The Snow Shovel Build System

Corey Reed

October 9, 2013

Contents

1 Introduction

1.1 What is Snow Shovel?

Snow Shovel is an infrastructure for ARIANNA software development that provides a dynamic build system for C, C++ and Fortran. It allows users to work on their analysis code with minimal concern for code compiling and library linking tasks. The build system will track the effects of changes to any particular source file and automatically re-compile and re-link objects as necessary when the user executes a `make` command.

It should be noted that the software contained in Snow Shovel is *not* documented in this article. Rather, this article serves as a reference guide explaining how the build system itself works and may be used.

1.2 Goals of Snow Shovel

- To provide a build system that is powerful, generic and easy to use.
  - Users do not write their own Makefiles.
  - Users do not execute compilation or linking commands (e.g. `g++`) explicitly.
  - Software dependencies automatically and correctly handled.
  - Can be configured to build on different systems (i.e. i386 Linux, x86-64 Linux, MacOSX, etc.)
- To facilitate collaborative software development.
  - Working groups can have their own packages inside Snow Shovel.
  - Generate shared object libraries to reduce copying of code.
- To help enforce coding standards.
  - See ROOT coding standards. [1]
  - Warning-free software.
- To be compatible with external software.
  - Not all packages in Snow Shovel need to be built; those relying on external software not present on a given machine may be skipped.
  - Dependence on ROOT [2] is transparent, but required.

These goals are met by implementing a dynamic build system, similar to that used in the ROOT package.

2 Installation

2.1 Building Snow Shovel for the First Time

1. Checkout the package named snowShovel

    1. Change to a directory in which you want Snow Shovel to be placed. A subdirectory `snowShovel` will be created, and the Snow Shovel contents will be downloaded, by the following command.
    2. `svn co https://arianna.ps.uci.edu/svn/repos/snowShovel/trunk snowShovel`

2. Setup your shell with the following environment variables. See the guide at [3] for an easy solution to changing the location of Snow Shovel or for switching between multiple copies.

    – ROOTSYS should point to the directory of the ROOT [2] package.

    – SNS should point to the full path of the newly created `snowShovel` directory.

    – LD_LIBRARY_PATH on Linux, DYLD_LIBRARY_PATH on MacOSX, should include $SNS/lib

    – PYTHONPATH should also include $SNS/lib

    – (optional) For code requiring DB access, the PQXXINC and PQXXLIB variables should be set as appropriate (on Ubuntu with the `libpqxx` package installed, PQXXLIB should point to /usr/lib and PQXXINC to /usr/include).

3. Run the ./configure script from the $SNS directory.

4. Compile the package using 'make'.

    – It is safe to compile Snow Shovel in parallel using the `-j` option of make. For example, `make -j3` is a good way to compile on a dual core processor.

3 Usage Reference

3.1 Compiling All Packages of Snow Shovel

Run the `make` command from the $SNS directory. This is the most common usage. It is safe to compile Snow Shovel in parallel using the `-j` option of make. For example, `make -j3` is a good way to compile on a dual core processor.

3.2 Compiling One Specific Package of Snow Shovel

Use the make target *package*, or *package*-libs, from the $SNS directory. For example, to build a package called "online", use the command `make online`.

3.3 Cleaning a Build (All Packages or One Package)

To clean all of SNS, use `make clean` from the $SNS directory.

To clean one specific package, use `make package-clean` from the $SNS directory. For example, to clean a package called "online", use the command `make online-clean`.

3.4 Adding a Class to a Package

To add a class to an existing package (directory) in Snow Shovel, several steps are involved. After creating the source code files, it is necessary to tell Snow Shovel to include the new code when building the ROOT dictionary (so it is available from CINT, the ROOT interpreter). It may also be necessary to update the dependencies of the package; for this, see Section 4.4.

Adding a class to a package is best illustrated with an example. To add a class related to FPN in the sigproc package:

1. In the package directory, write the interface to the class in a file with the following naming convention: *ClassName*.h, where *ClassName* begins with 'TSn'. In this example, a suitable name would be `TSnFPN.h`.

2. In the package directory, write the implementation of the class in a file with the following name: *ClassName*.cxx . In this example, a suitable name would be `TSnFPN.cxx`.

3. Edit the '*package*.mk' file in the package directory to tell Snow Shovel that this class should be added to the ROOT dictionary. In this example,

one would add `TSnFPN.h` to the "sigprocDH" list in the "sigproc.mk" file.

4. Edit the '*package*LinkDef.h' file in the package directory to tell rootcint how to add this class to the ROOT class dictionary. In this example, one would add the following line to the "sigprocLinkDef.h" file: `#pragma link C++ class TSnFPN+;` (See the ROOT User's Guide [4] for an explanation of this line.)

