Producing Debian packages for MOOS

Christian Convey (christian.convey@navy.mil)

2007-08-01

Contents

1 Overview

Debian, and related Linux distributions such as Ubuntu, use the APT package management system to let users easily download and install software. We now produce Debian packages for MOOS and (eventually) IvP to make it easier for Debian / Ubuntu / etc. users to try out and work with MOOS/IvP software.

This document is both a HOWTO guide for producing MOOS's Debian package, and a design document explaining how MOOS's build system has been modified to support the production of Debian packages.

2 The Packages

Package name	Provides
moos-core	MOOSDB
moos-core-dev	MOOSLib, MOOSGenLib, header files
moos-essentials	pMOOSBridge, pAntler, pLogger, pScheduler
moos-essentials-dev	MOOSUtilityLib, header files
moos-instruments	iActuation, iLBL, iCompass, ...
moos-navigation-and-control	pHelm, pNav, iRemote
moos-navigation-and-control-dev	MOOSNavLib, MOOSTaskLib, header files
moos-thirdparty-dev	FLTKVW (library and headers), newmat headers
moos-tools	uPlayBack, uMS, uMVS
moos-tools-matlab	iMatlab

By Paul Newman's request, the packages were defined by the top-level directories within the MOOS source code tree. The only exception was `moos-tools-matlab`, which was separated out because of its unique build-time requirement (an installed copy of Matlab).

Documentation (`-doc`) packages weren't included because as of this writing, the `Doc` subdirectory within the MOOS source code tree is not under CMake control. Once each document in the `Docs` directory has its own `CMakeLists.txt` file, this omission can be easily addressed.

2.1 Locations of installed files

Because of my ignorance about MOOS's policy regarding which changes in release version numbers indicate or counter-indicate a change in MOOS's libraries' APIs, I took a very conservative approach to the location in which these Debian packages will install their files.

Rather than placing the files directly into `/usr/lib`, `/usr/include`, and `/usr/bin`, the Debian packages will place their files into `/usr/(version-number)/lib`, `/usr/(version-number)/include`, and `/usr/(version-number)/bin`. This reduces the possibility of problems stemming from unplanned API breakages and of confusion regarding which version(s) of the software are being employed at a given time.

Someone on the MOOS project who is also familiar with Debian packaging standards should probably review this decision as soon as is convenient.

3 Modifications to MOOS' Build System

3.1 Shell scripts now, CPack later

MOOS uses the CMake build system (version 2.4 is available as of this writing) to manage the build process. CPack is a complementary piece of software designed to produce software that will install onto target systems the software you're developing.

Producing Debian packages (in addition to Windows installers, .tar.gz files, etc.) is the exact kind of job CPack was designed for. Unfortunately CPack doesn't produce Debian files at the time of this writing. However, CPack 2.6 is expected to be released in late 2007, and assuming it's reasonably bug free, it should be the best tool for producing MOOS's Debian packages once it's widely available.

Several CMake modules have been written to provide Debian package creation abilities to projects that use CMake. However when I reviewed these I judged none of them to have the quality and/or features required for our project.

Therefore the first pass at producing Debian packages for the MOOS software is accomplished using a combination of enhancements to the project's existing CMake files, some (text) template files, and a Bourne shell script. The details of this approach are explained in the rest of this section.

3.2 Modifications to MOOS's CMakeLists.txt files

The only significant change I made to MOOS's `CMakeLists.txt` files was to add a number of `INSTALL(...)` directives to the individual programs' and libraries' `CMakeLists.txt` files. (That is, to the files at the leaves of the source code directory tree.)

CMake's `INSTALL` command supports the notion of an *install component*. An install component is a named subset of the files that would be installed if you ran the command `make install`. (The particular installation component that will be installed when you run `make install` is specified by the value of the `CMAKE_INSTALL_COMPONENT` CMake variable at the time you configure your Makefiles using `cmake`.)

I used named components to specify the different subsets of programs, libraries, header files, etc. that should appear in each of MOOS's different Debian packages.

3.2.1 Why use CMake named installation components?

I used named installation components because of the way `dpkg-deb` works. To use `dpkg-deb`, you first populate a directory tree to look the way you want your files to appear when someone installs your package. For example, if you're using /tmp/foo/ as a working directory while you do your packaging, you might create a set of files like this:

```
/tmp/foo/usr/bin/pAntler
/tmp/foo/usr/lib/libMOOS.a
```

You would then put a few other special files into the `/tmp/foo/` directory as required by the Debian package system, and then run the `dpkg-deb` program with the directory `/tmp/foo` as one of its command-line arguments.

It turns out that CMake's named installation components is very helpful in populating the directory tree whose root is `/tmp/foo/`. You just build MOOS, and then use a variant of `make install` to specify the installation component whose files you want to install (e.g., to appear in the Debian package you're producing) and tell the installer that you want the installation to use `/tmp/foo` as a base directory into which the installation occurs.

