LBNE Plan for Software Installation of LArSoft-based code

Ben Morgan, Maxim Potekhin, Liz Sexton-Kennedy, Brett Viren

December 10, 2014

Contents

1 Introduction

LBNE must be able to install its required software from source code on all major collaboration platforms[1]. Whilst binary deployment systems such as `cvmfs` or those used by RPM or Debian based distributions, are useful, a simple, portable and automated from-source build mechanism must be provided in order to easily create these binary packages and to ease code development tasks. The single, largest hindrance in satisfying this requirement that has been encountered by LBNE is the design and implementation of the low-level CMake-based build system currently in use by the LArSoft and art packages. These packages provide the major components for the detector simulation portion of the LBNE software stack and thus this hindrance has had a large impact on the progress of LBNE.

The main element of the LArSoft/art build system that is problematic is the tight entanglement it has with the high-level `UPS` end-user environment/package management system. Rather than having a configuration system give configuration information to a build system, as the case in all industry standard systems such as `rpm/deb/ports/homebrew`, the LArSoft/art build system *takes* configuration from `UPS`. This inversion of the usual configuration-build hierarchy makes it impossible to build/run the O(2000/10) C++11 sources/applications of LArSoft/art without replicating the entire `UPS`-based software stack, down to the compiler level. Though possible, this replication is difficult due to the installation process being driven by a highly complex system of undocumented scripts, both generated and hand coded in various languages (shell, Perl and Python). It has been found that these factors essentially prevent porting LArSoft/art to different systems, even those that provide compatible C++11/14 compilers and the required third party software "out of the box".

The heart of the problem, this inverted-dependency between the CMake build and `UPS` configuration, was pointed out to the art development team in April 2013, and in subsequent meetings both informal and official. The art team disagrees that this is a problem. So we were left with a technical impasse, and we agreed to disagree on this point and all that follows from it. However a plan was put forward in which LBNE would develop an alternative solution that did not suffer from this problem, propose it, and then the art team would consider it for adoption.

It should also be noted that LBNE's criticisms are not unique. Other potential adopters of art (including the CAPTAIN and SuperNEMO experiments) have rejected it due to the complexity of the installation procedure despite its apparently small source footprint (more on this below).

The strategy of the solution is in two parts: The first part is to decouple the art/LArSoft CMake scripts from `UPS` by rewriting these using pure CMake functionality, hence increasing the portability and usability of LArSoft/art. The second part is to remove the configuration management logic and data that resided in the `UPS`-entanglement and move it into a higher-level layer in the form of a *Worch* configuration. This strategy has already been proven to work in an initial conversion of the art packages and subsequent application to LArSoft packages. The rest of this document describes more about the current status of this effort, a plan for carrying forward this strategy and a rough time-line.

2 The FNAL art/LArSoft Software Stack

2.1 Source and Dependency Footprints of art/LArSoft

Both packages contain small amounts of C++11/14 source code (including all unit testing code):

- FNAL foundation libraries contain in total O(200/200) C++ header/source files
- art contains O(400/400) C++ header/source files
- LArSoft packages contain in total O(500/600) C++ header/source files

These in turn use the following standard and widely available third party packages:

- Boost
- SQLite3
- ROOT
- CLHEP
- TBB

and only in LArSoft:

- Geant4
- GENIE

These source/dependency footprints should be contrasted with core HEP packages such as:

- ROOT O(10000) sources, $>$ 10 external/optional dependencies
- Geant4 O(7000) sources, O(10) external/optional dependencies

These numbers are given to demonstrate that art/LArSoft are very simple and lightweight packages by modern standards. It should also be noted that neither art nor LArSoft have a large technical footprint. That is, they follow the C++11/14 standard, and thus should not be tied to specific architectures nor compilers. These features should make art/LArSoft easy to build and install on any system providing a C++11/14 compliant compiler plus the requisite packages. LBNE's experience has been that this is not the case due to the coupling of the build system to the `UPS` configuration management system, yet there is no technical reason for this coupling to exist. Neither ROOT or Geant4 require a specific configuration management implementation to locate their required packages, making them highly portable and easy to install despite their significantly larger source/dependency footprint.