5. Use the 'make' command from the $SNS directory to build the new software.

6. Commit the interface, implementation, '*package*.mk' and '*package*LinkDef.h' files to SVN [5]. In this example, in the $SNS/sigproc directory, one would execute the command `svn commit -m ``a new FPN class'' sigproc.mk sigprocLinkDef.h TSnFPN.h TSnFPN.cxx`.

3.5 Adding a New Package to Snow Shovel

1. Use the 'build/add_package *package*' script to add a new package to Snow Shovel. This script does the following:
   - Creates a new directory in Snow Shovel.
   - Adds the *package*.mk and *package*LinkDef.h files to that directory.

2. In the new package directory, edit the *package*.mk and *package*LinkDef.h files as appropriate (i.e. replace "TMyClass" with your actual class(es), etc.). It will likely necessary to state the dependencies of the package; for this, see Section 4.4. In general, the package names that the new package depends on will be added to the *package*LIBDEP variable. Existing packages, such as "online", may be used as examples.

3. Follow the directions of Section 3.4 to put classes in this new package.

4. It may also be necessary to update the dependencies of the package; for this, see Section 4.4. In general, it will be necessary to add the package names that the new package depends on to the *package*LIBDEP variable.

5. Add the directory (package) name to the $SNS/Modules file. Be sure to add the package in the correct place. For example, if the new package will use Root, but not the database, it should be inside the `ifneq ($(ROOTSYS_CONF),)` block, but outside of the `ifneq ($(PQXX_CONF),)` block.

6. Commit the following files to SVN [5]:
   - $SNS/Modules
   - $SNS/*package*
   - $SNS/*package*/*package*.mk
   - $SNS/*package*/*package*LinkDef.h
   - Any interface and implementation files of classes you want to add to this package.
7. Set the ignore properties on the new directory. From the $SNS directory:
   1. `svn propget svn:ignore` *`package`* `> tempfile`
   2. `svn propset svn:ignore` *`package`* `-F tempfile`
   3. `rm tempfile`
   4. `svn commit -m ''set ignore properties''` *`package`*

3.6 Adding a New Executable to Snow Shovel

Implementation files in the special package **'prog'** will be compiled into executable binaries, rather than into a shared object library. The executables will be located in the $SNS/bin directory. This feature is intended to be used to generate steering programs only.

1. Write the steering program in an implementation file, *filename*.cxx . *Note:* This program should be as simple as possible, essentially doing nothing except instantiating and running classes from the various packages of Snow Shovel. All algorithms and significant procedures should be placed in separate classes within an appropriate package of Snow Shovel.
2. Use the '$SNS/add-prog.sh *filename*.cxx' script to generate the steering makefile $SNS/prog/*filename*.mk .
3. Edit $SNS/prog/*filename*.mk to tell Snow Shovel how to build the program.
   - In the *filename*OBJS target, list any extra object files (i.e. other than *filename*.o) that should be built directly into the executable.
   - In the *filename*LINK target, list any libraries that the executable should be linked against. By default, programs are linked against ROOT and Snow Shovel's libraries.

4. Run the 'make' command from the $SNS directory.

5. Commit both the $SNS/prog/*filename*.cxx and $SNS/prog/*filename*.mk files to SVN.

3.7 Compatibility With External Software

Some extra compiler flags may be necessary in order to use external classes or functions. To tell Snow Shovel that extra compiler flags are needed when building a package, the *package*CXXFLAGSEXTRA and *package*LIBEXTRA targets can be used in the *package*.mk file. For example, in the "pqdb" package, which depends on the external PostgreSQL interface library pqxx:

```
pqdbLIBEXTRA := \
        `root-config --glibs` \
        -L$(PQXXLIB) -lpqxx \
        -L$(SNS)/lib -ldb -ldbdat \

pqdbCXXFLAGSEXTRA := \
        -I$(PQXXINC) \

pqdbLIBDEP := \
        db dbdat \
```

A line by line explanation of this block follows.

1. `pqdbLIBEXTRA := \`
   This tells Snow Shovel to use the following directories and libraries against which to link the shared object library of this package.

2. `` `root-config --glibs` \ ``
   This executes the `root-config --glibs` command, which returns "-L$ROOTSYS" followed by a list of the common ROOT libraries.

3. `-L$(PQXXLIB) -lpqxx \`
   This tells Snow Shovel to look for the libpqxx.so (on Linux) library in the $PQXXLIB directory.

4. `-L$(SNS)/lib -ldb -ldbdat \`
   This tells Snow Shovel to link the pqxx package against the db and dbdat packages of Snow Shovel.

5. `pqdbCXXFLAGSEXTRA := \`
   This tells Snow Shovel to use the following compiler flags when building C++ files in the pqdb package.

