A Version Management System for GlueX

GlueX-Doc-2793-v14

Mark M. Ito
Jefferson Lab

December 7, 2016

Abstract

A system for building and managing GlueX software is described. The goal is to insulate the user from the need to the master details of building each of several software packages as well as from the details of setting up a working environment. Multiple versions of each of several packages can be maintained simultaneously. Particular combinations of package versions can be specified succinctly in an XML configuration file and this file can be used both to guide a complete build of all needed packages and to set up the shell environment to use the resulting build.

Contents

1 Introduction

There are three fundamental areas of concern that make up all software systems. They are:

1. a directory structure
2. a build system
3. a version management system

They are all related, aspects of one affects aspects of each of the others.

Having a standard system makes collaborative work more efficient, especially when one needs help from others to solve a problem.

2 The Packages

There are several software components that are needed to build and use GlueX software. Most of them are assumed to be provided by the native operating system or distribution, but there are some that have to be built by the GlueX user. They are:

1. build_scripts: scripts to manage building and the shell environment
2. Xerces-C: for reading XML files

3. CERNLIB: to support GEANT 3 simulations
4. GEANT4: simulation engine
5. ROOT: general purpose HENP toolkit
6. EVIO: CODA format data handling library
7. CCDB: Calibration Constants Database
8. RCDB: Run Conditions Database
9. JANA: event-based analysis framework
10. HDDS: detector geometry specification library
11. sim-recon: simulation and reconstruction for GlueX

Detailed description of these packages will not be given here; please see the GlueX Offline Software wiki page for more information.

There are in general multiple versions (releases) of each of these packages and it is often convenient to have access to more that one version of a package built and available for use. In addition some packages depend on one or several others for libraries and include files.

3 The Directory Structure

The VMS directory structure supports multiple versions of each package. For an example see Fig. 1. In the figure, "gluex_top" is a generic name, each installation may choose a different directory name. VMS looks for the name of this directory in the environment variable `GLUEX_TOP`.

Under gluex_top, each package has its own container directory (*e. g.*, `JANA, hdds, sim-recon`) and for each package container directory one or more specific versions of that package are built.

4 Scripts to Implement the VMS: the `BUILD_SCRIPTS` Directory

All scripts and makefiles to support VMS are found the in the `build_scripts` directory. At JLab, the full path is

`/group/halld/Software/build_scripts`

The directory can also be cloned from the Git repository at GitHub. The URL is

`https://github.com/jeffersonlab/build_scripts`

Many of the scripts and makefiles described in this note require that the environment variable `BUILD_SCRIPTS` be defined and point to an instance of this directory.

```
gluex_top
|-- build_scripts
|-- cernlib
|   `-- 2005
|-- evio
|   |-- evio-4.2
|   `-- evio-4.3.1
|-- hdds
|   |-- hdds-3.1
|   `-- hdds-3.2
|-- jana
|   |-- jana_0.7.2
|   `-- jana_0.7.3
|-- root
|   |-- root_5.34.04
|   `-- root_5.34.26
|-- sim-recon
|   |-- sim-recon-1.2.0
|   `-- sim-recon-1.3.0
`-- xerces-c
    |-- xerces-c-3.1.1
    `-- xerces-c-3.1.2
```

Figure 1: The directory structure.

Package	Home Directory Variable
Xerces-C	XERCESCROOT
CERNLIB	CERN
LAPACK/BLAS	see Section 6.2.3
Geant4	GEANT4_HOME
ROOT	ROOTSYS
EVIO	EVIOROOT
CCDB	CCDB_HOME
RCDB	RCDB_HOME
JANA	JANA_HOME
HDDS	HDDS_HOME
sim-recon	HALLD_HOME

Table 1: Packages and their home directories.

5 Setting Up the Shell Environment

Facility is provided for setting environment variables necessary both for building the software and for using it. Both Bourne-shell-like and C-shell-like shells are supported, but real testing has only been done with bash and tcsh. In the following all examples will be appropriate for bash. Note that whenever a script like foo.sh is mentioned, there is also a foo.csh in the build-scripts directory.

