ARMY RESEARCH LABORATORY

The BRL-CAD CMake Build System - An Overview

by Clifford W. Yapp

ARL-TR-6475 **June 2013**

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Army Research Laboratory

Aberdeen Proving Ground, MD 21005-5066

ARL-TR-6475 **June 2013**

The BRL-CAD CMake Build System - An Overview

Clifford W. Yapp

Survivability/Lethality Analysis Directorate, ARL

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
June 2013	Final	03/2010–04/2013

4. TITLE AND SUBTITLE
The BRL-CAD CMake Build System - An Overview

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Clifford W. Yapp

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Research Laboratory
ATTN: RDRL-SLB-S
Aberdeen Proving Ground, MD 21005-5068

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-6475

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
BRL-CAD, the Army's open source computer-aided design (CAD) system, must build and run successfully on a wide variety of operating systems and build tools to meet the requirements of its user community. The ongoing burden of simultaneously maintaining build system definitions for both the GNU Autotools and Visual Studio build tools motivated a search for approaches offering better support for BRL-CAD's standard build environments. A new cross-platform build system for BRL-CAD was successfully created using Kitware's CMake build tool. As of 2013, the new CMake-based system successfully handles all build tasks on all of BRL-CAD's primary target deployment platforms. This report serves as an introductory guide for software developers wishing to use the new build system to configure and build BRL-CAD.

15. SUBJECT TERMS
CMake, Distcheck, verification, validation, source code, archives, software

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Clifford Yapp
Unclassified	Unclassified	Unclassified	UU	40	19b. TELEPHONE NUMBER (*Include area code*) 410-278-1382

Contents

List of Figures

List of Tables

1. Introduction

The BRL-CAD computer software package needs to be successfully developed and deployed on a broad array of computer operating systems in order to meet the needs of its user community. Managing this process is a complex task, requiring complex instructions that must be continually updated and tested to ensure they can successfully produce a robust software product. Previously, BRL-CAD did not have a single method for encoding and executing the necessary steps in this process that worked reliably and robustly across all of its target platforms — this lack often resulted in duplication of effort in multiple instruction sets and tended to result in out-of-date instructions on less-used platforms. This problem required the development of a new system that entirely replaces both of the previous logic definitions with a cross-platform, robust, and comprehensive set of compilation instructions.

2. Background

The development of software on modern computer systems requires the translation of human-editable input into *output*. Output is generally either machine code that can be executed on an operating system or input to be processed by other software. This translation process is referred to as *compiling* — for example, source code written using the C (*1*) and C++ (*2*) languages are provided as input to compilers such as the GNU Compiler Collection (*3*) and Microsoft's Visual C++® (*4*) (MSVC), which, in turn, produce executable programs an end user can run. The operation of compiling is not limited to producing software executables — other examples include producing portable document format (*5*) (PDF) output from a human-editable input file, generating C/C++ source code from a language grammar definition using RE2C (*6*) and LEMON (*7*), and using Graphviz (*8*) to generate a portable network graphics (*9*) (PNG) image of a graph defined using the Graphviz DOT textual input format.

In virtually any real-world software project, building the entire project requires hundreds or thousands of individual compilation tasks as well as many additional *non*-compilation tasks such as moving files to their correct positions in the final installation directory structure. Each of these *build steps* are defined by specifying inputs, outputs, and options to be used by the particular tool performing a given task. To make this process practical for developers, *build automation* is essential. Automating those portions of the process that are consistent between multiple builds

allows developers to focus their attention on parts of the process that *do* change (individual source files being edited, changing options to particular build tools, etc.). To make this possible, *build tools* have been developed to assist developers with automating their build process. A build tool is one or a set of software programs that does at least one of the following:

1. Accept some definition of a software project and uses that description to invoke individual programs that generate the project's final outputs.
2. Accept high level definitions of software projects and uses those high level definitions to generate lower level definitions.

The definitions specific to an individual project — source file lists, options, settings, macro logic customizing build tool functionality, etc. — constitute the *build system* for that project. For individual software projects, most of the work done to achieve build automation involves defining a build system for a particular build tool.

BRL-CAD (*10*) is an open source computer-aided design (CAD) software project originally developed by the Ballistic Research Laboratory (now the U.S. Army Research Laboratory.) BRL-CAD has been continually developed since the mid 1980s and is descended from earlier software projects dating as far back as the late 1960s. The BRL-CAD project has been complex enough to warrant the use of a build system for virtually its entire history, and in its current state (over 1 million lines of source code and hundreds of executable programs), a properly defined and maintained build system is essential for productive software development.

The earliest builds of BRL-CAD were accomplished using basic Makefile-based build system definitions and the Unix build tool *make* (*11*). In 1988, Mike Muuss[1] introduced a new build system based on Zoltan Somogyi's recently released *cake* program (*12*) to take advantage of features not available in *make*. This build system served as BRL-CAD's primary means of build automation until 2003, when Erik Greenwald[2] began implementing a GNU Autotools (*13*) based build system.[3] While the Autotools build system reached maturity in 2004, portability and robustness requirements prompted significant enhancements to the original Autotools logic from Christopher Sean Morrison[4] over the next several years — particularly once BRL-CAD's

[1] BRL-CAD founding developer (Ballistic Research Laboratory).

[2] BRL-CAD core developer (U.S. Army Research Laboratory).

[3] Autotools consists of the individual build tools GNU Autoconf, GNU Automake, and GNU Libtool. These programs and the Autotools suite they define would be examples of the second type of build tool defined above — tools that accept higher level input files and generate lower level Makefile input for the make build tool.

[4] BRL-CAD core developer (Quantum Research International Inc.) and BRL-CAD open source project leader.

emergence as an open source project dramatically broadened the set of targeted operating system environments.

Software portability is vitally important to preserving a software package's utility over the long term. Software that can no longer be compiled or maintained is frozen, virtually unrepairable and unable to adapt to an ever-changing landscape of operating systems and compilation environments. (For example, two of BRL-CAD's currently supported operating systems — Mac OS X and Linux — did not exist when BRL-CAD development first started.) Although BRL-CAD has proven quite portable over the years, most of the operating system environments in which it has been compiled and run share a design philosophy similar to that of the original Unix operating systems. The Microsoft Windows® operating environment represents a significant departure from those conventions, presenting a challenge for BRL-CAD's portability efforts. A variety of developers over a period of years successfully ported and built the core parts of BRL-CAD with MSVC. Doing so required not only source code changes but also a completely independent set of build system files for MSVC. During the mid and late 2000s, most of the maintenance and enhancements to these files resulted from work by Robert Parker[5]. Inability to automatically generate MSVC project definition files proved to be a key limitation of BRL-CAD's Autotools-based build system.

Hunt and Thomas (*14*) codified "Don't Repeat Yourself" (DRY) as a principle for developers writing software — "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system." This seemingly simple idea is very important to long-term software maintainability, because it reduces the amount of work needed to change any single feature of the system. A large software system that does not strive to respect the DRY principle will find changes progressively harder to introduce both in terms of up-front effort required and increased risk of accidental breakage — exactly the problems BRL-CAD encountered using multiple build systems to support multiple build tools. Because changes to the source file lists in the Autotools input files were not automatically mapped to the corresponding MSVC project files, most attempts at building BRL-CAD on Windows required manually repairing[6] the MSVC project files before other work could proceed.

Issues with the Windows port of BRL-CAD also extended beyond the maintenance overhead of multiple input file sets. Due in part to the overhead of developing custom build logic for MSVC, only the key programs needed for active users were added to BRL-CAD's MSVC project

[5] BRL-CAD core developer (U.S. Army Research Laboratory).

[6] In addition to being frequently out of sync with the project source repository, the MSVC project files themselves tended to accumulate file paths specific to the most active MSVC developer (breaking the project for other developers and further discouraging them from working on the Windows port.)

definitions. Consequently, the Windows port of BRL-CAD offered no guarantee that features users were accustomed to on other operating systems would be available on Windows. Over and above the DRY failure of maintaining multiple build systems, the lack of a "configuration" step in the MSVC build process reduced flexibility by requiring that some values in headers be "hard-coded" instead of setting them based on the current state of the source code and the underlying system. Finally, most BRL-CAD developers other than Mr. Parker had little knowledge of the custom steps needed to produce a Windows release of BRL-CAD using the original MSVC build system. The "bus factor" [7] of the BRL-CAD development team's MSVC knowledge was far too low for comfort.

This report documents the implementation of BRL-CAD's third-generation build system (developed by Clifford Yapp[8]), which is based on the CMake (*15*) build tool developed by Kitware Inc. CMake is a *metabuild* tool — it solves the problem of multiple incompatible build system formats by accepting a relatively abstract build system description and generating the appropriate build system files for individual tools. Unlike GNU Autotools (which also accepts higher level inputs but generates only Makefile-based build systems), CMake can create Makefiles, MSVC project files, Xcode (*16*) projects, and a variety of other build system input formats from a single common definition. It is still necessary to specify appropriate compiler flags for various platforms within the CMake input files, but inputs common to multiple build systems (source code lists, include directories, etc.) are maintained in a single location and become the "single, authoritative representation" recommended by the DRY principle. When a new source file is added to a library, it is no longer necessary to add it to both the MSVC project and the Autotools Makefile.am. CMake instead generates the appropriate MSVC project and Makefiles during the configuration step.

