Fermilab

PU0000

Using Fermilab CVS Product Source Repositories

Marc W. Mengel

November 11, 1997

Abstract

The various CVS Product Source Repositories at Fermilab contain product source trees used to generate various Fermilab Software products. The product trees contain a Makefile at the top level which contains the information needed to build the product and make distribution images of it. The UAS archive contains a standard template product, to assist users who are developing new products or migrating existing products into a CVS Products Repository. Other tools are avaliable to assist users in large scale development of their software, and are described here as well.

Contents

1 Overview

The CVS Product Repositories at Fermilab have been created so that maintenance of products can be shared by various groups in Fermilab to facilitate effective and timely support of experiments, etc. This document will discuss:

- Tools Used in conjunction with the repository
- Accessing the repository
- Building a product extracted from the repository
- Standard Top-level Makefile Targets
- Checking product changes back into the repository
- Adding a new Product to the Repository
- Building on multiple platforms with buildmanager
- Tracking third party sources

2 Tools Used

There are several tools used in conjunction with the repository, the most important are *cvs* and *make*. These are tools to keep track of revision history of large groups of files, and to orchestrate the activities of a large software build, respectively. This section of the document will provide an introduction to these tools, for a more detailed look, the interested reader is refered to *Version Management with CVS* by Per Cederqvist, available at http://www.fnal.gov/docs/products/cvs/ and *GNU Make, A Program for Directing Recompilation* by Richard M. Stallman and Roland McGrath, available at http://www.fnal.gov/docs/products/gtools/

2.1 cvs

The *cvs* package is a tool to keep track of various versions of groups of files called *modules*. These groups of files are assumed to be kept in a directory tree, and a master copy of these files with revision history is kept in a *repository* which is potentially shared by various developers. A developer can *check out* a copy of a module, make changes to it, and then *check in* or *commit* those changes to the repository. A particular version of a module can be given a *tag* by which it can be retrieved later. For example, if you had a document which consists of 4 files, doc, ch1, ch2, and ch3, you could add them to the repository as a module using the commands:

% mkdir docmodule
% mv doc ch1 ch2 ch3 docmodule
% cd docmodule

% setenv CVSROOT cvsuser@hostname:/repository
% cvs import docmodule fermilab initial_revision

Notice that we first moved the files into a module directory, changed our current directory to be that directory, set our CVSROOT environment variable to tell *cvs* what repository we want to use (in this case we specified a repository on a remote host hostname, accessed as user cvsuser and in directory /repository), and then issued the cvs command with arguments. The arguments to the cvs command in this instance probably need some explanation at this point:

- The first argument is the operation we want to perform – we want to import a group of exisiting files into the repository as a module.
- Next is the name of the module. This is the name we will need to check this module back out of the repository.
- The third argument is a "vendor tag", which in this case I am calling "fermilab". If this is something that is developed elsewhere, but that we are keeping a copy of in our repository, you might want to put the name of the third party from whom we got the files.
- The remaining arguments are tag names to call this revision of the files. I am calling this revision the initial revision.

Now the files are stored in our repository, and we can remove or rename our initial copy, and *check out* a "working copy", to which we can make revisions. The "working copy" will have some extra files stored in subdirectories named CVS where the *cvs* command will keep bookeeping information.

% cd ..
% mv docmodule docmodule.bak
% cvs checkout docmodule
% cd docmodule

Now if we make changes to our document, we can use the command

% cvs commit

to file this new version in the repository, and

% cvs tag *name*

to tag the most recently committed version as *name*. We don't need to specify what files are included in the module, or what repository to use when we're sitting in the "working copy" directory, because all of that information is squirreled away in the CVS bookeeping directory.

2.2 make

The *make* utility uses a file (usually called Makefile) which describes dependencies between files and the modification timestamp on files to determine what work needs to be done to generate a given result. For example, a given executable program may require two source files to be compiled into object files, and the two object files to be linked with a library to generate the exeutable program. That means that the program depends on the object files and the library, and the object files in turn depend on the source files. A description for *make* for that arrangement looks like:

program: file1.o file2.o library.a
cc -o program file1.o file2.o library.a
file1.o: file1.c
cc -c file1.c
file2.o: file2.c
cc -c file2.c

If you put this in a file Makefile and type

% make program

i t will run all three compiles. If you type the command again, it will do nothing. If you then modify file2.c and re-run the command it will only compile that file and do the link, but not recompile file1.c, thus avoiding unneccesary work.