This is not to say that a configuration management system is not required to help in integrating and managing an overall software stack. Rather, the build systems of the packages comprising that stack should not depend on a configuration management system being present, nor that it has a specific implementation. This follows the basic software engineering principle of separation of concerns.

2.2 The `UPS` Environment Management System

The `UPS` system provides software as "products" which are collections of files with binaries for each OS/architecture/compiler/optimization level/debug/profiling combination installed in different directories. A product also contains table file(s) with metadata on the product and which other products it depends on. Configuration of packages for use by a user is through a series of environment variables, not only the UNIX `PATH` variables but also many undocumented per-package level variables. Though limited documentation exists for `UPS`, it is out of date and poorly broken down into sections for beginner and experienced users.

Although `UPS` is not ideal (it is noted that Fermilab's own patterns of use of `UPS` gives evidence of this) its use as the configuration management system is in itself not a blocker for LBNE. Rather, it is the way that a hard reliance on `UPS` has been built into the tools used to build art, LArSoft and their client packages. Clients of LArSoft/art, not only LBNE, are therefore unable to even **build** these packages without a full `UPS` system replicated and configured locally, **even if the local system provides all required packages already**.

It is noted that whilst `UPS` products are available through a `cvmfs` repository, the intent and purpose of `cvmfs` is as a **deployment** system, **not** a build system nor package manager. It is highly useful for distributing software efficiently to a limited set of platforms, but it provides no utility for building nor packaging that software easily and cleanly or those or any other platform (and nor should it).

2.3 The `cetbuildtools` CMake Add-ons

LArSoft/art use the CMake build tool to configure, build and install their runtime/development products. Many large software projects such as ROOT, Geant4, LLVM and KDE (among others) have adopted CMake due to its ease of use and portability, to quote from CMake's website:

CMake is a family of tools designed to build, test and package software. CMake is used to control the software compilation process using simple platform and compiler independent configuration files. CMake generates native makefiles and workspaces that can be used in the compiler environment of your choice.

To build a package, such as ROOT, using CMake to generate `Makefile`s, all one does is

```
$ cmake <args> /path/to/sourcedir
$ make -j4
$ make install
```

Custom options and configuration information, e.g. paths to needed packages, can be passed through the command line arguments `<args>`. The art/LArSoft packages should be similarly easy to build/install, but they are not for one key reason: they require use of FNAL's `cetbuildtools` add-ons for CMake. As an additional build-time dependency, `cetbuildtools` breaks the build and use of art/LArSoft because of its design and implementation:

- `cetbuildtools` is tightly coupled to FNAL's `UPS` configuration management system for finding things like GCC, Boost etc.
- This coupling and reliance on `UPS` is such that a user trying to build a package using `cetbuildtools` has no way to make it use system or any other non-`UPS` installs of the required packages, even if these meet all version requirements.
- `cetbuildtools` is highly specialized on using GCC as the compiler, and subsequently code in art has been found to contain GCC-isms and code non-compliant with the C++11/14 standard.
- If a package **A** uses `cetbuildtools`, then a package **B** which uses **A** will be required to also use `cetbuildtools` (and thus `UPS`). This makes decoupling any client of LArSoft/art from `UPS` and `cetbuildtools` via a build-time "firewall" very difficult to implement.
- Most functionality in `cetbuildtools` demonstrates a fundamental lack of understanding of CMake and its capabilities/limitations (including package finding, import/export targets, target properties and globbing).
- Much of the `cetbuildtools` functionality is in the form of undocumented, heavyweight wrappers around core CMake functions, making the tool *more difficult* to use. These wrappers also enforce source and binary layout conventions on the user which are of little benefit for either development or runtime use cases.