3.2.2 CMake installs: Specifying component and location

When you run `cmake` to set up a build tree / build system for your project, you can specify numerous details that become fixed within your Makefiles. These details (unlike some others) cannot be changed merely by setting certain environment variables prior to invoking `make`. You must re-run `cmake` in order to change these details. One such detail is the specification of which installation component will be installed when you run `make install`.

This is a problem for us, because as we build multiple different Debian packages we need to adjust which installation component will be installed when our packaging shell script invokes `make install`. There may be safe and acceptable ways to re-run `ccmake` each time our shell script moves on to produce a different Debian package, but none was obvious as I wrote the software. This led to the following solution:

CMake-generated `make install` make targets work in a perhaps surprising way. When you run `make install`, the Makefile invokes `cmake` and instructs it to execute a CMake script file (`cmake_install.cmake`) that was produced when you last used `cmake` to configure your Makefiles.

You can install the MOOS software by manually telling `cmake` to execute the `cmake_install.cmake` script, rather than doing so by running `make install`. In doing so you have the opportunity to change which installation component

will be installed. And this is exactly what our shell script

```
BuildScripts/DebianPackaging/make-debs
```

does.

This approach works, but it's undesirable in the long term for several reasons. First, it's complicated and relies on some underdocumented features of CMake. Second, according to the CMake team this approach is is unsupported, which is another reason to switch to using CPack 2.6 once it's widely available.

3.3 The make-debs shell script

I've created a new Bournse shell script within the project's source tree:

```
BuildScripts/DebianPackaging/make-debs
```

This script takes a lot of parameters, and it requires that you've already successfully built MOOS in some directory. But given those pre-requisites, this script should do a pretty good job of building all of MOOS's Debian packages and putting them into a directory you specify.

This script is manually invoked, rather than being executed to satisfy a Make target. For various reasons, attempts to fold this functionality into MOOS's CMake files to make the CMake files overly complex. I judged that keeping MOOS's build system simple and comprehensible was more important than finding some way to have the command `make debs` work.

3.4 File templates

The directory tree that contains the files to be packaged also must contain a few additional files that wouldn't normally be considered part of the proper MOOS project.

They are:

DEBIAN/control This specifies some details about the package that are pretty stagnant, such as a human-readable description of what the package provides, as well as some details that change frequently such as the version number of the software being packaged.

This is the one file in the package that will not by copied onto the file system of the target computer when the package is installed.

The format of this file is described in Section 5 of the Debian Policy Manual:

`http://www.debian.org/doc/debian-policy/index.html`

/usr/share/doc/(package-name)/copyright This is a copyright statement regarding the files in the package. As of this writing, all of MOOS's Debian packages use the same version of this file.

/usr/share/doc/(package-name)/changelog.Debian.gz This is a change log describing the changes that have occurred from one revision of the software to the next. This file has a very particular format, described Section 4.4 of the Debian Policy Manual.

These versions of these files that appear in the Debian packages are all generated from files in the MOOS source tree's

`BuildScripts/DebianPackaging/FileTemplates`

directory.

Each `CONTROL-...` file, one per Debian package, is copied and customized by `build-debs` to become a Debian package's `DEBIAN/control` file. `build-debs` generally modifies each of these files, as it's being copied, to inject into the file the desired version number of the package being built. This lets `build-debs` save a human the work of modifying these files as part of the Debian packaging exercise.

Currently there's only one copyright file:

`COPYRIGHT-mit-and-oxford-gpl`

I wrote this file based on the copyright statement I found at the top of one of the MOOS source code files. This file's exact text appears in the appropriate `/usr/share/doc/...` subdirectory of systems that install the MOOS Debian packages.

Each `CHANGELOG-...` file, one per Debian package, is copied and compressed by `build-debs` to become a Debian package's `changelog.Debian.gz` file. The `CHANGELOG...` files are meant to be maintained by a human each time the MOOS project is about to release a new version of its software. If these aren't maintained, users of the Debian packages will not be able to use the existing package management tools to review the package change history and to decide whether or not they want to install a new version of the MOOS packages.

4 Creating Debian Packages

4.1 Pre-requisites

Here's what you need on your computer to be able to produce MOOS' Debian packages:

Debian-based operating system The general wisdom is that to build Debian packages, you're best off doing so on a Debian-based (Debian, Ubuntu, etc.) operating system. Doing so ensures you have ready access to the tools needed to produce Debian package files (`dpkg-deb`, `lintian`, etc.)

Ability to build MOOS Building the MOOS software is an integral part of producing MOOS's Debian packages. Refer to MOOS documentation regarding what this entails.

Software used for creating packages You need the following programs installed on your system:

`dpkg-deb` This is used to create the actual Debian package files.

`lintian` This is used to confirm that a package you just build conforms to Debian's rather strict packaging rules.

Matlab (optional) The MOOS project includes a Matlab interface: a shared library named `iMatlab`. To build this shared library you need a copy of Matlab installed on your system (I'm not sure which version(s) of Matlab suffice).