The dependency rules have three parts. The left hand side is called the *target* which is the item we want to create or update. The right hand side is a *dependency list* which states other targets are prerequisites of this one. Commands on subsequent lines (identified by a leading tab) are the mechanism to generate the target from the prerequisites.

You can also define *macros* in a Makefile, to simplify maintenance and make the file more readable. For example, the two lists of object files in the preceding example really ought to be identical. We can enforce this by using a macro to hold the list of object files, and using it in two places, like:

DEPS=file1.o file2.o library.a
program: $(DEPS)
cc -o program $(DEPS)
file1.o: file1.c
cc -c file1.c
file2.o: file2.c
cc -c file2.c

Similarly, we may want to be able to easily switch compilers, leading to:

DEPS=file1.o file2.o library.a
CC=cc
program: $(DEPS)
$(CC) -o program $(DEPS)
file1.o: file1.c
$(CC) -c file1.c
file2.o: file2.c
$(CC) -c file2.c

Finally, we may have targets whose commands don't actually create a file. These targets will cause their commands to be executed each time they are referenced, since the target is *never* up to date. This is commonly used to provide utility targets, like ones that print the source code, or make distribution files, etc. which do not leave a visible result with a timestamp in the current area.

3 Accessing the Repositories

The repositories are accessed using **cvs** with a remote CVSROOT environment variable set, in particular to:

Repostiory	**CVSROOT value**	Product
UAS	cvsuser@dcldaa.fnal.gov:/usr/products/olsshare/cvs	uascvs
OLS	cvsuser@fndaub.fnal.gov:/cvs/ols	olscvs
SDSS	cvsuser@sdss.fnal.gov:/cvs/sdss	sdsscvs

This requires that your host and userid be registered with the cvs repository. So for example, if you were going to checkout the module **demo** from the UAS repository to work on it, and make changes based on version v1_2 , you would:

% setup cvs
% setenv CVSROOT cvsuser@dcdlaa.fnal.gov:/usr/products/olsshare/cvs
% cvs rtag -b -r v1_2 v1_2_custom demo
% cvs checkout -r v1_2_custom demo

This example makes a branch revision based on v1_2 of the product sources, so that any changes you make will be relegated to that branch. This is reccomended for any changes to released products made by folks who are not the primary developers of the product.

4 Building/Distributing a product from the Repository

If you have extracted a product from the repository, (like **demo** from the previous section) you can declare, build, and test the module with **make**, using the Makefile that comes with the product. For example to automatically declare the product to a local **ups** database, build it, test it, and put the product into the **kits** distribution area, you would:

% cd demo
% make build_n_test kits

which would do the entire procedure. Of course, sometimes you want to do smaller pieces of the whole proceudre, in order to reproduce a problem, make a repair, and test the repaired version before distirbuting the software. The following discussion shows how to do this:

4.1 Finer grained Makefile Targets

While sitting in the product toplevel directory, there are several make targets available to provide useful, and more fine-grained functionality, as well as various macros in the Makefile that you may want to redefine. These targets are useful both in a copy of the product checked out of the repository, and in an installed copy of the product. Makefile targets which are useful include the following.

All-uppercase words in the following are make macros which you may want to override on the command line.

declare declares a product to the **ups** database DPRODUCTS with chain CHAIN and version VERS, with dependencies DEPEND and flavor FLAVOR. For example, if you wanted to use the Makefile to declare a test version of the product, you could execute:
make CHAIN=test declare
or if you wanted to declare it in your home ups database $HOME/declared, you could
make DPRODUCTS=$HOME/declared declare

undeclare undeclares the above

all requires that the product be setup, and builds the software, and puts it in the appropriate places in the product (e.g. executables in the **bin** directory, libraries in the **lib** directory, etc). So if you had just obtained the sources for this product from a repository, you would need to declare this instance with make declare (above) and setup the product, and then do a make all.