Although automated MSVC project generation is the most compelling advantage CMake has over GNU Autotools from BRL-CAD's perspective, there are also other advantages:

- CMake requires mastery of only a single language to maintain the input CMakeLists.txt files, as opposed to learning the shell script, m4, and Automake/Autoconf syntaxes needed for understanding non-trivial Autotools projects.
- CMake is a single, easily installed, and self contained tool. On Windows, for example, all

[7] A measure of how many developers would need to become unavailable for the team to lose the knowledge and system familiarity needed to work effectively on a given task. Unavailability could be due injury (such as getting hit by a bus), job change, reassignment, etc. – the key point is that however it happened, the resource is no longer available to the team.

[8] BRL-CAD core developer and model manager (U.S. Army Research Laboratory).

that is necessary is to run Kitware's CMake installer — no additional packages are needed. It also serves as a consistent and portable replacement for a variety of system tools.

- CMake provides a "configuration" step for ALL platforms, including Windows — this allows for a variety of build system features when using MSVC that would otherwise be much more difficult to achieve.
- File installation locations in CMake do not change depending on what version of CMake is used — inconsistent installation locations were observed in production use between different versions of Autotools.
- CMake has proven to be flexible when it comes to adding third-party CMake build logic as sub-builds of a parent project — this was often a challenge with Autotools, particularly with Autoconf based build inputs that did not use Libtool.

CMake provides a number of resources in addition to the primary CMake documentation (*15*). While it is not within the scope of this report to teach the basics of CMake, the following online resources are worth noting (links current as of April 2013):

- Wiki: http://www.cmake.org/Wiki/CMake
- Useful variables list: http://www.cmake.org/Wiki/CMake_Useful_Variables
- Compatibility matrix: http://www.cmake.org/Wiki/CMake_Version_Compatibility_Matrix

3. Build Process Overview

When breaking down the BRL-CAD build process into components, the various activities can be characterized as manually prepared inputs (orange nodes), operations manually triggered by the user (red nodes), processing steps (blue nodes), and automatically produced inputs/outputs (green nodes) – these steps and their relationships are illustrated in figure 1. Examining the process breakdown within that framework, it becomes clear that from the user's point of view there are two broad stages that make up the BRL-CAD build process – *configuration* performed by working with CMake and *compilation* performed by working with build tools. All other aspects of the process are either inputs produced by developers (build system and source code) or the products of software programs executed by the build tools.

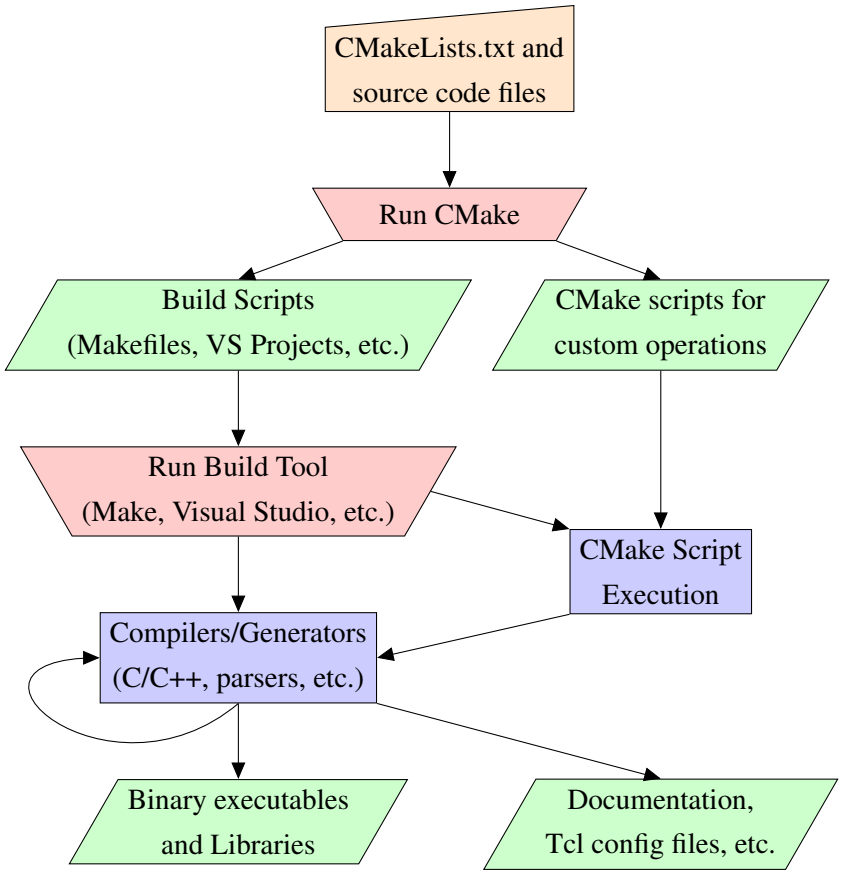

Figure 1. High level flowchart of BRL-CAD's build process.

3.1 Configuration with CMake

When using GNU Autotools, configuration is accomplished with the *configure* script. In a CMake build configuration, the equivalent configuration step is handled by the *cmake* executable.[9] Compilation is accomplished by running the chosen build tool (Make, MSVC, etc.) after the configuration stage is complete. Compilation initiates and manages the multitude of individual steps that generate the binary executables and libraries, as well as building any documentation or other custom outputs. This step works similarly in both Autotools and CMake builds, with the main differences being the formatting of CMake's build output and CMake's support for a broader array of build tools. CMake-based Make builds of BRL-CAD will also make somewhat more efficient use of multi-CPU machines, due to the way CMake manages build target dependencies. Table 1 shows equivalent commands for BRL-CAD's Autotools and CMake builds, enabling all local copies of libraries.[10]

[9] Alternately, one can use the the graphical program *cmake-gui* or curses-based terminal program *ccmake*. For new users, *cmake-gui* is recommended.

[10] The autogen.sh command is run when a source repository is first checked out from version control. After that, it only needs to be re-run when the high level Autotools build inputs change.

Table 1. Comparison between GNU Autotools and CMake configuration commands.

GNU Autotools	CMake
`~/source$ ./autogen.sh`	N/A
`~/build$ ./configure -enable-all ../source`	`~/build$ cmake -DENABLE_ALL=ON ../source`

For users who wish to see the major options offered by BRL-CAD's configuration process, the *cmake-gui* tool provides an excellent alternative to command line tools. Figure 2 illustrates the first two stages of using *cmake-gui* to build BRL-CAD. First, it is necessary to specify the directory containing the source files and the directory intended to hold the compilation outputs.[11] Once those directories are specified, CMake needs to know what build tool the user intends to use for the actual compilation stage. CMake supports a wide variety of build tools. As of 2013, BRL-CAD has been successfully compiled using the Make, Eclipse (*17*), Ninja (*18*), Xcode, and MSVC build generators.

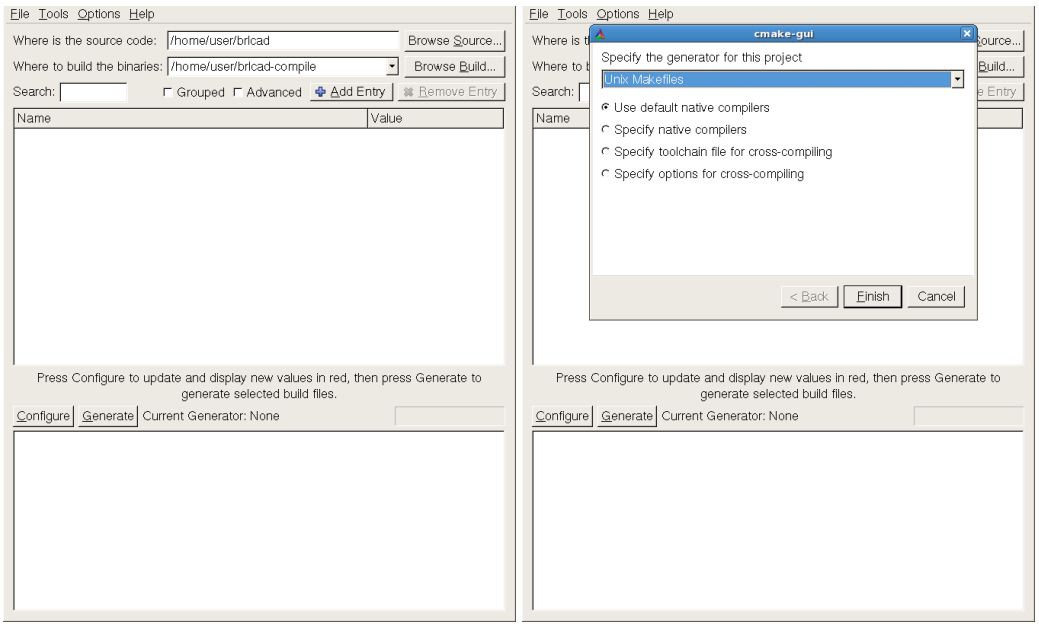

Figure 2. (left) Initial cmake-gui screen and (right) dialog allowing the user to select the build tool for which inputs will be generated.