5.1 Low-Level Environment Set-Up: `gluex_env.(c)sh`

The `gluex_env.(c)sh` script will define all environment variables needed to run and build GlueX software. It takes as input the home directories of each package as found in the environment variables listed in Table 1.

Given the home directory of a package, there may or may not be other environment variables that need to be set, those variables derived from the value of home. `gluex_env.(c)sh` takes care of this. For example, `XERCES_INCLUDE` is used in the build system and must be defined as `$XERCESCROOT/include`. Directories containing binaries must be added to the `PATH` variable and similarly for `LD_LIBRARY_PATH`, and `PYTHONPATH`. For these path variables directories are always added at the front with any pre-existing directories maintained on the list.

5.1.1 Pre-Defined Home Variables and Defaults

If `GLUEX_TOP` is defined in advance, the pre-defined value will be used. If not it will be defined as `/usr/local/gluex`.

If `BUILD_SCRIPTS` is defined in advance, again the pre-defined value will be used. If not defined it will be defined as `$GLUEX_TOP/build_scripts`.

If any of the home directories are defined before sourcing `gluex_env.(c)sh`, those values will be respected. If any are not defined, then a default will be provided, usually the `prod` directory in the package container directory. Because of this behavior, the user can define as many or as few of the home directories as desired in advance of

```
export GLUEX_TOP=/home/gluex/gluex_top
export BUILD_SCRIPTS=$GLUEX_TOP/build_scripts
export HALLD_HOME=/home/username/sim-recon
source $BUILD_SCRIPTS/gluex_env.sh
```

Figure 2: Example of a custom set-up script. The build of sim-recon in the user's home directory will be used. All other packages be set up to use with their default builds under `/home/gluex/gluex_top`. Note that `GLUEX_TOP` and `BUILD_SCRIPTS` are defined explicitly rather than letting them default.

sourcing `gluex_env.(c)sh`, letting the script finish the environment settings keying off of the user definitions (or lack thereof). The user is thus only responsible for setting the values of desired home directories. A side effect of this behavior is that the environment that results is non-deterministic in the sense that the result depend on the values of pre-existing home directory variables. Different initial conditions will give different environments.

5.1.2 Consistency Checking

The final step in `gluex_env.(c)sh` is to check the resulting environment for consistency using the prerequisites system. Each home directory is checked for a prerequisites version file. Those files list versions of prerequisite packages used at build time. The build-time version are checked against the versions used in the just-set-up environment and warnings are printed when mismatches are detected. See Section 8 for the details.

5.1.3 Cleaning the Environment: `gluex_env_clean.(c)sh`

Since `gluex_env.(c)sh` is sensitive to definitions hanging around in the environment, there is a script provided that will undo all GlueX-related definitions: `gluex_env_clean.(c)sh`. Sourcing it will eliminate unintended consequences from previously made definitions. For the path variables the script only removes the GlueX-related elements leaving all others present in the path.

5.2 High-Level Environment Set-Up

5.2.1 Custom Scripts

The most common reason to have a custom script is when you want to use a package that is outside the standard directory structure. Since `gluex_env.(c)sh` will respect a pre-defined value of any of the home directories, this can be done without making a private version of `gluex_env.(c)sh`. See Fig. 2 for an example. Here a version of sim-recon built in a non-standard location (the user's home directory) will be used in the resulting environment.

5.2.2 Using the Versioning System

/labelsection:env-use-versioning

```
export GLUEX_TOP=/home/gluex/gluex_top
export BUILD_SCRIPTS=$GLUEX_TOP/build_scripts
source $BUILD_SCRIPTS/gluex_env_version.sh \
    /home/username/my_versions.xml
```

Figure 3: Example of a set-up script driven by a private version file. See Section 7 for the format of the file.