clean removes intermediate files from the build, but leaves executables, etc. neccesary to use the product. This helps you save space, as well as letting you test that your product still works in distribution form.

spotless removes *all* generated files, so the product is like it was just checked out of the repository. The product is genrally not useable after this step.

test runs a test script, etc. to regression test the product, assuming any were provided by the product developer, of course.

distribution makes a distribution file (currently a tarfile) of the product suitable for distribution

kits installs the distribution file on distribution platform on ADDPRODUCT_HOST using **cmd addproduct.** There are some caveats here; namely ADDPRODUCT_HOST must be able to see the filesystem where the tarfile has been created, and have the ability to run the cmd addproduct command.

setversion sets the version string for the product in the Makefile and other files known to contain it, commits the sources into **cvs**, and tags them with that version as their tag.

install makes a local copy of the product, in directory LOCAL,

The following macros are used in these targets, which may be overidden on the **make** command line when using the above targets:

DIR the proper name to use for the current directory

PROD the product name for ups declares, setups, cmd addproduct, etc.

PRODUCT_DIR the shell variable created by doing **setup $(PROD)** for this product

VERS the version of this product instance

DEPEND dependency flags to give to **ups declare** when declaring this product

CHAIN chain name to use for **ups declare** or **cmd addproduct**

UPS_SUBDIR directory containing files for ups when doing **ups declare**

FLAVOR system flavor for declares, etc. Not usually overriden directly, see below

OS operating system component of flavor

QUALS flavor qualifiers part of flavor.

CUST customization part of flavor

ADDPRODUCT_HOST host to rsh to to perform product additions to the distribution platform

DISTRIBUTIONFILE name of distribution file (tarfile) to create.

LOCAL directory to make a product copy in when doing a **make install** copy of the product

So for example, to make the **make declare** target declare the product as a "test" instance, you could:

% make CHAIN=test declare

or to have the product declared locally, and added to kits with a qualifier of "+mips3", you could:

% make QUALS=+mips3 declare kits

5 Checking Changes back into the Repository

If you have made changes to the product, you want to have **cvs** commit those changes, and give an appropriate log message about what you changed, for example

% cvs commit -m "fixed the fizmumble bug"

6 Making a New Product for the Central Repository

To start a new product for the Central Repository, we reccomend that you start with the product **template_product** which is in the central repository. You should then fix the product name, customize a few items in the Makefile, and you should have a product you can check into the repostory. For example, lets say we have a product we want to distribute called **hello** which is the classic "hello world" program. We would start by exporting a copy of the template, and fixing the product name, etc.

```
% setup uascvs,template_product
% mdkir hello;
% cd hello
% CloneTemplate
```

Next we would want to add our hello world program to the Makefile; this involves editing the Makefile and changing the **all:**, and **clean:** targets to build our product. This would look something like:

```
all: proddir_is_set
        test -d bin || mkdir bin
        cd fnal/hello ; make PREFIX=../.. install
clean:
        cd fnal/hello; make clean
spotless: clean
        rm -rf bin
```

The actual source for the hello world program would go in a subdirectory "fnal/hello", which would have files like **hello.c:**

```
#include <stdio.h>

main() {
    printf(hello world\n);
    return 0;
}
```

and its own **Makefile:**

```
PREFIX=/usr/local/bin
all: hello
hello: hello.c
        cc -o hello hello.c
install: hello
        cp hello $PREFIX/bin
```

And then you can, from the top level of this file tree, **cvs import** this whole filetree as a new product, and check out the filetree

```
% cvs import demo local initial
% cd ..
% mv demo demo.bak
```

% cvs checkout demo
% rm -rf demo.bak

7 Building products with buildmanager on the build cluster

Now that we have a product source tree in the archive, we'd like to get it built and tested on our available platforms. While you can log onto each machine of the appropriate architecture, **cvs checkout** the product and build it by hand, the **buildmanger** product is available to automate this process. For example, to use buildmanager to build the demo product, you would need to put a tag on that version, and use the buildmanger executable to automate the build.

% setup buildmanager
% make setversion NEWVERSION=v1_3
% buildmanager demo v1_3

...