Once the generator has been selected, run the configure process by pressing the *Configure* button on the left side of the interface. The initial configuration pass typically takes around 1 min, and once it is complete, the top panel will list the available options for customizing the BRL-CAD build (figure 3, left image). After an initial pass, all variables will have red backgrounds, indicating they are new to the variable list after the last configuration run. The user then surveys the new variables and make changes as appropriate.

[11] While BRL-CAD's build system has a custom *distclean* build target allowing users to clear build outputs from a source directory, this is not a standard CMake feature and the best practice is to use a separate build directory.

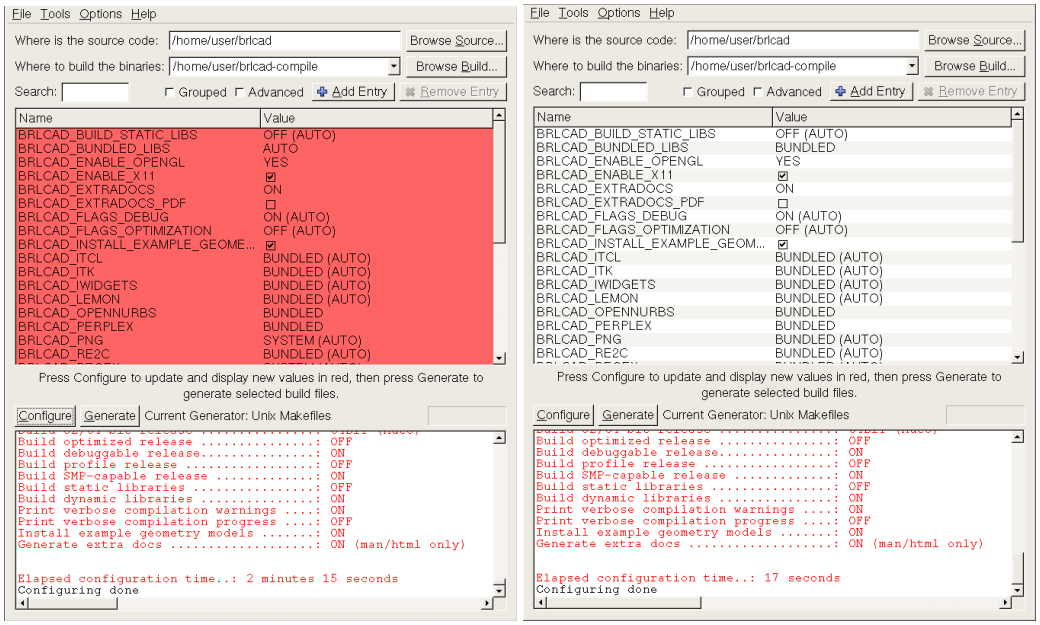

Figure 3. The CMake variable list after the first configure pass (left) and second configure pass (right). After two passes, no new variables are present — ready to generate.

If default values are allowed to remain for all variables, a second run of *Configure* using the configure button will clear all of the red backgrounds. Once this process is complete, CMake is ready to *generate* build tool inputs — this process is triggered with the *Generate* button to the right of the *Configure* button.

For Windows users, note that the CMake build process adds a configuration step and generates an MSVC project file "on the fly," effectively treating the project file the same way Makefiles are treated when building with Make. This is a significant departure from the standard approach of building and maintaining a project definition in MSVC — *all* build information must live in the CMake files. Different versions of Visual Studio require different CMake generators (see figure 4), so the user must be careful to select the generator corresponding to their specific version of Visual Studio.

3.2 Compilation

The compilation process is going to vary substantially, depending on which build tool is used to guide the process. For the purposes of this report, we illustrate the two most common tools used to build BRL-CAD: Make and MSVC.

3.3 Building with Make

Using *make* with a CMake build is relatively straightforward. Table 2 lists some common uses of the *make* command when building BRL-CAD.

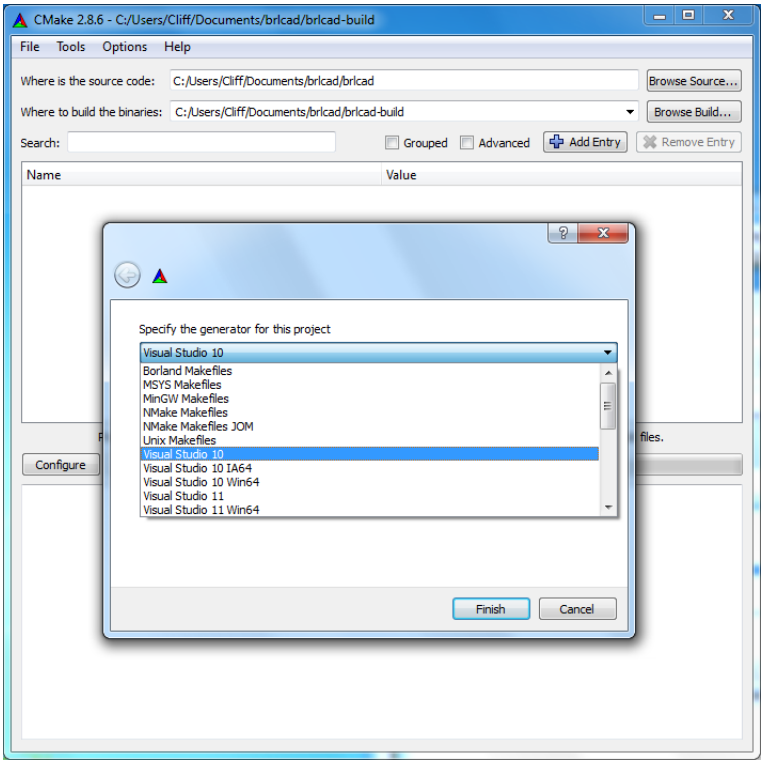

Figure 4. *cmake-gui* showing MSVC generator options.

Table 2. Common *make* commands used when building BRL-CAD.

Command	Purpose
`~/build$ make`	Perform a single threaded compilation of all targets.
`~/build$ make -j8`	Perform a multithreaded compilation of all targets.
`~/build$ make install`	Build and install all targets.
`~/build$ make -j8 install`	Build and install all targets, multithreaded.

The most significant differences in this part of the process as compared to GNU Autotools building are the order in which files are compiled — CMake's make files are somewhat better at parallelism, resulting in both faster compilation and different compilation ordering — and the formatting of the build reporting. CMake-generated Makefiles will produce color output if the terminal environment supports colors.

Figure 5 represents a typical sample of the standard compilation progress report, where details are suppressed unless there are warnings or errors generated by the compiler. To see full details, verbose output is available by compiling with the *VERBOSE* flag set: *make VERBOSE=1*. An example of the verbose output style for the same build targets can be seen in figure 6.

When building specific subsets of BRL-CAD, there are some minor differences from the GNU Autotools

```
[ 2%] Building C object src/other/libtermlib/CMakeFiles/termlib.dir/tgoto.c.o
[ 2%] Building C object src/other/libtermlib/CMakeFiles/termlib.dir/tputs.c.o
Linking C shared library ../../../lib/libtermlib.so
[ 2%] Built target termlib
```