Another way to get a non-default environment is to use the versioning system to set home directory locations which in turn are respected by `gluex_env.(c)sh`. The versioning system is described in detail in Section 7, but an example set-up script is shown in Fig. 3. In this example, package version information is contained in the XML file `my_versions.xml` in the user's home area. An example version file is shown in Fig. 4. Alternate combinations of package versions can be tried by making alternate versions of the version file.

5.2.3 Default Environment at JLab

At JLab there is script that packages the steps shown in Fig. 3, with the appropriate values of `GLUEX_TOP` and `BUILD_SCRIPTS`. It uses the latest version of `version.xml`. It can also be used as an example for using a custom version of `version.xml`. To use it, for bash:

```
source /group/halld/Software/build_scripts/gluex_env_jlab.sh
```

and for tcsh:

```
source /group/halld/Software/build_scripts/gluex_env_jlab.csh
```

6 The Build System

Each of the packages have their own native build system and each build system has its own set of details that have to be understood. In addition, the technology used to do the build varies from system to system. It may be make, imake, cmake, SCons, or something else. The VMS system makes a choice of build options for each package so that the user need not master these details.

6.1 The Makefiles

The VMS build system is implemented in GNU Make. These makefiles invoke the native package-specific build system. The top-level makefile builds packages into the standard directory structure described in Section 3. It in turn uses a "package makefile" for each package (*e. g.*, `Makefile_jana, Makefile_sim-recon`). Invoking make with a package makefile will build that package with the home directory placed in the current working directory. In other words, the package makefiles have no knowledge of the directory structure within which they are used; they just build locally. Only the top-level makefile knows about the directory structure.

6.1.1 The Top-Level: Makefile_all

Complete builds are orchestrated by Makefile_all. The highest-level targets of Makefile_all, shown with their dependencies are:

```
all: env xerces_build cernlib_build lapack_build root_build \
    clhep_build geant4_build gsl_build evio_build rcdb_build \
    ccdb_build jana_build hdds_build sim-recon_build

gluex: env xerces_build cernlib_build lapack_build root_build \
    clhep_build evio_build rcdb_build ccdb_build jana_build \
    hdds_build sim-recon_build

gluex_jlab: env xerces_build root_build clhep_build evio_build \
    rcdb_build ccdb_build jana_build hdds_build sim-recon_build
```

The all target builds every package that Makefile_all knows about. The gluex target builds only the packages necessary for using GlueX software. The gluex_jlab target is the same as gluex except that it does not include cernlib_build (useful for JLab public builds where we use the community-built versions of CERNLIB).

Makefile_all should always be invoked from the $GLUEX_TOP directory.

6.1.2 Individual Package Targets

Each of the individual package targets (*e. g.*, evio_build and hdds_build) use the corresponding package makefile. Directories are created and the package makes are executed in way that gives the directory structure described in Section 3. To do this, the package container directory is created if it does not exist and the requested package makefile is invoked from within the package container directory.

Of course, each individual package build target can be invoked directly. More on this in Section 6.3

6.1.3 Special Package Target

The target cernlib_debug is non-standard. If this target is invoked, it well create a separate container directory cernlib_debug for the debug versions. Also it does not have its own package makefile, rather on 64-bit machines, it invokes Makefile_cernlib_Vogt with command line options that cause appropriate debug compiler flags to be used.

To use the resulting debug version of CERNLIB, the CERN variable must be set to point to the cernlib_debug directory, either explicitly as an environment variable (as described in Section 5.2.1 or by using the home attribute in the package element of a version file (as described in Section 7.

6.2 The Package Makefiles

Each of the package makefiles is sensitive to environment variables that control which version of the package to build. The makefiles themselves take care of obtaining the source code. In general, there are two ways to get the code: downloading a tarball or checking the code out from a version control repository, although the later option is not available for all packages.

