Managing a LArSoft Development Environment with Google Repo

BV

[2014-07-25 Fri 20:09]

This topic describes how to set up a development environment for modifying LArSoft code or developing packages built on top of LArSoft. The methods here do not make use of "`mrb`" but do rely on a "Fermilab compatible" UPS products area which provide the packages satisfying any dependencies any of the development packages require.

The development area consists of these directories:

`source/` a directory where all repositories are cloned

`build/` a directory from where building the source is done

`install/` a directory where the built binaries are placed

Each of these directories are independent from each other and may be placed where convenient. Their contents are managed by the various steps as described below. In the examples below, the use of these as relative directory paths indicate you should use whatever path locates them. Where an absolute path is required, the absolute path `/path/to` will be prepended. For example:

```
$ cd source/
$ cd /path/to/build
```

1   Manual Setup

This section describes how to produce a development environment for a single package, "lbnecode" a manual manner. It uses as low-level methods as reasonable but stops before exposing the user to the morass of the underlying UPS/CET/CMake build system.

1

1.1 Preliminaries

Make the three areas, here assuming they are all next to each other:

```
$ mkdir source build install
```

The `install/` directory will need to be primed with some UPS files which can be copied from the central UPS "products" area for the site. This example assume you are using BNL's RACF.

```
$ cp -a /afs/rhic.bnl.gov/lbne/software/products/.up[sd]files install/
```

1.2 Source

Clone the package repository into the source area

```
$ cd source/
$ git clone http://cdcvs.fnal.gov/projects/lbnecode
```

Note, this URL allows anonymous cloning of the repository but does not allow any commits that you may make to be pushed. Later, if desired, you can add an additional remote that allows pushes so any commits you may make can be shared.

From this clone, decide which tag or branch to start with. If you don't know what is available you can query the repository like:

```
$ cd source/lbnecode/
$ git tag
$ git branch -a
$ gitk --all
```

A likely starting point is either the "master" or "develop" branch or some tag. If you intend to have other people use your modifications then starting from "master" is wise as it will make future merging easier than if you start from an older tag. However, if you want to explicitly modify the code from some past tagged release, of courses use that tag next.

Once a starting point is found use its label in place of "`<branch-or-tag>`" in the following "`checkout`" command. If not given then the default branch will be assumed.

```
$ git checkout -b feature/MYWORK [<branch-or-tag>]
```

The "`feature/MYWORK`" follows the convention for holding development. Pick a unique name for "`MYWORK`" that indicates the intent of the development. It should be brief but evocative. It need not include any identifier as to who will be doing the work as any commits to the branch will, as always, will be attributed to your identity.

1.3   The version lie

The build, which is described below, will result in a UPS "product" binary package holding files for any executable, library, include, etc produced by the build. This package lays out its files in a pattern that includes a version string and this version string must be supplied to the UPS "`setup`" command by anyone who wishes to use these build outputs.

This version string is intended for release builds but it gets forced on you, the poor developer. It will be set to whatever value was needed at the time of the last release build that occurred in the branch you decided to check out above. Any modifications to the source past this release point will likely **not** modify this version and thus will create a lie. When reporting any results produced from a package that is in development you must take care not to quote them as being due to the release but qualify that the release was the basis for the development modifications.

With those caveats appreciated, you ignore the rest of this section.

To make plain that your development build is not a release you may modify the release string. It is set here:

```
$ emacs source/lbnecode/ups/product_deps
```

Look for a line beginning with "`parent`". For example:

```
parent lbnecode v02_03_01
```

Modify this string to indicate the development. For example, tack on an identifier that can be associated with the branch name you chose above.

```
parent lbnecode v02_03_01MYWORK
```

The rest of the file can often be left untouched. If the development requires a new direct dependency it may need to be added.

3

1.4 Set up environment for building

Before any development is started and before the first build of the checked out code one has to provide a meticulously crafted environment for the brittle UPS/CET/CMake-based build system to work. This setup is site specific but in general it involves sourcing a shell script associated with a base UPS "products area" followed by sourcing one associated with the package being built.

```
$ source /afs/rhic.bnl.gov/lbne/software/products/setup
$ mkdir -p build/lbnecode
$ cd build/lbnecode
$ source /path/to/source/lbnecode/setup_for_development -p
...
env CC=gcc CXX=g++ FC=gfortran cmake -DCMAKE_INSTALL_PREFIX="/install/path" -DCMAKE_BU
```

The "`-p`" flag in the last source indicates a "profile" build variant is desired.

Take note of the "`cmake`" command echoed by this second script as it will be used later. In general, building with CMake is best done in a directory outside the source directory and specific to each package:

Also, take note of that this last sourced file will add files to your current working directory which is why it is important to run it from the directory where build outputs should go.

1.5 Build the package

Next, issue that cmake command which was echoed by `setup_for_development`. Take note to edit the absolute path for the install prefix to suit your desired layout.

```
$ cd build/lbnecode/
$ env CC=gcc CXX=g++ FC=gfortran cmake -DCMAKE_INSTALL_PREFIX="/path/to/install" -DCMAI
$ make
```

1.5.1 Using the build directly

After the "`make`" the "lbnecode" package is built into the "`build/`" direc-
tory. In sourcing the "`setup_for_development`" script your environment
was munged in order to locate the basic OS-level outputs of this build in-
cluding executable and library files. However, application-level files may not

be yet be found if their location depends on additional environment variables. Some examples:

FCL files these are located through the environment variable "`FHICL_FILE_PATH`". This variable may be defined already but may not explicitly contain any elements pointing in to the build area. It may contain the relative paths "`.`" and "`./job`" which may find FCL files while you remain in the `build/lbnecode/` directory

1.6 Install the package

The build products can be installed as a UPS "product" into the location specified by the `CMAKE_INSTALL_PREFIX` directive to the `cmake` command with:

```
$ make install
```

Note that you should observe the output of this command copying files into a location with a directory named with your modified version string as above.