Figure 5. Example of typical CMake build progress log.

```
[ 2%] Building C object src/other/libtermlib/CMakeFiles/termlib.dir/tgoto.c.o
cd /home/user/brlcad/build/src/other/libtermlib && /usr/bin/gcc -Dtermlib_EXPORTS -DCM_N
-DCM_GT -DCM_B -DCM_D -DB_TERMCAP=\"/usr/brlcad/dev-7.21.0/share/brlcad/7.21.0/etc/termcap\"
-pipe -fno-strict-aliasing -fno-common -fexceptions -msse3 -std=gnu99 -m64 -ggdb3 -fPIC
-I/home/user/brlcad/build/include -I/home/user/brlcad/include -w
-o CMakeFiles/termlib.dir/tgoto.c.o -c /home/user/brlcad/src/other/libtermlib/tgoto.c
/usr/bin/cmake -E cmake_progress_report /home/user/brlcad/build/CMakeFiles
[ 2%] Building C object src/other/libtermlib/CMakeFiles/termlib.dir/tputs.c.o
 cd /home/user/brlcad/build/src/other/libtermlib && /usr/bin/gcc -Dtermlib_EXPORTS -DCM_N
-DCM_GT -DCM_B -DCM_D -DB_TERMCAP=\"/usr/brlcad/dev-7.21.0/share/brlcad/7.21.0/etc/termcap\"
-pipe -fno-strict-aliasing -fno-common -fexceptions -msse3 -std=gnu99 -m64 -ggdb3 -fPIC
-I/home/user/brlcad/build/include -I/home/user/brlcad/include -w
-o CMakeFiles/termlib.dir/tputs.c.o -c /home/user/brlcad/src/other/libtermlib/tputs.c
Linking C shared library ../../../lib/libtermlib.so
cd /home/user/brlcad/build/src/other/libtermlib && /usr/bin/cmake -E cmake_link_script
CMakeFiles/termlib.dir/link.txt -verbose=1
/usr/bin/gcc -fPIC -pipe -fno-strict-aliasing -fno-common -fexceptions -msse3 -std=gnu99 -m64
-ggdb3 -m64 -ggdb3 -shared -Wl,-soname,libtermlib.so -o ../../../lib/libtermlib.so
CMakeFiles/termlib.dir/termcap.c.o CMakeFiles/termlib.dir/tgoto.c.o
CMakeFiles/termlib.dir/tputs.c.o -Wl,-rpath,::::::::::::::::::::::::::::::::::::::::
make[2]:  Leaving directory '/home/user/brlcad/build'
/usr/bin/cmake -E cmake_progress_report /home/user/brlcad/build/CMakeFiles
[ 2%] Built target termlib
```

Figure 6. Example of verbose CMake build reporting of figure 5's compilation.

approach. Specific target scenarios are illustrated in table 3. Only the single threaded versions (processes that use ony a single CPU core) are shown. All of these scenarios will also accept the *-j* flag for parallel building, although parallel building will not speed up compilation when the build input is a single C file.

Table 3. Focused *make* commands used when building individual components of BRL-CAD.

Command	Purpose
`~/build$ make librt`	Build librt and its dependencies.
`~/build$ make librt/fast`	Build *just* librt, without ensuring dependencies are current.
`~/build$ cd src/librt && make prep.c.o`	Compile just the file prep.c from librt.

In addition to the standard Make rule for removing build outputs, BRL-CAD provides a *distclean* rule to remove *all* generated files, along the same lines as the Autotools *distclean* rule. Both variations are

documented in table 4. When run in an external build directory, the *distclean* build target should remove every file and leave an empty directory unless there are files present in the build directory not placed there by compilation targets or CMake. When run in the *source* directory, *distclean* can be used to scrub out files added by CMake if a configuration process was run there.

Table 4. *make* commands for clearing build output.

Command	Purpose
`~/build$ make clean`	Remove all compiled files, but keep CMake generated files.
`~/build$ make distclean`	Remove *all* files generated, whether by CMake or by compilation processes.

Other points for *make* users to be aware of include the following:

- When performing a "make install" from a sub-directory in the build tree (instead of the top level build directory) CMake will not (as of version 2.8.4) perform the install step for all dependencies of the sub-directories targets — it will install *only* those in the *current* directory.
- To override compilation flags on an individual, per-target basis for debugging purposes one must edit the contents of file `CMakefiles/<targetname>.dir/flags.make` where `<targetname>` is the specific target for which you wish to customize the flags. Note that changes to the flags.make file are *not* propagated back to the CMake source files — they are a purely temporary override. Editing flags.make files is *never* a substitute for fixing problems with the parent build files, but it is a way to rapidly experiment with different compilation flags without requiring CMake to be continually re-run.

3.4 Building with MSVC

As seen in figure 4, CMake supports generating Visual Studio project files for building on Windows platforms. The configure/generate process is the same with CMake, and the output at the end is a project file in the top level of the build directory (figure 7)[12].

To build, the `BRLCAD` project file is loaded into Visual Studio (figure 8). As of 2013, a BRL-CAD generated MSVC project will need to load over one thousand solution projects.

Once loading is complete, the Solution Explorer presents a list of all available build targets. To simply build *all* of BRL-CAD (the equivalent of *make all* when using Make), right-click on the `ALL_BUILD` target and select "Build" from the pop-up menu (figure 9.) Note that the standard "build all" capability of

[12] Screen shots of Microsoft products are used with permission from Microsoft — see http://www.microsoft.com/en-us/legal/intellectualproperty/Permissions/default.aspx.

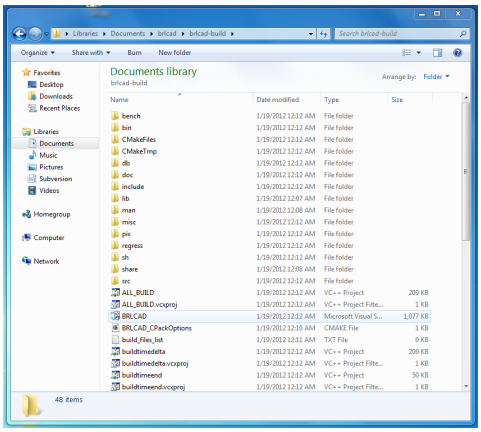

Figure 7. MSVC project generated by CMake.

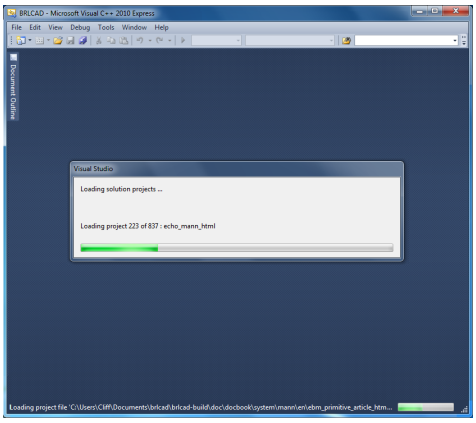

Figure 8. Loading BRL-CAD projects.

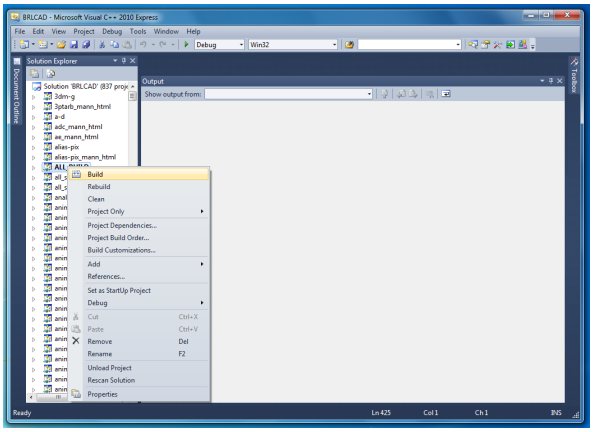

Figure 9. Build all of BRL-CAD.

MSVC (usually bound to the F7 key) is *not* a viable approach to building BRL-CAD with CMake and MSVC. MSVC's "build all" command triggers the custom CMake targets created for tasks such as distribution checking and build directory cleaning in addition to the more standard compilation targets.

Once the process is complete there should be a log in the Visual Studio compilation window summarizing the compilation time, which is typically 30 to 40 min with MSVC on Windows. Below that a report of how many builds succeeded and how many failed. Figure 10 is an example of a report from a successful build. If all builds succeeded, BRL-CAD is now runnable from the build directory.

Typically on Windows, an executable installer is generated and used for installation. BRL-CAD uses CMake's support for the Nullsoft Scriptable Install System (*19*) (NSIS) to generate a self-contained installer. If NSIS is installed, the `PACKAGE` target (figure 11) will build an installer and place it in the top level build directory.

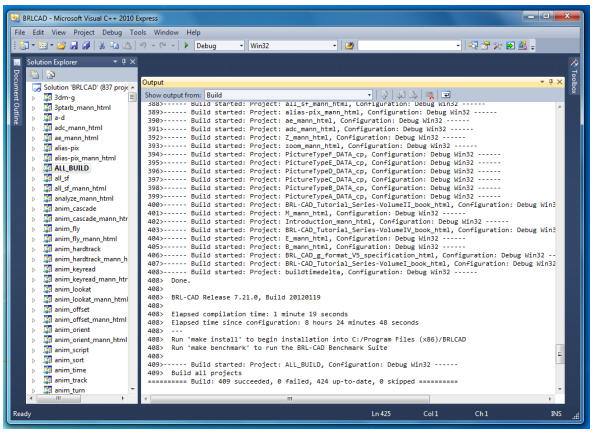

Figure 10. A successful build of BRL-CAD.