If you don't meet this requirement you can still build most of MOOS's Debian packages; just not the `moos-tools-matlab` package, which provides `iMatlab`. `iMatlab` has been placed ni its very own Debian package so that if you don't own Matlab you're still able to produce Debian packages for every other piece of MOOS software.

4.2 Steps to Create Packages

1. Configure and build MOOS:

   Build MOOS. Ensure that the build system is able to build those subprojects which depend on the `FLTK` library. The current version of the MOOS system will simply skip those projects if it can't find the needed `FLTK` libraries and headers, as well as the related `fluid` program (which comes in a separate Ubuntu package, by the way).

   You should be sure to enable the building of every part of MOOS, with the optional exception of the `BUILD_MATLAB_TOOLS` piece. Otherwise some of the Debian packages may lack expected files and/or not build at all.

2. Create (or just use) to working directories:

   One directory will be where `make-debs` does its scratch-work. The other directory will be where `make-debs` places the completed `.deb` files.

3. `cd` to the `BuildScripts/DebianPackaging/` subdirectory within your MOOS source code tree.

4. Run `make-debs`

   Run the `make-debs` script. If you don't know what it's command-line arguments should be, run it without any arguments and help will be printed to the console.

   Note that if you didn't build the `iMatlab` library (that is, when you configured the MOOS build system you set `BUILD_MATLAB_TOOLS` to `OFF`), then you *must* use the `--no-matlab` option when invoking `make-debs`. Otherwise `make-debs` will fail when it tries to build the `moos-tools-matlab` package.

5 Suggested Near-term Future Work

- Review the package descriptions and dependencies as stated in the `CONTROL-...` files.
- Review the `COPYRIGHT-mit-and-oxford-gpl` to confirm that the copyright statement communicates what it's supposed to communicate.
- Review the decision to include the specific MOOS version number in the installation path for the packaged files.
- The `Essentials` and `Core` subdirectories have a strange kind of cyclic dependency:

  `Essentials` provides `MOOSUtilityLib`, on which the code in `Core` has a compile-time and link-time dependecy. This dependency, as well as its name, suggests that `Essentials` is an underpinning of the rest of the project's code.

  But `Core` provides the `MOOSDB` program, and `Essentials` has some programs (such as `pMOOSBridge`that have a *runtime* dependency on a running `MOOSDB` instance.

  Both of the names (`Essentials` and `Core`) indicate a kind of foundational role with respect to the rest of the project's code, but the dependencies described above, as well as their synonymous-seeming names, contradict that interpretation.

  I suggest that this is sorted out so that the two directories don't both have names that indicate the same, exclusive role in the project. I also suggest that the runtime vs. build-time dependencies decribed above are sorted out. One way to accomplish this would be to move those programs that connect to `MOOSDB` out of `Essentials` and into `Core`.
- MOOS's `Docs` directory isn't currently build by the CMake build system. Once it is, it would be easy to create one or more `-doc` documentation packages to accompany the other MOOS Debian packages. This would be

helpful to would-be MOOS users, as it reduces the burden needed to find useful documentation.

- `lintian` issues warnings because the packaged MOOS programs don't include manpages. Creating these manpages would be useful to users, and would reduce the number of `lintian` warnings we see. Including manpages would also add an air of robustness and polish to MOOS that may help persuade people it's worth investing time in learning to use.
- MOOS uses a customized CMake module called MOOSFindFLTK. This package, as well as the standard FindFLTK that's disbributed with CMake 2.4, seem to have some problems that make it unnecessarily difficult to build MOOS but could be addressed:
  - At least on Ubuntu 7.04, this package fails to find FLTK even when it's installed. I've gotten around this by setting the `FLTK_INCLUDE_DIR` CMake variable to have a value of `/usr/include/`. But when I don't do this or something else to fix the problem, MOOS builds with only the meekest of warnings that it's going to skip building those modules that require FLTK.
  - After invoking this module, MOOS's build system looks at the `FOUND_FLTK` CMake variable to decide whether or not the FLTK-dependent parts of MOOS can be built. But this is overly conservative.

    `FOUND_FLTK` will be set to `1` if, in addition to the libraries and headers being found, the program `fluid` was also found. As far as I can tell, MOOS doesn't use `fluid`. But `fluid` appears in its own Debian package; if a user hasn't installed the (unnecessary) `fluid` Debian package, MOOS will unnecessarly skip building its FLTK-dependent parts.

    This is a problem because to the typical programmer trying to build MOOS, it's not at all clear that the solution to the problem of FLTK-dependent parts not buiding is to install a program named `fluid`.

6 Suggested Medium-term Future Work

- Switch this whole mess over to using CPack 2.6 when it comes out.
- Either create an `APT` repository to host MOOS's packages, or get them added to Debian's `universe` repository group. This would make it *trivially* easy to introduce MOOS to Debian / Ubuntu / etc. users.