1.6.1 Using the development UPS products area

If the `install/` area was prepped as a UPS "products area" as above then you will now have your own products area that you or anyone who can access it may use. Since it only contains the development packages you have built and relies on the packages from the central UPS products area one must set up the environment to tell UPS about both. You do this by prepending your products area to the `PRODUCTS` environment variable:

```
$ export PRODUCTS=/path/to/install:$PRODUCTS
```

This assumes you have already sourced the central site UPS "`setup`" script.

You can now see that your package is found by UPS, for example with:

```
$ ups list -aK+ | grep lbnecode | grep MYWORK
"lbnecode" "v02_03_01MYWORK" "Linux64bit+2.6-2.12" "e5:prof" ""
```

Where "`MYWORK`" is the label you added to break the version lie as shown in section 1.3. To munge your environment to use this package do the usual dance:

```
$ setup lbnecode v02_03_01MYWORK -q e5:prof
```

This will munge your environment to give precedence to the installation in your personal UPS products area. In particular, it will likely shadow any direct use of the build outputs. You can see this in the example:

```
echo $LD_LIBRARY_PATH|tr ':' '\n'
/path/to/install/lbnecode/v02_03_01manual-build/slf6.x86_64.e5.prof/lib
/path/to/build/lbnecode/lib
# ... central UPS products library directories
```

This means that you will need to do a full "make install" in order to access the build output following any development of the source.

You may also wish to examine "$FHICL_FILE_PATH" and determine that an absolute path into the "lbnecode" package in your personal UPS products area has been added.

```
$ echo $FHICL_FILE_PATH|tr ':' '\n'
.
./job
/path/to/install/lbnecode/v02_03_01manual-build/job
.
./job
# ...
```

Why are the "." and "./job" directories repeated? Dunno, don't ask me, I didn't write this.

1.6.2 Rebuilding and using with a personal UPS products area

As noted above, if you elect to munge your environment to use the contents of your personal UPS products area be aware that they will take precedence over the contents of your build area. This means you must do a "make install" before you may use the output of the rebuild.

If you are developing multiple packages at once this environment must also be kept in mind.

2 Using Google Repo

The above procedures in section 1.2 for preparing the source area can be carried out manually for each package that shall be developed in conjunction.

However, this can be tedious and error prone. Some things that must be kept in mind:

- clone a suite of package source repositories

- assure checkouts of each package are from a consistent point in their development history

- assure correct build dependencies are used

This last one include the need to correctly use the correct packages from the centrally installed UPS products except when these packages are subject to the development effort. For these packages one must build them in proper order and assure that their results are used when building subsequent packages.

2.1   Source preparation with Repo

To facilitate managing the source code for multiple packages the repo tool developed by Google for Android development may be used to automate some steps. This section walks through how to use Repo to prepare the packages providing the LArSoft source.

Repo is installed simply by downloading the tool into a location that is picked up in your "$PATH" like so:

```
$ curl http://commondatastorage.googleapis.com/git-repo-downloads/repo > ~/bin/repo
$ chmod +x ~/bin/repo
```

Repo uses a file called a "manifest". These should not to be confused with similarly named but unrelated files which are part of a certain binary UPS product installation method. A Repo manifest file is in XML format and describes the basic elements of a set of source repositories. In particular it states for each repository:

- the base URL

- the branch from which to checkout

- the local directory

- the repository name

These last two are usually identical and this last is appended to the base URL to locate the repository.

Repo expects the manifest to, itself, live in a git repository. This allows one to track development at a meta-level. Feature branches across multiple repositories can be grouped in a manifest file which is itself tracked in a branch of the manifest repository. Multiple manifest files may also be maintained.

LBNE maintains a manifest repository for LArSoft and it may be used with Repo as in this example:

```
$ cd source/
$ repo init -u https://github.com/drbenmorgan/larsoft-manifest.git
$ repo sync
$ ls -a
.   larana    lardata          larevt        larpandora  larsim
..  larcore   lareventdisplay  larexamples   larreco     .repo
```

Note, the same top-level directory location conventions from Section 1 are used here and this will become the source area for building.

After the "init" the manifest repository will be cloned into Repo's working area ".repo/". After the initial "sync" the package source repositories listed in the manifest will be cloned. These will be left checked out in a "headless" state not explicitly associated with a branch.

2.2   Initial repository checkouts

Each repository will need to be checked out to a local branch from a suitable starting location. This may be done by repeating the "git checkout" directions from section 1.2 in each repository. If there exists some symmetry among the repositories one may exploit it by applying the git checkout to each with a single command:

```
$ repo forall -c git checkout -b feature/MYWORK [<branch-or-tag>]
Switched to a new branch 'feature/MYWORK'
Switched to a new branch 'feature/MYWORK'
...
$ repo branches
*  feature/MYWORK            | in all projects
```

3 Building multiple packages

At this point one can manually follow the instructions from section 1.5 for each package honoring the correct build order. The UPS/CET/CMake build system encodes this ordering in the `ups/products_deps` file found in the source code. If one wishes to correct the version lie described in section 1.3 each package's `products_deps` file will need modification in order to change the value of the `parent` setting. In addition, the packages dependencies satisfied through UPS are enumerated in this file in terms of their product name and version. If any of these products will be supplied by the development build and with version strings that do not implement the version lie then these will need modification to match. These are listed under the line starting with `"product"`. For example the "larana" package has:

```
product        version
larreco        v02_03_01
gcc            v4_8_2
```

If the development of the "larreco" is reflected with a version string `"v02_03_01MYWORK"` then this same version string must be set here.