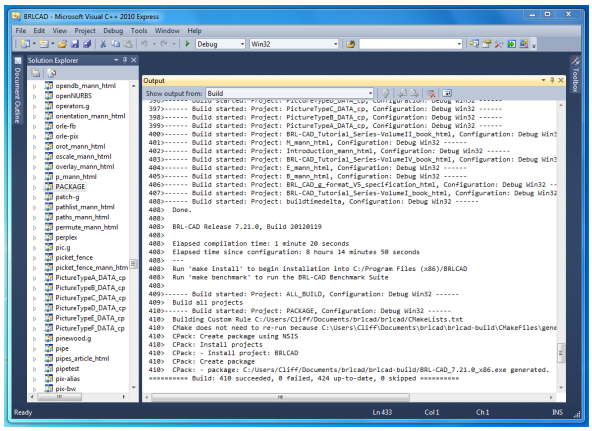

Figure 11. Successful build of the PACKAGE target.

The final step is to verify that the installer works correctly (figure 12). The BRL-CAD NSIS template is

customized with BRL-CAD specific background images and logic to optionally install desktop launchers and menu entries for the major BRL-CAD graphical user interface (GUI) programs.

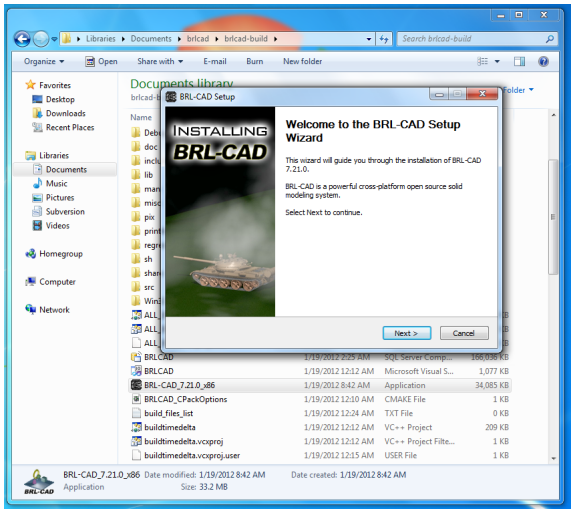

Figure 12. Running the BRL-CAD installer.

Note: although there is an "INSTALL" target that can be run from within MSVC, it is not well tested with BRL-CAD and not recommended for use in development. Developers can run compiled programs directly from the build directories without needing an "install" step. To verify that everything works in the final directory layout, the standard approach is to generate the NSIS installer and verify that the installer's logic does indeed produce a working BRL-CAD.

4. Build System Features — Compilation Options

Now that the basic work flow has been introduced for configuration and compilation, questions naturally arise about what specific features BRL-CAD's CMake build provides, and what is their impact? At a high level, those features might be broken down into *compilation options*, which control what a build produces, and *development features*, which are important capabilities that save developer time while improving the quality of BRL-CAD's distributions. BRL-CAD's high level compilation options include the following:

- Build configurations
- Management of compilation word size
- Third party library and tool management
- Documentation compilation

4.1 Build Configurations

The concept of build configurations, broadly speaking, denotes *sets* of build option settings that are commonly used during software development. While the concept of pre-defined configurations is quite common, how such configurations are defined and used varies considerably across different build tools. GNU Autotools projects that use overall pre-packaged configurations tend to select the working configuration during the *configuration* stage. When using generators for tools such as *make* and *ninja*, the CMake configuration process assumes responsibility for build configuration settings and generated build files provide very little support for build-time decisions. Any changes require altering build options in CMake and re-running the configuration process. On the other hand, tools like MSVC and XCode allow for switching the build configuration from their interfaces just prior to building. CMake's generators for these tools support using and customizing the pre-defined configurations, but with some limitations. For example, MSVC cannot be switched between 32 and 64 bit compilers without selecting a different CMake generator and repeating the full configuration process. For tools that do not specify build configurations at build time, the CMake variable `CMAKE_BUILD_TYPE` is used to set a build type.

CMake provides several "out of the box" options for build configuration, but there are only two primary build configurations used to build BRL-CAD: a "Debug" configuration intended for use during software development and a "Release" configuration used to prepare source and/or binary distributions. BRL-CAD's build system customizes the definitions of Debug and Release builds while eliminating the other standard CMake build types. Note that BRL-CAD's build process does not *require* a build type be specified explicitly to CMake — doing so is primarily a convenient shorthand for enabling common sets of options. If no explicit build configuration is set and the Make generator is used, most settings will default to the debugging configuration.

Build type compilation options are individually overridable. A build configuration establishes defaults, but setting individual variables to values other than those defaults will work. Table 5 identifies the defaults set by the Debug and Release build types for various key settings. Static library targets must be defined as build targets during the configuration step — there is no good way to enable or disable building them at run time. Consequently, multi-configuration build tools always have static library compilation enabled unless specifically disabled by the user. Debug flags are always on unless specifically disabled, in order to allow for better bug reporting.

4.2 Changing Compilation Word Size

When computer users refer to a "32-bit" or "64-bit" computer, they are using a short-hand label that refers to a number of low-level central processing unit characteristics determining limits on (among other things) the type sizes and addressable memory of the computer hardware. From a software compilation standpoint, the important point to note is that for most operating systems one must select either a 32 or 64-bit target platform when building binaries. For a 32-bit computer, 32-bit executables must be built.

Table 5. Build configuration settings for single and multi-configuration build tools.

Property	Debug		Release	
	single config	**multi-config**	**single config**	**multi-config**
Output Directory	<build_dir>	<build_dir>/Debug	<build_dir>	<build_dir>/Release
Install Directory	<prefix>/dev-#.#	<prefix>/dev-#.#	<prefix>/rel-#.#	<prefix>/rel-#.#
Optimization Flags	OFF	OFF	ON	ON
Debug Flags	ON	ON	ON	ON
Static Libraries	OFF	ON	ON	ON

CMake allows scripts to determine what type of platform they are on by using the `CMAKE_SIZEOF_VOID_P` variable. Generally, BRL-CAD relies on that variable, but for Visual Studio project files there are both 32-bit and 64-bit versions of CMake's generators. Thus, in that situation the user must be aware of the correct generator to use.

In cases where it is desirable to build BRL-CAD targeted for a 32-bit platform while on a 64-bit platform, this is achieved (given the availability of the correct system library files) by setting the variable `BRLCAD_WORD_SIZE`. Changing this variable will cause CMake to attempt to flush out the cache, update search paths, and start again.

If a user should specify a BRLCAD_WORD_SIZE that is not supported by the compiler, configuration will halt with a fatal error when it cannot find a working compiler flag to support the given word size.

4.3 Management and Use of Local Third Party Components

BRL-CAD has a long history of bundling local copies of required third party libraries. In addition to developer convenience, this practice enables the following:

- Supports configuration control — if a system version of a library is absent or does not work, the bundled version can be used instead.
- Ensures the availability of a consistent testing environment — to eliminate possible sources of error during debugging, bundled components are sometimes used instead of system components.
- Preserves availability of dependencies that may survive longer in BRL-CAD than they do as independent projects.

Even when the external project is active, it is also occasionally necessary for BRL-CAD to make local changes that differ from the upstream sources. Whenever possible, such changes are incorporated back

into the upstream source trees to minimize maintenance overhead, but in a case where the upstream project is not interested in them, BRL-CAD must be able to maintain changes locally.

While libraries are the main type of third party component stored in BRL-CAD's repository, there are also a number of basic utilities used during the compilation process that are not always available on all systems. These executables are compiled at need, but are not usually installed with BRL-CAD.

BRL-CAD's build logic provides multiple layers of control when it comes to third party code. There are many usage scenarios when it comes to included third party sources. Linux distributions tend to frown on using local copies of libraries, and some have strong policies prohibiting it. Individual users, on the other hand, may not be willing or able to install third party packages. In order to allow the maximum degree of flexibility when enabling and disabling individual third party components, BRL-CAD defines custom logic (figure 13) that specifies common rules for all managed third party components. In order of decreasing priority, those rules are as follows:

1. Feature-based disablement. If a setting is defined in a BRL-CAD compilation to disable all features using the Tk graphical toolkit (for example), all third party components having to do with or requiring Tk will be disabled, regardless of other settings.
2. Has a particular third party component's variable `BRLCAD_<COMPONENT>` been specifically set to SYSTEM? If so, regardless of other settings or system detection results, BRL-CAD will not build the component in question. This may result in compilation failures if system versions of the component are not available.
3. Does BRL-CAD's copy of a particular component have local modifications that may impact behavior? If so, and if neither of the previous two conditions are met, enable local compilation regardless of what is present in the compilation environment.
4. Has a particular third party component's variable `BRLCAD_<COMPONENT>` been specifically set to BUNDLED? If so, and none of the other conditions are in effect, enable compilation of the local copy of this component.
5. Is the top level variable `BRLCAD_BUNDLED_LIBS` set to BUNDLED or SYSTEM (instead of AUTO)? If so, and if none of the previous conditions preclude using the `BRLCAD_BUNDLED_LIBS` setting for a given component, use the value from BRLCAD_BUNDLED_LIBS.
6. If no specific behavior is specified, base all BUNDLED/SYSTEM decisions on introspection of the compilation environment as defined in the BRL-CAD compilation environment feature tests.

