision=[single,double,quad]
    b. XSDK_PRECISION=[SINGLE,DOUBLE,QUAD]
8. Determine index size for packages that build only for one index size (default is 32). Packages that handle all index sizes automatically are free to ignore this option.
    a. --with-index-size=[32,64]
    b. XSDK_INDEX_SIZE=[32,64]
9. Set location of BLAS and LAPACK libraries (default is to locate one on the system automatically).
    a. --with-blas-lib=”linkable list of libraries” --with-lapack-lib=”linkable list of libraries” It is fine to use -L and -l flags in the lists
        i. TPL_BLAS_LIBRARIES=”linkable list of libraries”, TPL_LAPACK_LIBRARIES=”linkable list of libraries” (Should not use -L or -l flags in the lists)
10. Determine other package libraries and include directories.
    a. --with-<package> --with-<package>-lib=”linkable list of libraries” --with-package-include=”-I list of include directories”
    b. TPL_ENABLE_<package>=[YES,NO]

    Packages must provide a way for a user to specify a dependent package to use. Packages are free to locate a package on the file system if none is specifically provided by the user. If the user does provide one, however, it must be used; if it is not able to be used, then an error must be generated. A package cannot silently substitute a different installation.

11. In the XSDK mode, XSDK projects should not rely on users providing any library path information in environmental variables such as LD_LIBRARY_PATH.
12. After packages are configured, they can be compiled, installed and “smoke” tested with the following commands: make ; [sudo] make install ; make test_install.
13. After an install the package should provide a machine-readable output to show provenance, that is, what compilers were used and what libraries were linked with, as well as other build configuration information, so that users with problems can send the information directly to developers.

Discussion and Examples

For configure we are trying to match as closely as possible the GNU autoconf and CMake standards and conventions.

1. --prefix=/usr/local/; cmake -DCMAKE_INSTALL_PREFIX=/usr/local
2. CC=/usr/local/bin/mpicc ./configure
    - The reason to support environmental variables is that Linux package managers use environmental variables to set the compiler options, not command line arguments.
    - When reading environmental variables, the configure output should clearly show which variables are being used.
3. ./configure CC=/usr/bin/mpicc; cmake -DCMAKE_C_COMPILER=/usr/bin/mpicc -DCMAKE_CXX_FLAGS=”-O3 -Wall”
    - With CMake projects, compiler flags are passed to the compiler as follows:

      ${CMAKE_<LANG>_COMPILER> ${CMAKE_<LANG>_FLAGS} ${CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE>}

      Therefore, CMAKE_<LANG>_FLAGS never overrides the build type (e.g., DEBUG, RELEASE) specific compiler flags.
4. ./configure --disable-debug; cmake -DCMAKE_BUILD_TYPE=RELEASE

- Debug is the default because it helps users while developing (writing) their code, which is most of the time.
- The optimized/release version may or may not contain debug symbols. Although the consensus is that including debug symbols is a good idea for deeply templated C++ libraries, the object size can become very large. Therefore, we do not require such symbols.

5. ./configure --disable-shared; cmake -DBUILD_SHARED_LIBS=FALSE
   - Shared is the default because linking against the libraries is dramatically faster.
6. ./configure --disable-cxx --enable-fc ; cmake -DXSDK_ENABLE_CXX=FALSE
   - The default is to have C and C++ enabled and Fortran disabled. Packages that do not use Fortran (or C++) are free to ignore that flag.
7. ./configure --with-precision=single
   - If a package automatically supports multiple versions, it can ignore this option.
8. ./configure --with-index-size=64
   - If a package automatically supports multiple versions, it can ignore this option.
9. ./configure --with-lapack-lib="-llapack -lblas"
   - Packages are free to locate a BLAS/LAPACK installation on the file system if none is specifically provided by the user. If the user does provide one, however, it **must** be used; if it is not able to be used, then an error must be generated. A package cannot silently substitute a different installation.
10. ./configure --with-x --with-metis-lib=/usr/local/lib/libmetis.a
    --with-metis-include=-I/usr/local/include
    - In CMake, the analogous approach would be cmake -DTPL_ENABLE_METIS=ON
      -DTPL_Metis_INCLUDE_DIRS=/usr/local/include
      -DTPL_Metis_LIBRARIES=/usr/local/lib/libmetis.a
      However , a package may use CMake's find_package() command to load a dependent library as long as the package provides a way for a user to specify an installation of the dependent library to use, and the package guarantees that the specified installation is not substituted.
    - Packages are free to locate a package on the file system if none is specifically provided by the user. If the user does provide one, however, it **must** be used; if it is not able to be used, then an error must be generated. A package cannot silently substitute a different installation.

- There does not exist any CMake standard allowing an external user to set what external package dependencies should be enabled or disabled when configuring. Therefore, this is a TriBITS/Trilinos standard is which calls external packages “TPL”s and therefore the name “TPL_ENABLE_<package>”.
- MPI is never considered a <package>, and xSDK packages do not need to support --with-mpi-lib and --with-mpi-include. In fact, we recommend against it.

11. In order for linking of applications with a multitude of libraries without users needing to set LD_LIBRARY_PATH, each package likely needs to manage how it handles the rpath linker options when building its libraries.
    - Packages are also free to have configure modes that require setting LD_LIBRARY_PATH.
12. Note that the “make test_install” is run **after** the “make install” and utilizes the **installed** versions of of the library. This type of test is often called a *smoke test*, as it verifies that at least something can be built and run using the installed library. It can consist of one or several distinct tests but should not require parallelism nor take more than a couple of minutes.
13. This information is useful for debugging; it can, for example, be emailed to the package developer when problems arise.
    - The “pkgconfig” format and the “module” are two examples of such representations. Both are unfortunately neither complete nor fully standard.
    - We may want to develop an extension of the pkgconfig standard

Changes:

- Changes in version 0.5.0, June 27, 2019:
  - Changed installation policies 8, 13, and 10 and examples in 10

This document was prepared by Roscoe Bartlett, Jason Sarich, and Barry Smith, with key input from Todd Gamblin. We thank xSDK software developers and the IDEAS team for insightful discussion about issues and approaches.

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.