In short, `cetbuildtools` fails to implement a portable and easy to use build interface. Any project using `cetbuildtools` is unbuildable without an entire local replication of a `UPS` stack. This is an inversion of the usual hierarchy used in industry standard build systems, e.g. a `Makefile`, sitting under a configuration/packaging system, e.g. RPM.

3 New Components

3.1 The *Worch* build automation tool

The purification process leaves us with a package-level build system which is no longer entangled with a build configuration system, as is best practice. Instead a higher level build configuration and automation system based on *Worch* [6] is utilized.

Worch is a general purpose task configuration and automation tool with a focus on developing a cohesive build system for an entire suite of software.

Unlike the `UPS`/CET build system it can be used to orchestrate the production of all of LBNE software instead of just being limited to art based packages.

Worch is an extension of an existing, proven tool called `waf` [7]. `waf` provides a low-level build system akin to CMake (although it does not rely on Make) and it is used to build large, high profile projects such as Samba. In addition, `waf` provides facilities to extend it to create meta-build systems which work by driving whatever native build system may exist for a given package (eg, CMake or autoconf). It is exactly this support that *Worch* exploits. The "heavy lifting" is done by `waf` and *Worch* merely adds a simple, declarative build configuration language and the `waf`-based "glue code" to interpret this language into waf installation tasks. `waf` does the rest. The set of suite-specific *Worch* configuration files (and any suite-specific waf extension modules) are then all that are needed to fully specify and define the production of a release of a complex software suite. Placing these in a code repository allows a single tag to precisely specify a release for the entire software stack used by the experiment.

The Fermilab build system does not provide a concise method to define a release and instead spreads configuration management across many scripts, generated and hand written. *Worch* allows for precise description of the build configuration information in a single text file or factored out, where it is convenient, into a few domain-specific ones. These simple text files completely define what will be part of the release. This includes which packages, their versions and file system layout of the results. This configuration mechanism naturally lends itself to suite-wide release management mechanisms based simply on keeping the configuration files in their own code repository.

Unlike the Fermilab build system, *Worch* does not place any requirements on the packages it builds. Because it does not assume any policy but rather allows for policy to be expressed in the configuration, it can produce a variety of build products from the same source and at the same time. For example, it can be used to build `UPS` tarballs, Environment Modules, RPM and Debian packages or others.

Also in contrast to the Fermilab build system which is a collection of intermixed scripts and configuration information, *Worch* follows a design that can be easily extended and in a way where extensions may be shared by different projects. *Worch* makes better use of build hardware which is important for large software suites. Besides being automated if allows for inter-package parallelism. Tasks run as parallel as possible limited only by their inter-dependencies and the available hardware.

Worch is a general purpose tool that will make managing the production of releases of complex suites of software for multiple, partly related experiments far easier than how things are currently done by Fermilab. *Worch* is also simple enough that even groups with relatively modest software stacks can benefit from the automation and configuration management that it provides.

3.2 Current Status

The current status of the "purification" of the low-level CMake build system is described. Here the art packages are `cpp0x`, `cetlib`, `fhicl-cpp` and `messagefacility` and art itself, and LArSoft refers to its current count of ten packages.

- An LBNE GitHub organization[2] has been established as an interim center of development. This is done to minimize disruption that might otherwise be caused for the current users and developers of `lbnecode`, LArSoft and art.
- The art repositories are forked into this organization in a way that "upstream" commits pushed to Fermilab Redmine repositories continue to be tracked. Git's decentralized nature also means that commits on the GitHub repositories can be pulled into the Redmine repositories in the future.
- A new `FNALCore` package[3] is developed that aggregates the art packages (except the main art package itself) as well as holds their purified CMake files. An aggregation has been performed due to the heavy interdependencies between these packages and their intent as a foundation library for art.
- Purified CMake files are developed for art itself in the `fnal-art` repository[4].
- The `lbne-build` repository[5] was created in the LBNE GitHub organization. It houses a *Worch* configuration and tools to build all the 3rd-party external packages, `FNALCore` and `fnal-art` from source.
- The patterns applied to art packages have been pushed up the stack through LArSoft and `lbnecode`.
- A `python-ups-utils` package[8] was developed to provide installation methods for binary UPS tarball packages as well as initialize a green field as a `UPS` products area, including downloading, building and installing `UPS` itself.
- A `worch-ups` package[9] was developed which provides *Worch*/`waf` extensions to support the creation of binary `UPS` tarball packs from the results of *Worch* builds.
- Support for `worch-ups` has been added to `lbne-build` with some initial testing complete.
- Building these packages with *Worch* has been tested on at least Ubuntu (14.04) and Scientific Linux (6.4) and in a by-hand manner on Mac OS X.