Tcl/Tk packages have an additional special case in that they are enabled and disabled based on the local Tcl/Tk compilation settings as well as higher-level controls. If the local Tcl build is enabled, there is no

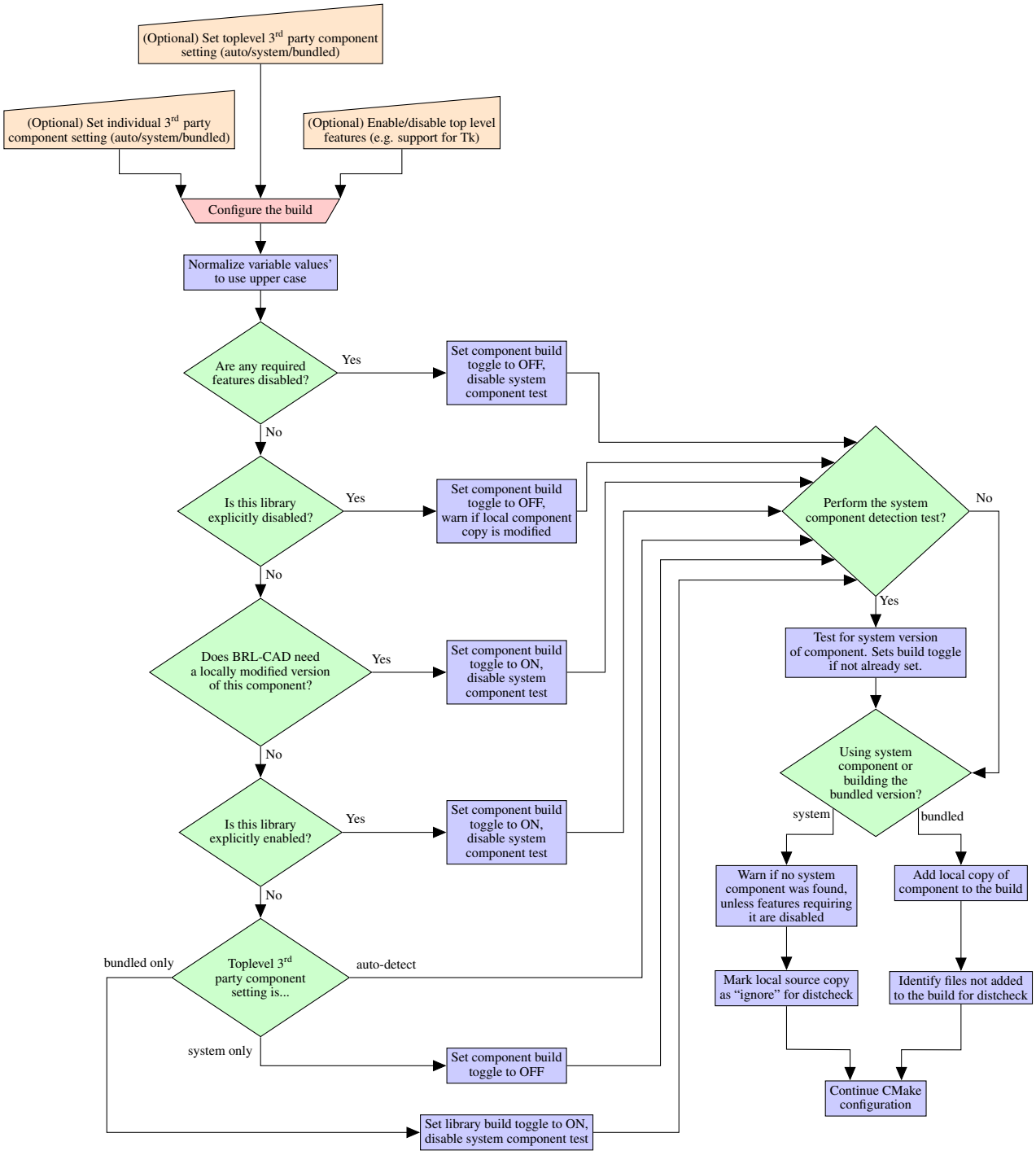

Figure 13. Logical flow of third party component configuration.

point in checking for system versions of the packages — all Tcl/Tk packages are enabled if the local Tcl build is enabled, barring a feature-based override or explicit individual disablement.[13]

[13]Individual Tcl/Tk package disablement is not recommended when building the included Tcl/Tk. Such a setting implies trying to use a Tcl/Tk package from the system with a Tcl/Tk for which it was not built. That configuration

For user convenience, the CMake GUI will display variable values for each third party component that reflect both the setting (AUTO/BUNDLED/ SYSTEM) and — if automatic detection was used or feature-based disablement was applied — the result indicating whether the component will be built or not. Figure 14 illustrates this menu being used to change the build settings for the local BRL-CAD copy of the *libpng* image library.

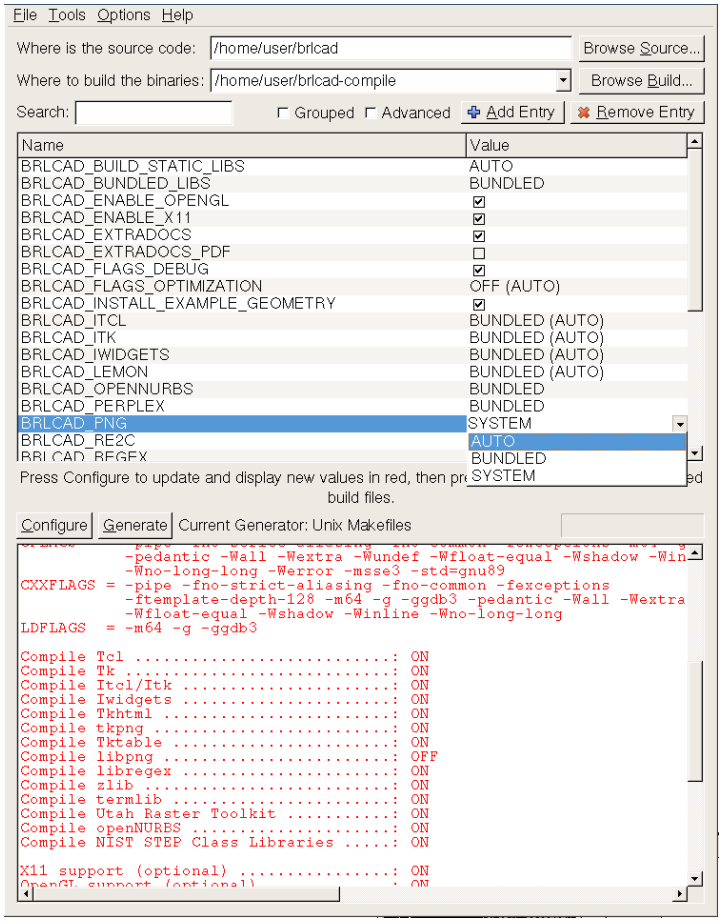

Figure 14. CMake GUI drop-down menu options.

4.4 DocBook-Based Documentation

BRL-CAD needs documentation in a wide variety of formats: Unix man pages for systems supporting the traditional "man" command, HyperText Markup Language (*20*) (HTML) for the World Wide Web and platforms without support for Unix man pages, and PDF for documents needing a well-defined, consistent appearance. This presents a considerable challenge for documentation maintainability. It is difficult enough to keep software documentation up to date without having to update multiple documents using different formats to store the same information. In documentation as in software, the DRY principle remains a valid guide to reducing the work required for development and maintenance. BRL-CAD's

is unsupported and is likely to exhibit subtle (or even not-so-subtle) problems at run time.

solution to this problem is to make use of the DocBook documentation format and tool chain.

Ongoing efforts to convert existing documentation into DocBook format have resulted in the successful conversion of Volumes I, II, and III of the BRL-CAD Tutorial Series as well as several hundred multidevice graphics editor (MGED) and command line manual pages. Contributions from Tom Browder[14] in 2011 added sophisticated formatting of PDF output for the tutorial series volumes (figure 15.)

