

BUILD SCRIPTS: A Version Management System for GlueX

GlueX-Doc-2793-v25

Mark M. Ito
Jefferson Lab

October 12, 2021

Abstract

BUILD SCRIPTS, 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	2
2	Getting Started	2
2.1	Getting a Copy of Build Scripts	3
2.1.1	Pre-Installed at JLab	3
2.1.2	Pre-Installed Outside JLab	3
2.1.3	Install via zip or tar file	3
2.1.4	Clone from GitHub	3
3	The Packages	3
4	The Directory Structure	4
5	Scripts to Implement Build Scripts	4
6	Setting Up the Shell Environment	6
6.1	Low-Level Environment Set-Up: <code>gluex_env.(c)sh</code>	6
6.1.1	Pre-Defined Home Variables and Defaults	6
6.1.2	Consistency Checking	6
6.1.3	Cleaning the Environment: <code>gluex_env_clean.(c)sh</code>	8
6.2	High-Level Environment Set-Up	8
6.2.1	Custom Scripts	8
6.2.2	Using the Versioning System	8

6.2.3	Default Environment at JLab	9
6.2.4	Simple Environment Set Up	9
7	The Build System	10
7.1	The Makefiles	10
7.1.1	The Top-Level: <code>Makefile_all</code>	10
7.1.2	Individual Package Targets	11
7.1.3	Special Package Target	11
7.2	The Package Makefiles	12
7.2.1	Specifying the Version of a Tarball	12
7.2.2	Checking-Out from a Repository	12
7.2.3	Extra Features for Specific Packages	12
7.3	Adding New Package Builds to an Existing Tree	13
7.3.1	Use the Top-Level <code>Makefile_all</code>	13
7.3.2	Use Individual Package Makefiles	13
7.4	Building Custom Versions of Packages	14
7.4.1	<code>my_halld_build_jlab</code>	14
7.4.2	<code>my_halld_update.py</code>	15
8	The Versioning System	15
8.1	Version Set File Format: an XML	15
8.2	Setting Up the Environment with a Version Set File	16
8.3	Specifying Alternate Source Code Sources with a Version Set File	16
8.4	Directory Tags	18
8.5	Specifying Alternate Home Directory Locations	18
8.6	Public Version Set Files	19
9	The Prerequisites System	19
10	The <code>gluex_install</code> System	20
10.1	Installation Steps	21
10.2	Using the Build	21

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 Getting Started

This section give concise instructions for performing a few common tasks with BUILD SCRIPTS.

2.1 Getting a Copy of Build Scripts

There are several sources.

2.1.1 Pre-Installed at JLab

At JLab, BUILD SCRIPTS is installed in `/group/halld/Software/build_scripts`. You can start by defining an environment variable:

```
export BUILD_SCRIPTS=/group/halld/Software/build_scripts
```

for bash or

```
setenv BUILD_SCRIPTS /group/halld/Software/build_scripts
```

for tcsh.

2.1.2 Pre-Installed Outside JLab

Outside JLab, when the `gluex_install` procedure has been followed (see section 10), BUILD SCRIPTS will be installed under `$GLUEX_TOP/build_scripts`. You will have to contact your local expert to find the location of `$GLUEX_TOP`. Once that is known the BUILD_SCRIPTS environment variable to point to `$GLUEX_TOP/build_scripts`.

2.1.3 Install via zip or tar file

All of the releases of BUILD SCRIPTS are available from GitHub at

```
https://github.com/JeffersonLab/build\_scripts/releases
```

The latest release will be listed first. You will see links for downloading the files in zip and tar format. After unpacking the files, set your BUILD_SCRIPTS environment variable to the full path of the resulting top-level directory.

2.1.4 Clone from GitHub

You can clone the Git repository:

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

3 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. ROOT: general purpose HENP toolkit
2. JANA: event-based analysis framework
3. sim-recon: simulation and reconstruction for GlueX (deprecated)
4. HDDS: detector geometry specification library

5. CERNLIB: to support FORTRAN-based HENP code programs
6. Xerces-C: for reading XML files
7. CCDB: Calibration Constants Database
8. Geant4: simulation of particle interactions with matter
9. EVIO: CODA format data handling library
10. RCDB: Run Conditions Database
11. HDGeant4: Geant4 implementation of the GlueX detector simulation
12. hd_utilities: miscellaneous scripts and programs
13. gluex_root_analysis: ROOT-based analysis of post-reconstruction data
14. AmpTools: framework for amplitude-based physics analysis
15. SQLiteC++: a C++ SQLite wrapper library
16. SQLite: server-less SQL database engine
17. gluex_MCwrapper: system for submitting simulation jobs
18. halld_sim: simulation programs and utilities for GlueX
19. halld_recon: reconstruction for GlueX
20. LAPACK: old-school linear algebra package
21. HepMC: utility library for particle physics
22. PHOTOS: package for simulating radiative effects
23. EvtGen: handles particle decays in a user-configurable library
24. Diracxx: general root/c++ toolkit for computing Feynman amplitudes

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 than 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.