6.2.1 Specifying the Version of a Tarball

Each packages respects a version-specifying environment variable. Here is an example of how they might be set in the C-shell:

```
setenv JANA_VERSION 0.7.2
setenv SIM_RECON_VERSION 1.2.0
setenv HDDS_VERSION 3.2
setenv CERNLIB_VERSION 2005
setenv XERCES_C_VERSION 3.1.1
setenv CLHEP_VERSION 2.0.4.5
setenv ROOT_VERSION 5.34.26
setenv CCDB_VERSION 1.05
setenv RCDB_VERSION 0.00
setenv EVIO_VERSION 4.3.1
```

In each case there is standard system for distributing tarballs marked with the version name. Each package has different conventions, but the package makefiles have that knowledge of the appropriate convention coded in. Also, the name of home directory created depends on the name that appears in the tarball (with exceptions as mentioned in Section 7.4. Note that the version variable can be set on the make command line as well.

6.2.2 Checking-Out from a Repository

Some packages can be checked out from a Subversion or Git repository. If that is the desired source of the code, the version variable should not be set. Instead a URL variable should be used to specify the location of the repository. For example:

```
setenv HDDS_URL https://github.com/jeffersonlab/hdds
```

will cause the HDDS package makefile to check out the master branch of the HDDS Git repository at GitHub. The names of the variables for other packages are `JANA_URL` (Subversion), `CCDB_URL` (Git), `RCDB_URL` (Git), and `SIM_RECON_URL` (Git) respectively.

For packages that use a Git repository, there are two additional variables that can be used to control the checkout. `SIM_RECON_BRANCH` is used to check-out a specific branch and `SIM_RECON_HASH` is used to check-out a specific commit of sim-recon. If one is set, the other should not be. There are analogous variables for CCDB, RCDB, and HDDS.

6.2.3 Extra Features for Specific Packages

SCons Options for `Makefile_sim-recon`

For the sim-recon package, there is a variable, `SIM_RECON_SCONS_OPTIONS` that can be defined, either in the shell environment on on the make command line, that will supply optional arguments to the `scons` command invoked by make. For example,

```
make -f $BUILD_SCRIPTS/Makefile_sim-recon \
    SIM_RECON_SCONS_OPTIONS='SHOWBUILDS=1'
```

will cause SCons to show the compiler commands explicitly.

Building LAPACK/BLAS

The `LAPACK` and `BLAS` libraries are needed by CERNLIB. They are downloaded and build automatically, but rather than being installed in their own home directory, The "install" target of `Makefile_lapack` adds them to the lib directory of your CERNLIB build.

6.3 Adding New Package Builds to an Existing Tree

There are two ways:

6.3.1 Use the Top-Level `Makefile_all`

After setting the desired values of the version environment variables and/or the URL environment variables (see Section 6.2 you can invoke `Makefile_all` with the target(s) needed or with a high-level target like `gluex`,

```
cd $GLUEX_TOP
make -f $BUILD_SCRIPTS/Makefile_all gluex
```

If some of the versions of individual packages requested already exist, then make will do the usual thing: try to remake them and find that there is nothing to do.

6.3.2 Use Individual Package Makefiles

Since the individual package makefiles build in the local directory, they can be used directly by going to the appropriate container directory. For example,

```
cd $GLUEX_TOP/sim-recon
make -f $BUILD_SCRIPTS/Makefile_sim-recon SIM_RECON_VERSION=1.4.0
```

Note that in this example the version is specified on the make command line rather than through an environment variable. That is not necessary; it is an option supported by make and defining `SIM_RECON_VERSION` in the environment would work as well. Also note that doing the build in the sim-recon container directory is not necessary for the build to succeed; any directory will work. In this example however we are adding to an existing standard directory structure so we `cd` to the standard directory.

7 The Versioning System

7.1 Version File Format: an XML

The versioning system uses an XML-formatted version file to specify both package version information and package home directory definition in the shell environment. An example file is shown in Fig. 4.