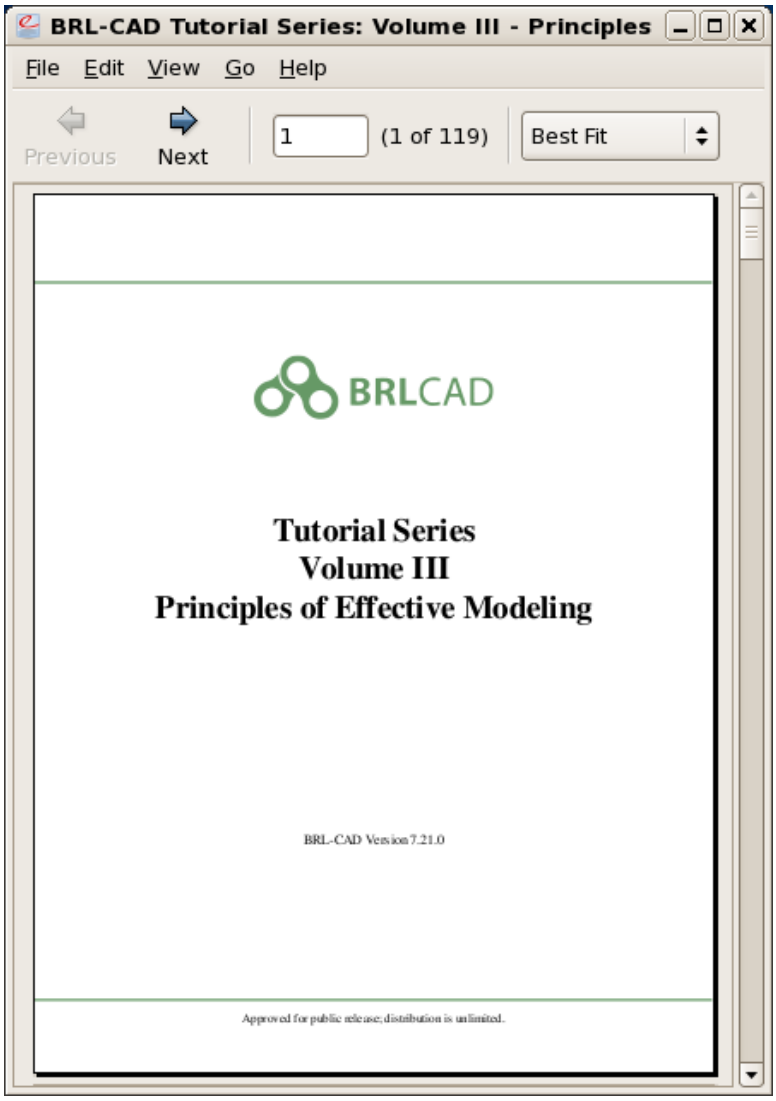

Figure 15. Customized PDF output for DocBook Tutorial III.

The BRL-CAD compilation process automates the conversions needed to produce HTML, Unix man page, and (optionally) PDF output. BRL-CAD does not bundle all the tools necessary to produce PDF output in its own source tree — only HTML and Unix man page outputs are guaranteed to be available from BRL-CAD's own internal resources. Currently, generating PDF output requires a working installation of

[14] BRL-CAD open source project contributor.

Apache Formatted Objects Processor (*21*) (FOP) in the compilation environment. PDF output is disabled by default, even when FOP is present, because PDF generation adds significantly to the total compilation time and PDF documents are not directly used in any of BRL-CAD's standard help systems. If PDF documentation is enabled, a conditional option is activated when the configure step is re-run (figure 16) that will allow the deactivation of *just* the man page PDF outputs, while producing PDF output for other DocBook targets. This option is useful in situations where PDF versions of the tutorials are required but not hundreds of individual PDF man pages. Table 6 lists the available DocBook options.

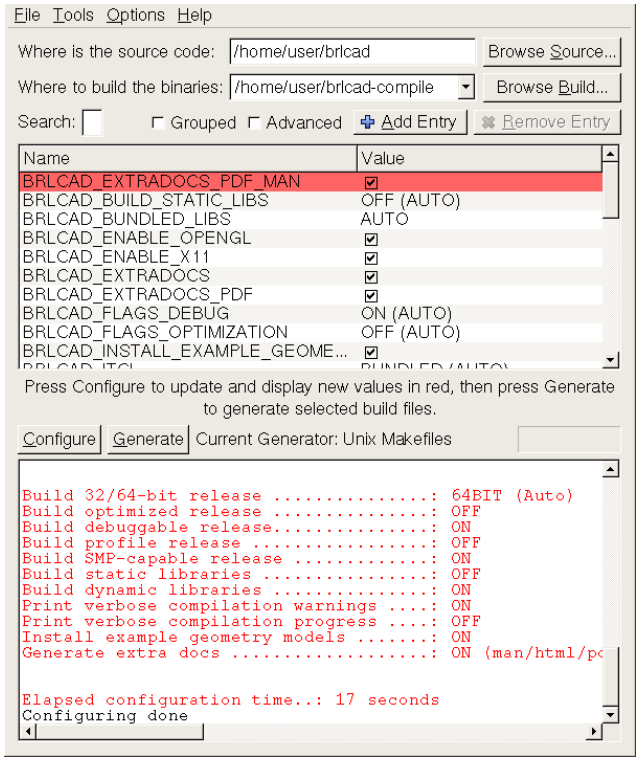

Figure 16. Addition of PDF variable after enabling PDF output.

Table 6. BRL-CAD DocBook build options.

Variable	Description	Default Setting
BRLCAD_EXTRADOCS	Enable DocBook documentation	ON
BRLCAD_EXTRADOCS_HTML	Enable HTML output	ON
BRLCAD_EXTRADOCS_MAN	Enable man page output	ON (OFF with MSVC)
BRLCAD_EXTRADOCS_PDF	Enable PDF output (needs FOP)	OFF
BRLCAD_EXTRADOCS_PDF_MAN	Enable PDF man page output	${BRLCAD_EXTRADOCS_PDF}
BRLCAD_EXTRADOCS_VALIDATE	Validate XML input	${BRLCAD_ENABLE_STRICT}

By default, XML (extensible markup language) Stylesheet Language for Transformations (XSLT) processors (the primary tool used to convert DocBook inputs to other formats) will not require that XML inputs are strictly valid according to the DocBook schema. In order to enforce strict compliance with the DocBook schema as a precondition for successful building (roughly analogous to forcing a C compiler to treat warnings as errors with the -Werror compilation flag), BRL-CAD's CMake logic runs a validation tool on each file the first time it is converted to an output format (validation does not depend on any particular output format being chosen, so a file only needs to be validated once in a build.) By default, successful validation is required for a successful build. This is why the `BRLCAD_EXTRADOCS_VALIDATE` variable is conditionally set according to whether BRL-CAD is performing a build that treats warnings as errors (it takes its default value from the value of `BRLCAD_ENABLE_STRICT`). By default strict compilation is always enabled.

XML validation in a BRL-CAD compilation is typically performed using the tool *xmllint* (*22*), which is built by BRL-CAD as needed. Other validators are not bundled with BRL-CAD, but if a developer prefers another validator and has it installed, there is a way to use it. The Oracle Multi-Schema XML Validator (*23*) can be used by setting the variable `VALIDATE_EXECUTABLE` to "msv", and the Relax NG Compact Syntax Validator (*24*) can be used by setting the same variable to "rnv". Other tools may also be used but will require additional work; see the documentation in the doc/docbook source directory for details.

---

5. Build System Features — Development Features

The work of a build system is not just to present options to users; it must also manage compilation tools to produce output that is correct, functional and convenient. Sometimes this is a matter of setting options correctly, and in other cases, a great deal of custom logic is needed. In either case, the frequency with which common development tasks are performed results in small efficiency gains adding up over the long term.

5.1 Packaging — Preparing Source and Binary Archives

When the BRL-CAD project prepares a versioned release, the canonical definition of that release is a "tag" in the version control repository. Most users will not directly obtain the "tagged" version control repository checkout, but instead obtain the files using a source archive. The creation of source archives is a standard activity performed when any release is made. The overall release procedure is defined in BRL-CAD's top-level HACKING document, which also specifies naming rules for BRL-CAD archive files. Source archives use the following filename template:

```
brlcad-{VERSION}.{EXTENSION}
```

Although they are not generated for all releases, binary archives containing pre-compiled executables and libraries should be generated when a release offers major improvements or new features. The naming convention for binary archives is somewhat more elaborate, due to the broader range of options that must be covered by binary releases:

```
BRL-CAD_{VERSION}[_{OS}][_{VENDOR}][_{CPU}][_{NOTE}].{EXTENSION}
```

Note also that this naming convention is only for binary releases from the BRL-CAD open source project itself — operating system package repositories are more likely to use their own naming conventions.

These templates are automatically respected by CPack, deriving necessary information from BRL-CAD build system settings. The EXTENSION for source archives is dictated by the type of compression used. Table 7 lists the available options.

Table 7. BRL-CAD's standard source archive formats.

Compression Type	Extension
Tape archive (*25*) compressed using the DEFLATE (*26*) compression method	tar.gz
Tape archive (*25*) compressed using the Burrows-Wheeler (*27*) compression method	tar.bz2
ZIP (*28*) archive	zip

Table 8 lists the binary package formats currently supported by BRL-CAD's CMake build system.[15]

Table 8. BRL-CAD's binary package formats.

Package Type	Extension
Windows executable installer via Nullsoft Scriptable Install System (*19*)	exe
*NIX style binary archives	tar.gz, tar.bz2, zip
RPM Package Manager packages (*29*)	rpm

The standard way to create these files is to use the *package_source* and *package* build targets, as demonstrated in abbreviated form by figure 17.