4 The Directory Structure

The BUILD SCRIPTS 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. BUILD SCRIPTS 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.

5 Scripts to Implement Build Scripts

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

/group/halld/Software/build_scripts

```
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.

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.

6 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.

6.1 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 `$XERCESROOT/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.

6.1.1 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 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.

6.1.2 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 set 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 9 for the details.

Package	Home Directory Variable
ROOT	ROOTSYS
JANA	JANA_HOME
sim-recon	SIM_RECON_HOME
HDDS	HDDS_HOME
CERNLIB	CERN_ROOT
Xerces-C	XERCESCROOT
CCDB	CCDB_HOME
Geant4	G4ROOT
EVIO	EVIOROOT
RCDB	RCDB_HOME
HDGeant4	HDGEANT4_HOME
hd_utilities	HD_UTILITIES_HOME
gluex_root_analysis	ROOT_ANALYSIS_HOME
AmpTools	AMPTOOLS_HOME
SQLiteC++	SQLITECPP_HOME
SQLite	SQLITE_HOME
gluex_MCwrapper	MCWRAPPER_CENTRAL
halld_sim	HALLD_SIM_HOME
halld_recon	HALLD_RECON_HOME
LAPACK	none
HepMC	HEPMCDIR
PHOTOS	PHOTOSDIR
EvtGen	EVTGENDIR
Diracxx	DIRACXX_HOME

Table 1: Packages and their home directories.

```

export GLUEX_TOP=/home/luex/luex_top
export BUILD_SCRIPTS=$GLUEX_TOP/build_scripts
export HALLD_HOME=/home/username/sim-recon
source $BUILD_SCRIPTS/luex_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/luex/luex_top. Note that GLUEX_TOP and BUILD_SCRIPTS are defined explicitly rather than letting them default.

```

export GLUEX_TOP=/home/luex/luex_top
export BUILD_SCRIPTS=$GLUEX_TOP/build_scripts
source $BUILD_SCRIPTS/luex_env_version.sh \
    /home/username/my_versions.xml

```

Figure 3: Example of a set-up script driven by a private version file. See Section 8 for the format of the file.

6.1.3 Cleaning the Environment: `luex_env_clean.(c)sh`

Since `luex_env.(c)sh` is sensitive to definitions hanging around in the environment, there is a script provided that will undo all GlueX-related definitions: `luex_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.

6.2 High-Level Environment Set-Up

6.2.1 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 `luex_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 `luex_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.

6.2.2 Using the Versioning System

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 `luex_env.(c)sh`. The versioning system is described in detail in Section 8, 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 set file is shown in Fig. 4. Alternate combinations of package versions can be tried by making alternate versions of the version set file.

6.2.3 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 current default version set file, `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
```

Optionally, these commands will take an argument naming a version set file to use rather than the current default. For example, for bash:

```
source /group/halld/Software/build_scripts/gluex_env_jlab.sh \  
/home/username/my_versions.xml
```

6.2.4 Simple Environment Set Up

The simplest (and recommended) form of environment set-up under BUILD SCRIPTS is the `gxenv` command. To define this command, use `gluex_env_boot_jlab.(c)sh` (at JLab) and `gluex_env_boot.(c)sh` (for `gluex_install` builds). These scripts also define the `gxclean` command. `gxenv` is used to set up the GlueX environment and `gxclean` is used to reverse the set up. The scripts also provide an environment variable, `HALLD_VERSIONS`, that points to the directory containing all of the official version set files (the `version_*.xml` files).

How to use the scripts to define the commands.

For tcsh, include the following line in your `.cshrc`, `.login`, or `.tcshrc`:

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

at JLab and

```
source /path/to/gluex_top/gluex_env_boot.csh
```

for `gluex_install` builds.

For bash, include the following line in your `.bashrc`, `.profile`, or `.bash_profile`:

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

at JLab and

```
source /path/to/gluex_top/build_scripts/gluex_env_boot.sh
```

for `gluex_install` builds.

Note that these “boot” scripts *do not* set up the GlueX environment. They merely define the command that will do the set up.

How to use the commands.

Usage is the same for both tcsh and bash, JLab and `gluex_install`.

To Set-Up the Environment: `gxenv`

- To get the default environment:

```
gxenv
```

- To get the environment for a particular official version set, say `version_3.5_jlab.xml`, use the `HALLD_VERSIONS` environment variable:

```
gxenv $HALLD_VERSIONS/version_3.5_jlab.xml
```

- To get the environment for an arbitrary xml file, use its path:

```
gxenv /path/to/my_version.xml
```

To Take-Down the Environment: `gxclean`

To clear all of the GlueX-specific parts of the environment, including GlueX-specific path elements (*i.e.*, those in `PATH`, `LD_LIBRARY_PATH`, and `PYTHONPATH`):

```
gxclean
```

This is not guaranteed to completely clean all gluexy things in all circumstances, but should work well when using `gxenv` to do the set-up.

Why implement things this way and not just do the set-up in the `.*rc` file? The underlying set-up command, `'gluex.env_jlab.(c)sh,'` has become a bit heavy. You may not want to have it executed everytime you start a new sub-shell. Also having the version set hard-wired in your initial environment can be confusing if you change version sets depending on what you are working on. For these reasons, putting the set-up in the global shell set-up is not recommended. Sourcing these scripts is very fast and once sourced, it is easy to switch from one version set to another, even from the command line.

7 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 `BUILD SCRIPTS` system makes a choice of build options for each package so that the user need not master these details.

7.1 The Makefiles

The `BUILD SCRIPTS` 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 4. 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.

7.1.1 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: gluex_pass2
```

```
gluex_pass1: xerces_build lapack_build cernlib_build root_build \
    geant4_build evio_build sqlite_build sqlitecpp_build \
```

```
rcdb_build ccdb_build
```

```
gluex_pass2: gluex_pass1 jana_build hdds_build amptools_build \  
  hepmc_build photos_build evtgen_build halld_recon_build \  
  halld_sim_build hd_utilities_build hdgeant4_build \  
  gluex_root_analysis_build mcwrapper_build
```

The `all` target builds all officially supported packages. These are broken into two sets, `gluex_pass1` and `gluex_pass2` to solve the issue of packages that can only be set-up after they are built. This is a problem when starting from scratch. In that case the build must proceed in four steps.

1. Initial environment set-up.
2. Build the `gluex_pass1` target.

```
make -f $BUILD_SCRIPTS/Makefile_all gluex_pass1
```

3. Final environment set-up using contents of packages built in pass 1.
4. Build the `gluex_pass2` target.

```
make -f $BUILD_SCRIPTS/Makefile_all gluex_pass2
```

`Makefile_all` should always be invoked from the `$GLUEX_TOP` directory.

If you would like the build to proceed in parallel (for those packages that support parallel builds), set an environment variable (eight threads in this example):

```
export NTHREADS=8
```

or set the variable on the make command line. For example:

```
make -f $BUILD_SCRIPTS/Makefile_all gluex_pass2 NTHREADS=8
```

7.1.2 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 4. 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 7.3

7.1.3 Special Package Target

The target `cernlib_debug` is non-standard. If this target is invoked, it will 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 6.2.1) or by using the `home` attribute in the `package` element of a version set file (as described in Section 8).

7.2 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.

7.2.1 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 8.4. Note that the version variable can be set on the make command line as well.

7.2.2 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`.

7.2.3 Extra Features for Specific Packages

SCons options for `halld_recon`, `halld_sim`, and `sim-recon`

For the `halld_recon`, `halld_sim`, and `sim-recon` packages, there are variables, one per packages, that can be defined, either in the shell environment or on the make command line, that will supply optional arguments to the native `scons` command invoked by `make`. They are

```
HALLD_RECON_SCONS_OPTIONS
HALLD_SIM_SCONS_OPTIONS
SIM_RECON_SCONS_OPTIONS
```

respectively. For example,

```
make -f $BUILD_SCRIPTS/Makefile_halld_recon \
      HALLD_RECON_SCONS_OPTIONS='SHOWBUILD=1 -j4'
```

will cause SCons to show the compiler commands explicitly and build `halld_recon` with four threads.

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.

Forcing use of Python 2 for HDGeant4

For systems that do not have Python 2 as the default, setting the make variable `PYTHON2_EXPLICIT` will force the use of Python-2-specific versions of the `boost-python` library and the `python-config` command in `Makefile_hdgeant4`.

7.3 Adding New Package Builds to an Existing Tree

There are two ways:

7.3.1 Use the Top-Level `Makefile_all`

After setting the desired values of the version environment variables and/or the URL environment variables (see Section 7.2 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.

7.3.2 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.