6. `-I$(PQXXINC) \`
   This tells Snow Shovel to look for include files in the directory specified by the environment variable PQXXINC. This

7. `pqdbLIBDEP := \`
   This tells Snow Shovel that the pqxx package depends on the following other packages in Snow Shovel, so that it should be built only after the others. Specifying these dependencies is vital for parallel compilation to function properly.

8. `db dbdat \`
   This tells Snow Shovel that pqxx depends on the db and dbdat packages.

4 The Build System

4.1 Mechanics

In general, the build system looks in all packages listed in the $SNS/Modules file and compiles any .cxx, .c or .f files it finds there. For each package, it then builds a shared object library, a PROOF archive (par) file and generates the rootmap for the package. The rootmap can be used to determine which shared object library stores a particular class.

For the special package 'prog', it will build binary programs for all .cxx files in that package.

It also keeps track of all dependencies for every .h file in the various packages. This allows proper re-building of only those classes which require re-building and dramatically reduces the need for calls to 'make clean'.

Shared object libraries are located in $SNS/lib, binary executables are located in $SNS/bin, par files are located in $SNS/par and a link to (or copy of) each interface file is located in $SNS/include.

4.2 Configuration

The 'configure' script sets up the use of external packages (i.e. their locations) and specifies the system architecture. This configure file is intended to be editted directly in order to change options; it does not parse options from the command line.

The system architecture setup files are located in the $SNS/build/config directory, and they define the general compilation setup, such as the compiler program and universal compiler flags. Currently, configurations only exist for i386 and x86-64 Linux and MacOSX. More configurations can be written if there is a demand for more system architecture support. However, the dependency of software inside Snow Shovel on external packages may hinder the usefulness of Snow Shovel on other architectures.

The build system stores the values of the relevant environment variables ($SNS, $ROOTSYS, etc.) when `$SNS/configure` is run. The values of these environment variables are then checked whenever 'make' is run, and if they have changed, the user is instructed to re-run the configuration script.

Running the configuration script will also execute a 'make clean'. This is done to ensure that the linking against external packages is properly handled, in case their locations have changed.

The configuration script generates the $SNS/config.mk file which is used to store the values of configuration variables at the time when configure was run.

4.3 The Top-Level Makefile

The $SNS/Makefile essentially only includes $SNS/build/Makefile . Thus, references to "the top-level Makefile" should be understood to be referencing $SNS/build/Makefile .

The Makefile first checks that the configuration file, $SNS/config.mk, has been generated and then includes it. This defines a host of variables that may be available during making, such as $(PQXXINC) .

It then parses the $SNS/Modules file (ignoring comments) to get a list of the packages which should be built, and generates the bootstrap Makefile for each package. This bootstrap file is called $SNS/*package*/Module.mk . It is generated using the template file $SNS/build/template/Module.mk.in . The bootstrap Makefiles will be discussed in further in the next section.

Next, the dependency files (from all packages) are included. These dependency files are simple make rules of the following form:

```
package/TMyClass.d package/TMyClass.o: deps
```

where *deps* is a list of all interface files on which this class depends. Dependency generation will be described in further detail in Section 4.5.

Then, the 'all' target is defined as:

```
        all: libs pars extras map
```

Thus the build system will make shared object libraries, par files, will follow any extra targets that a particular package may have defined and will generate the ROOT library map.

Next, the bootstrap Module.mk files are included. This defines all package-specific make targets.

Finally, some generic make targets, related to clean and map are defined, as well as the general rules for making object files from a .cxx, .c or .f file. ROOT dictionary files (G__*.cxx) have a special make rule since they require a slightly different set of compiler flags.

4.4 Building a Package

The building of a package is controlled by the automatically generated bootstrap Makefiles, $SNS/*package*/Module.mk, which are included into the top-level Makefile.

The Module.mk file first defines the variables needed to build the package. This includes the list of C++ and Fortran implementation files, the list of all interface files in the package, the desired shared object library output name, etc.

It then includes the $SNS/*package*/*package*.mk file. This allows users to control the building of this package in a straight-forward way. The *package*.mk file can define any of the following targets:

- *package*DH - Defines which header files should be used by rootcint to generate the ROOT class dictionary. Normally this is all header files in the package. Note that any class which wants to use the "ClassDef" and "ClassImp" macros *must* should be listed here.

- *package*DHEXTRA - Lists any header files that should be used to generate the ROOT class dictionary, but which are not part of this package.

- *package*CF - Compiler flags passed to rootcint when generating the ROOT class dictionary.

- *package*LIBEXTRA - List of extra libraries to be used when linking the shared object library for this package. This is commonly used to link with external packages such as ROOT.