There is only one type of element, the `package`. Attributes are:

name: The name of the software package.

version: The version number of the package.

```
<gversions>
<package name="jana" version="0.7.3"/>
<package name="sim-recon" version="1.4.0"/>
<package name="hdds" version="3.3"/>
<package name="cernlib" version="2005" word_length="64-bit"/>
<package name="xerces-c" version="3.1.1"/>
<package name="clhep" version="2.0.4.5"/>
<package name="root" version="5.34.26"/>
<package name="ccdb" version="1.05"/>
<package name="evio" version="4.3.1"/>
<package name="rcdb" version="0.00"/>
</gversions>
```

Figure 4: An example version file. `version_1.7.xml` is shown.

url: A URL to be used to checkout (Subversion) or clone (Git) the code. The URL should point to an appropriate repository.

branch: When using a Git repository, the branch to be checked-out.

hash: When using a Git repository, the hash of the commit to be checked-out.

dirtag: A string (directory tag) to be appended to the standard directory name of the package when it is built.

home: Force the location of the package home directory when setting up the environment.

7.2 Setting Up the Environment with a Version File

As we saw in Section 5.1, environment setting via `gluex_env.(c)sh` is sensitive to the definition of the package home variables. In Section 6.2, we saw that the package makefiles are sensitive to either the version-defining environment variables or the URL-defining environment variables, using them to choose the version of code to build. The version file can be used to define both classes of variables. In this way it can be used to both build a consistent set of packages and to set-up the environment to use the build. Executing

```
$BUILD_SCRIPTS/version.pl version_1.7.xml
```

where for the purposes of this example `version_1.7.xml` is the file shown in Fig. 4 and `GLUEX_TOP` is `/home/gluex/gluex_top`, creates the output shown in Fig. 5. Since you would want these commands applied to the current shell level, in practice you use

```
eval `$BUILD_SCRIPTS/version.pl version_1.7.xml`
```

Following this step, one normally would invoke `gluex_env.(c)sh` to complete the set-up of the environment.

In this example, the variable definitions come (mostly) in pairs, a version variable and a home directory variable. The version variable affects only the build process since

```
setenv JANA_VERSION 0.7.3;
setenv JANA_HOME \
    /home/gluex/gluex_top/jana/jana_0.7.3/Linux_RHEL7-x86_64-gcc4.8.3;
setenv SIM_RECON_VERSION 1.4.0;
setenv HALLD_HOME /home/gluex/gluex_top/sim-recon/sim-recon-1.4.0;
setenv HDDS_VERSION 3.3;
setenv HDDS_HOME /home/gluex/gluex_top/hdds/hdds-3.3;
setenv CERNLIB_VERSION 2005;
setenv CERN /home/gluex/gluex_top/cernlib;
setenv CERN_LEVEL 2005;
setenv CERNLIB_WORD_LENGTH 64-bit;
setenv XERCES_C_VERSION 3.1.1;
setenv XERCESCROOT /home/gluex/gluex_top/xerces-c/xerces-c-3.1.1;
setenv CLHEP_VERSION 2.0.4.5;
setenv CLHEP /home/gluex/gluex_top/clhep/2.0.4.5;
setenv ROOT_VERSION 5.34.26;
setenv ROOTSYS /home/gluex/gluex_top/root/root_5.34.26;
setenv CCDB_VERSION 1.05;
setenv CCDB_HOME /home/gluex/gluex_top/ccdb/ccdb_1.05;
setenv EVIO_VERSION 4.3.1;
setenv EVIOROOT /home/gluex/gluex_top/evio/evio-4.3.1/Linux-x86_64;
setenv RCDB_VERSION 0.00;
setenv RCDB_HOME /home/gluex/gluex_top/rcdb/rcdb_0.00;
```

Figure 5: Output of `$BUILD_SCRIPTS/version.pl`.

the corresponding package makefile keys off it (see Section 6.2). The home directory variable affects the build as well in that it tells the package makefile where to find any prerequisite packages and in addition it affects use of a build via its effect on path variables.