7.4 Building Custom Versions of Packages

7.4.1 my_halld_build_jlab

Script location at JLab:

```
/group/halld/Software/build_scripts/my_halld_build_jlab
```

Usage message:

```
my_halld_build_jlab [-h] [-x filename] [-n nthreads] [package_1] [package_2] ...
```

where:

```
-h          show this help text
-x          set name of version set XML file
            (default: /group/halld/www/halldweb/html/dist/version_jlab.xml)
-n          set number of threads to use in builds (default: 1)
-k          keep potentially inconsistent packages in environment
            (default: remove them from the environment)
```

package_n package name, choose from:

```
hdds
sim-recon
halld_recon
halld_sim
hdgeant4
gluex_root_analysis
```

if no package name supplied all except sim-recon will be built

Examples:

1. Build all GlueX packages (hdds, halld_recon, halld_sim, hdgeant4, and gluex_root_analysis):

```
my_halld_build_jlab
```

2. Build halld_recon only, using hdds from the default global build:

```
my_halld_build_jlab halld_recon
```

3. Build halld_recon, halld_sim, and hdgeant4, use hdds from the global build defined in version_3.1_jlab.xml:

```
DIST=group/halld/www/halldweb/html/dist
my_halld_build_jlab -x $DIST/version_3.1_jlab.xml \
  halld_recon halld_sim hdgeant4
```

4. Build halld_sim, using halld_recon from the default global build, using 4 threads:

```
my_halld_build_jlab -n 4 halld_sim
```

5. Build halld_recon, using hdds from the default global build, and retaining environment set-up of halld_sim, hdgeant4, and gluex_root_analysis from the default global build even though the latter three were built against a different version of halld_recon

```
my_halld_build_jlab -k halld_recon
```

7.4.2 my_halld_update.py

Usage message:

```
usage: my_halld_update.py [-h] [-x XML] [-n NTHREADS]
                        [inputPackages [inputPackages ...]]
```

positional arguments:

```
inputPackages          list of packages to update
```

optional arguments:

```
-h, --help            show this help message and exit
-x XML, --xml XML     version set xml file name (default: version.xml)
-n NTHREADS, --nthreads NTHREADS
                        number of threads to use in rebuild (default: 1)
```

Examples:

1. Update all private packages as identified by version.xml in the current working directory. Private packages contain a "home" attribute:

```
my_halld_update.py
```

2. Update halld_sim only:

```
my_halld_update.py halld_sim
```

3. Update all private packages as identified by /home/user/project/version_special.xml:

```
my_halld_update.py -x /home/user/project/version_special.xml
```

8 The Versioning System

8.1 Version Set File Format: an XML

The versioning system uses an XML-formatted version set 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.

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.

```

<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 set file. `version_1.7.xml` is shown.

8.2 Setting Up the Environment with a Version Set File

As we saw in Section 6.1, environment setting via `gluex_env.(c)sh` is sensitive to the definition of the package home variables. In Section 7.2, 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 set 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 the corresponding package makefile keys off it (see Section 7.2). 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`.

8.3 Specifying Alternate Source Code Sources with a Version Set File

We saw in Section 7.2 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


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

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

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.

8.4 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 set 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.

8.5 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.

8.6 Public Version Set Files

The public version set files are stored in a GitHub repository, JeffersonLab/halld_versions, available at

```
https://github.com/JeffersonLab/halld_versions
```

At JLab the repository is checked out in

```
/group/halld/www/halldweb/html/halld_versions
```

and can be access on the web at

```
https://halldweb.jlab.org/halld_versions/.
```

The names of the files are of two main types, (1) version sets that correspond to a periodic package update release and (2) version sets that correspond to a “launch” over GlueX data.

1. **File names for package update releases** are of the form `version_i.j.k.xml`, where `i`, `j`, and `k` are a major version number for the version set, a minor version number, and a sub-minor version number. For example, `version_4.1.0.xml`.
2. **File names for launches** are of the form `launch-name_i.xml`, where `launch-name` corresponds to the name assigned to the launch and `i` is a version number. For example `recon-2018_08-ver00_1.xml`.

9 The Prerequisites System

Each package may or may not have a build dependency on other packages managed by `BUILD SCRIPTS`. 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 set 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:

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.

```

<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`.

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. To stop the version check from being performed, set the environment variable

`BUILD_SCRIPTS_CONSISTENCY_CHECK`

to the value `"false"`.

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.

10 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.

10.1 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 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 zip file and a tar file with the scripts described here are available at

`https://github.com/JeffersonLab/gluex_install/releases/latest`

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.

Note that if parallel builds are desired (for the packages that support them) set the NTHREADS environment to the number parallel processes desired. Since `gluex_install.sh` invokes `Makefile_all`, the value will be used. See Section 7.1.1.

10.2 Using the Build

After the build is complete, to set up the environment to use it, follow the steps in Section 6.2.4 that are designated for `gluex_install` builds.