- *package*LIBDEP - List of any extra dependencies that should be included in the make target when building the shared object library for this package.
- *package*PAREXTRA - List of any extra files that should be included in the PROOF archive file of this package. Custom par setup macros do *not* need to be listed here. Simply making a $SNS/*package*/*package*_setup.C file will result in its being used as the setup macro of the PROOF archive of the package.
- *package*MODULES - List of other packages that should be included in this one. This can be used to generate shared object libraries that simply combine other packages. This may become necessary when classes in two (or more) packages depend on each other.
- *package*CXXFLAGSEXTRA - List of extra compiler flags to use when building .cxx (C++) files.
- *package*CFLAGSEXTRA - List of extra compiler flags to use when building .c (C) files.
- *package*F77FLAGSEXTRA - List of extra compiler flags to use when building .f (F77) files.
- *package*EXTRAS - Used to list any local, user-created make targets. These extra targets will therefore be added to the default build procedure. See $SNS/prog/prog.mk for an (albeit sophisticated) example.
- *package*LOCALCLEAN - Used to list any local, user-created make rules that should be added to the default 'make clean' procedure.
- *package*NOLIB - If defined, then no shared object library will be generated for this package.

Next, the list of all object, dependency and header files for the package are generated and the make rules for linking or copying the header files to $SNS/include are defined.

Finally, the make rules for building the shared object library, the par file, the ROOT dictionary and the dependency files are defined. In the case that extra compiler flags were defined, the generic rules for building object files are made specific for this package (and will use this package's extra flags).

4.5 Dependencies

Dependency make rules are defined in the bootstrap Makefile of a package. As an example, the make rule for generating the dependencies of all C++ files in a package is:

```
package/%.d: package/%.cxx
        $(MAKEDEP) $@ "$(CXXFLAGS) \
        $(packageCXXFLAGSEXTRA)" $< > $@
```

The $(MAKEDEP) variable points to the $SNS/build/depend.sh script. This script uses the $ROOTSYS/bin/rmkdepend program to determine which interface files the class in question depends upon and stores the information in a dependency file. These dependency files are simple make rules of the following form:

```
        package/TMyClass.d package/TMyClass.o: deps
```

where *deps* is a list of all interface files on which this class depends.

4.6 Compilation

The generic make rule for compiling a C++ file is defined at the end of the top-level Makefile as follows.

```
        %.o: %.cxx
                $(CXX) $(OPT) $(CXXFLAGS) -o $@ -c $<
```

The variables used here have been defined during configuration.

Thus, first the dependency file will be generated and this dependency file will be included into the top-level Makefile. This will define an extra make rule for the object (.o) file that lists all of the appropriate interfaces as dependencies. Finally, after all dependencies have been resolved, the object will be compiled.

The procedure for C, F77 and ROOT dictionary files are analogous.

4.7 Library Linking

The make rule for the library linking is in the bootstrap Makefile of each package. It has the following form:

```
$(packageLIB): $(packageLIBDEP) $(packageO) \
               $(packageOMOD) package/Module.mk \
               package/package.mk
        @echo "Making libpackage.$(SOEXT)" \
        @$(MAKELIB) $(PLATFORM) $(LD) "$(LDFLAGS)" \
        "$(SOFLAGS)" libpackage.$(SOEXT) $@ \
        "$(packageO) $(packageOMOD)" \
        "$(packageLIBEXTRA) $(packageLIBEXTRAMOD)"
```

Thus the library depends on

1. Any user defined dependencies (via *package*LIBDEP).
2. All .o files in the package.
3. All .o files in other packages against which this library will be linked (via *package*MODULES).
4. The bootstrap Makefile.
5. The *package*.mk steering Makefile.

The $(MAKELIB) variable points to the $SNS/build/makelib.sh script. This script can control library linking on many different system architectures and can be used to give "major.minor.revision" version numbers to shared object libraries (although this feature is currently not used by Snow Shovel). The $(LD) variable defines which program to use to perform the actual linking. This variable is defined in the system architecture configuration file in the $SNS/build/config directory.

References

[1] ROOT Team. ROOT C++ Coding Conventions, 2013. URL `http://root.cern.ch/drupal/content/c-coding-conventions`.

[2] R. Brun and F. Rademakers. ROOT: An object oriented data analysis framework. *Nucl.Instrum.Meth.*, A389:81–86, 1997. doi: 10.1016/S0168-9002(97)00048-X.

[3] C. Reed. Tips, tricks and shell scripts, 2013. URL `http://arianna.ps.uci.edu/~arianna/media_wiki/index.php/Tips,_tricks_and_shell_scripts`.

[4] ROOT Team. ROOT Reference Guide, 2013. URL `http://root.cern.ch/drupal/content/reference-guide`.

[5] The Apache Software Foundation. Apache subversion software, 2013. URL `http://subversion.apache.org/`.