Finally, the script `gluex_env_version.(c)sh` combines use of `version.pl` and `gluex_env.(c)sh` to more conveniently set up the environment. We have already seen an example of its use in Fig. 3. The script uses `version.pl` as shown above to set the stage for an invocation of `gluex_env.(c)sh`.

7.3 Specifying Alternate Source Code Sources with a Version File

We saw in Section 6.2 that a URL variable can be used to instructed the package makefiles to get the source code from a version control repository rather downloading a tarball. The `url` attribute in the package element calls out the value of the URL to use directly. In a particular `package` element, either the `version` attribute or the `url` attribute should appear; if both appear then the `version` attribute will be used (*i. e.*, tarball). If the `url` attribute is used, each package makefile will interpret the URL as is appropriate for that package, either as a Subversion repository or a Git repository; there can be only one answer and it is coded into the package makefile. For Git repositories, the optional `branch` attribute controls which branch is checked out. If it is absent, the master branch is used. For example, to clone sim-recon and checkout the branch `test_stuff`, use

```
<package name="sim-recon"
         url="https://github.com/jeffersonlab/sim-recon"
         branch="test_stuff"/>
```

This will cause the `SIM_RECON_BRANCH` variable to be set in the environment. Similarly the `hash` attribute can be used to specify the hash of the particular commit to be checked-out.

```
<package name="sim-recon"
         url="https://github.com/jeffersonlab/sim-recon"
         hash="22fe917"/>
```

Note that for Subversion repositories, the branch specification is encoded in the URL itself and the branch and hash attributes are ignored.

7.4 Directory Tags

The `dirtag` attribute can be used to distinguish different builds of a package where the only difference between them is the version(s) of one or more prerequisite packages. The string used is arbitrary. A directory tag can be attached to either a source directory made from a tarball or one from a source code repository. The tag name is appended after a caret symbol (^), for example,

```
<package name="hdds" version="3.3" dirtag="xerces_test"\>
```

in a version file would cause `version.pl` to add an additional variable to the environment:

```
setenv HDDS_DIRTAG xerces_test
```

and `Makefile_hdds` would then produce a directory named `hdds-3.3^xerces_test`, with source code obtained from the standard tarball, `hdds-3.3-src.tar.gz` in this case.

The corresponding home directory variable will also reflect the directory tag, of course.

There are a lot possible meanings for the directory tag. It could mark different combinations of prerequisites as well as designating packages where the source code does **not** come from a standard source (tarball or repository). Because of the large number of possibilities, the form of the tag string is left to the user; no assumption is made about it meaning.

7.5 Specifying Alternate Home Directory Locations

Often one wants to use a build of a specific package that lies outside of the standard directory structure. This can be put into the environment by setting the `home` attribute of the corresponding `package` element. For example

```
<package name="sim-recon" home="/home/my/sim-recon"/>
```

will cause `version.pl` to generate

```
export HALLD_HOME=/home/my/sim-recon
```

Note that this feature is mainly useful for creating an environment for use; when building it (a) gives no guidance on where the source code should come from and (b) does not cause the build to be done in the named directory. It is useful when a pre-built package needs to be referenced for the current task.

8 The Prerequisites System

Each package may or may not have a build dependency on other packages under the VMS. For example a particular version of sim-recon can be built against any of a number of versions of HDDS, including custom versions provided by the user. To insure that the environment being set-up has a consistent set of package versions, a facility is provided to warn the user if possible inconsistencies are detected.