3.3 Plan

The plan going forward is meant to satisfy these goals:

- Push the commits of the purified CMake work into "upstream" repositories so that they no longer need to be held in separate tracking forks.
- Minimize disruption on the user base and provide an partly adiabatic change.
- Provide time for ongoing testing and improving of the purified CMake files while furthering and allowing the other goals.

The plan is in three major parts:

- Continue to apply the CMake purification up through the LArSoft and the LBNE-specific `lbnecode` packages. In the same manner as with `fnal-art`, push commits to GitHub in forks which track their upstream repositories and in step add to codelbne-build support to build each newly purified package. During this phase, *Worch*-related development is also needed in order to create `UPS` binary product "tarballs" from the build results and thus retain user-level status quo in the end.
- Change over from GitHub-based repositories to pushing commits to upstream repositories (in Fermilab Redmine/git). Do this by first purifying `lbnecode` as above (and in GitHub) and then porting these changes into the `lbnecode` Redmine git repository with all changes placed behind a "switch" that defaults to the `UPS`-entangled build. Factor `lbne-build` to support building this "switched" pure-CMake `lbnecode` package against dependencies provided by `UPS`.
- With acceptance (and hopefully assistance) by the LArSoft group, continue porting the CMake purification, still kept switched off by default, to the LArSoft Redmine repositories and updating `lbne-build` to follow suit. Then, do likewise for the art packages. At some suitable point "flip the switch" so the entire stack is built in a pure-CMake manner with *Worch*.

3.4 Interaction with other efforts

Up until step three, this effort does not interfere with others. At step three, buy-in by LArSoft and art developers and the Fermilab software builders is required. However, before even making significant process on step one it must be determined if Fermilab will accept the changes that will be made in steps two and three. If not accepted, LBNE will revise this plan since it would be easier to remove the backward compatibility features, including removing `UPS` entirely, and use a git merging strategy to maintain the LBNE alternative build system.

4 Work Schedule

Due to the decision making and buy-in process described above, a precise timeline for rollout of the plan cannot be given, however, we can provide estimates of effort needed for a specific, non-comprehensive set of items:

- Complete purification of LArSoft/`lbnecode` (largely done) and test : 1 week FTE
- Further testing of *Worch* feature to produce `UPS` tarballs: 1 week FTE
- Port purification from GitHub repositories to Redmine ones (lbnecode, larsoft, art): 1 week FTE

Estimates are integrated coding time and do not take into account necessary discussions, assume that art and LArSoft developers agree to this approach and with not contingency to account for unforeseen technical problems. Also not included is defining and enacting a campaign of testing and validation including integration into the Fermilab Jenkins continuous integration system.

References

[1] "LBNE Software and Computing Requirements", LBNE DocDB 8035.

[2] `https://github.com/LBNE`

[3] `https://github.com/LBNE/FNALCore`

[4] `https://github.com/LBNE/fnal-art`

[5] `https://github.com/LBNE/lbne-build`

[6] `https://github.com/brettviren/worch`

[7] `https://code.google.com/p/waf/`

[8] `https://github.com/brettviren/python-ups-utils/`

[9] `https://github.com/brettviren/worch-ups`