At build time, a version xml file is created in the home directory of a package if that package has dependencies on others in the system. For example, $HALLD_HOME will have the file sim-recon_prereqs_version.xml, listing the versions used to build sim-recon. An example is shown in Fig. 6

At set-up time, when `gluex_env.(c)sh` is invoked, if a version file with prerequisites is found in the package home directory, then each package in that file is checked for version consistency. A match is sought between the version number specified in the version file and the version number encoded in the home directory for the prerequisite package, *i. e.*, the directory defined as home in the environment being set-up. Here the version from the home directory is extracted in two ways depending on how the package was built:

```
<gversion version="1.0"
><package name="evio" version="4.3.1"
 /><package name="cernlib" version="2005"
 /><package name="root" version="5.34.26"
 /><package name="jana" version="0.7.3"
 /><package name="hdds" version="3.3"
 /><package name="ccdb" version="1.05"
 /></gversion
>
```

Figure 6: An example of sim-recon_prereqs_version.xml.

1. **Tar File**. If the source code came from a tar file, then the version number is parsed out of the name of the home directory.
2. **Subversion Check-Out**. If the source was checked out of a subversion repository, the `svn info` command is used to get the name of the subversion directory checked out and the version is parsed from that directory name.
3. **Git Clone and Check-Out**. If the source code was checked-out from a Git repository, the `git remote -v` command is issued and the URL is parsed from the "fetch" line. The branch is obtained from the `git status` command. If the prerequisite file does not contain a branch specification, then the version check will require the master branch to have been checked out. At this writing, version checking for specific commit hashes has not been implemented.

If a version mismatch is found, a warning is written to the screen.

There are cases where the home directory does not contain any information about the source of its source code. For example, the code could have come via the `svn export` command or the `git archive` command. If a such a build (for example HDDS) is a prerequisite of another package (for example, sim-recon), then the dependent package (sim-recon) will usually have listed the prerequisite (HDDS) with neither a `version` nor a `url` attribute defined in its (sim-recon's) prerequisite file. In that case, a warning will be issued noting the absence of both `version` and `url` attributes.

9 The gluex_install System

The `gluex_install` system uses `build_scripts` to create a complete install of GlueX software from scratch. This is especially useful for new machines.

No interaction from the user should be required to get a successful build. The only assumption made is that the basic packages that come in a minimal install are present. The definition of minimal depends on the installation. In all cases, the distribution was tested by first installing from a DVD or CD ISO image. Typically, the "live DVD" version was chosen since that installs the smallest number of packages.

The scripts have been tested on the following distributions listed in Table 2.

Distribution	Package Type
CentOS	RedHat
Scientific Linux	RedHat
Ubuntu	Debian
Fedora	RedHat
LinuxMint	Debian
openSUSE	RedHat
RedHat Enterprise	RedHat

Table 2: Gluex_install tested distributions.

9.1 Installation Steps

Root access is required for steps (1) and (3).

1. **System Update**. It is recommended that you update your system to the latest versions of of all system supplied software. For RedHat-like distributions you do a "yum update". For Debian-like systems you do a "apt-get update".

2. **Get the Scripts**. A tar file with the scripts described here is available at

   ```
   https://github.com/JeffersonLab/gluex_install/archive/latest.tar.gz
   ```

   You can also do a git clone of the latest version:

   ```
   git clone https://github.com/jeffersonlab/gluex_install
   ```

3. **Prerequisites: gluex_prereq_<distribution>.sh**. The prerequisites script installs packages from the distribution repository necessary for the GlueX build. As such, it must be executed by root. In addition it makes some symbolic links in system directories that are necessary for the cernlib build. These scripts are specific to particular distributions. You must run this script from inside the "gluex_install" directory created when you get the scripts (see Get the Scripts, step 2).

4. **Install: gluex_install.sh**. Creates a directory, "gluex_top", in the current working directory to house the build, sets up an environment, downloads all source files, and builds all libraries and executables needed to run GlueX software. The install assumes a directory structure that accommodates multiple versions of the GlueX packages if they are needed later. The script is distribution independent.

9.2 Using the Build

After the build is complete, there are two files in the gluex directory, setup.sh and setup.csh, that can be used to set-up the complete GlueX environment under Bourne-like shells or C-like shells respectively.