A cautionary note is in order concerning file permissions and binary packages. While CMake does not directly respect umask settings on platforms that use umask during installation, it *does* use umask settings when creating files in the build directory. The permissions on those files are then preserved during the installation process, so it is important (particularly for binary packages such as RPM that will be installed to system locations) that umask settings be set to values that are appropriate for system binaries. In

[15] BRL-CAD can also produce Debian Linux and Mac OSX packages, but those formats are not (yet) handled by the CMake build logic. Other tools and manual effort are needed to produce them.

```
bash-4.0$ make package_source
Run CPack packaging tool for source...
CPack: Create package using TGZ
CPack: Install projects
CPack: - Install directory: /home/user/brlcad
CPack: Create package
CPack: - package: /home/user/brlcad/build/brlcad-7.21.0.tar.gz generated.
bash-4.0$ make package
[check build targets, build any incomplete]
Run CPack packaging tool...
CPack: Create package using RPM
CPack: Install projects
CPack: - Run preinstall target for: BRLCAD
CPack: - Install project: BRLCAD
CPack: Create package
CPackRPM: Will use GENERATED spec file:
/home/user/brlcad/build/_CPack_Packages/Linux/RPM/SPECS/brlcad.spec
CPack: - package: /home/user/brlcad/build/BRL-CAD_7.21.0_Linux_x86_64.rpm generated.
```

Figure 17. Using standard CMake build targets to generate source archives and binary packages.

addition, a CMake bug has resulted in RPM packages that change the permissions of /usr (for example) if the permissions of the RPM archive differ from existing directory permissions. The wrong permissions on such a directory can render the majority of executables on a system inaccessible to most users. CMake will try to warn at configuration time if umask settings are not set properly, but package builders should double check their settings to make sure the results are what they expect.

5.2 Distribution Checking

Source archive and binary package generation is an essential component of preparing a BRL-CAD release, but by itself it is insufficient. How is a developer to know that the archives they created contain the correct files and produce a working BRL-CAD installation? In the software release context performing these introspective verifications is known as *distribution checking*. BRL-CAD's previous Autotools-based build system defined extensive distribution checking logic, building on the standard Autotools *distcheck* feature. The current design of BRL-CAD's CMake distcheck target is based on that work.

Distribution checking for a BRL-CAD release involves three distinct stages:

1. Determine what files are known and unknown to the version control system and the build logic, as well as what is actually present on the file system, and check for inconsistencies.
2. Using CPack, generate source archives as explained in the previous section.
3. Verify that unpacking, configuring, building, installing, and testing the contents of one of the source archive files is successful.

Figure 18 demonstrates the first stage of the process, reporting that a file is present in the source tree but both version control and the build system agree it is not currently part of BRL-CAD and can thus be ignored for subsequent packaging operations. Had distcheck found that (1) the build system knows about the file but it was not checked in to the version control system, or (2) the version control system knew about it but the build system has no record of it, the process would have halted with a fatal error instead of simply printing an informational message.

```
bash-4.0$ make distcheck
Scanning dependencies of target distcheck-repo-verify
*** Check files in Source Repository against files specified in Build Logic ***
 -- Building file list from Subversion manifests:  ...
 -- Building file list from Subversion manifests:  Done
 -- Building list of files known to the build system:  ...
 -- Building list of files known to the build system:  Done
 -- Building 'ground truth' list of files actually present in source tree:  ...
 -- Building 'ground truth' list of files actually present in source tree:  Done
 -- Performing comparisons...

Files unknown to both Subversion and the Build logic (will not be incorporated
into dist):
src/librt/tree2.c

Source Repository Verification:  Passed
```

Figure 18. Source repository state verification.

Once the repository verification is successfully completed and source archives are created, the most difficult part of the process — full testing of the archive sources — begins. BRL-CAD's build system adds a new feature to this stage of distribution checking, allowing the build system to define build configurations that will be part of the testing process. These configurations are more detailed than the Debug and Release configurations that constitute the standard testing configurations. The basic *distcheck* build target will not trigger all of these tests, since any significant number of configurations will result in a rather arduous and prolonged distribution check, but for situations where the resources are available or the testing coverage is essential, the *distcheck-full* target will build *all* defined configurations.[16] Figure 19 shows an example distcheck-full with three build configurations being tested simultaneously, using the *tail* and GNU *screen* programs to simultaneously view multiple log files.

[16]Because of the way this part of the test logic is written, a parallel *distcheck* will launch multiple parallel compilations of BRL-CAD simultaneously. A highly parallel *distcheck-full* build may prove to be a significant stress test for the host computer and operating system.

Figure 19. Viewing the progress of a *distcheck-full* build using GNU screen.

6. Conclusion

Software configuration and compilation is a complex and essential part of any large development project. As of 2013, the new CMake-based system successfully handles all build tasks on all of BRL-CAD's primary target deployment platforms. Advanced features improve the efficiency of developers working with the BRL-CAD source code by providing accessible, intuitive, flexible, and powerful controls over the most important aspects of the BRL-CAD compilation process, while automating large portions of the verification process necessary for release quality assurance. Windows compilation is now far more integrated with the standard BRL-CAD development work flow, and Windows binary releases no longer require custom scripts and extensive manual updating for each release. Given the software tools and operating systems in standard use as of 2013, BRL-CAD's CMake build successfully meets the project's needs.

One of the lessons of the GNU Autotools → CMake conversion effort is that build system *logic* documentation (as opposed to the *syntax* specifics associated with any particular build tool) is a valuable long-term resource. The time may come when CMake is no longer the best tool for the job of building

BRL-CAD. Just as *cake* was replaced by GNU Autotools and Autotools was, in turn, replaced by CMake, future developers may need to replace CMake with a new build tool. Experience with the CMake conversion suggests that clear descriptions of specific tasks and control flows, unencumbered by build tool specific syntax, will help make that process faster and cleaner. For example, the decision flowchart in Figure 13 remains the same regardless of the build tool used to implement it. Distribution checking will still need to complete the same set of tasks handled by the current system. Going forward an effort will be made to document and publish the abstract logic of key BRL-CAD build system features both to clarify intent for current maintainers and preserve knowledge for the future. We recommend other teams managing software with similar complexity and long-term maintenance needs consider whether such documentation would be an asset to their project.

7. References

1. American National Standards Institute, *INCITS/ISO/IEC 9899-1999 (R2005): Programming Languages — C (formerly ANSI/ISO/IEC 9899-1999)*; American National Standards Institute: 1430 Broadway, New York, NY 10018, USA, 2005.

2. ISO, *ISO/IEC 14882:2011 Information technology — Programming languages — C++*; International Organization for Standardization: Geneva, Switzerland, 2012.

3. Free Software Foundation, Inc., GCC, the GNU Compiler Collection. 2013.

4. Microsoft Corporation, Visual Studio. 2010.

5. ISO, *ISO 32000-1:2008 Document management – Portable document format – Part 1: PDF 1.7*; International Organization for Standardization: Geneva, Switzerland, 2008.

6. Bumbulis, P.; Cowan, D. D. RE2C - A More Versatile Scanner Generator. *ACM Lett. Program. Lang. Syst* **1994**, *2*, 70–84.

7. Hipp, R. D. Lemon Parser Generator. 2013.

8. Gansner, E. R.; North, S. C. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE* **2000**, *30* (11), 1203–1233.

9. ISO, *ISO/IEC 15948:2004 Information technology – Computer graphics and image processing – Portable Network Graphics (PNG): Functional specification*; International Organization for Standardization: Geneva, Switzerland, 2004.

10. BRL-CAD Open Source Project, BRL-CAD - Computer Aided Design Software. 2010.

11. Feldman, S. I. Make – a program for maintaining computer programs. *Software: Practice and Experience* **1979**, *9* (4), 255–265.

12. Somogyi, Z. Cake: a Fifth Generation Version of make. 1987.

13. Free Software Foundation, Inc., GNU Autotools. 2013.

14. Hunt, A.; Thomas, D. *The Pragmatic Programmer: from journeyman to master*; Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 1999.

15. Martin, K.; Hoffman, B. *Mastering CMake: A Cross-Platform Build System*; Kitware Inc: Clifton Park, NY, 2003.

16. Apple, Inc., Xcode. 2012.

17. Eclipse Foundation, Eclipse IDE for C/C++ Developers. 2010.

18. Martin, E. Ninja Build System Tool. 2012.

19. Nullsoft, Inc., Nullsoft Scriptable Install System. 2009.

20. ISO, *ISO/IEC 15445:2000 Information technology – Document description and processing languages – HyperText Markup Language (HTML)*; International Organization for Standardization: Geneva, Switzerland, 2000.

21. Apache XML Graphics Project, Apache Formatted Objects Processor. 2013.

22. The Gnome Project, The XML C parser and toolkit of Gnome. 2012.

23. The Oracle Multi-Schema XML Validator (MSV). 2013.

24. RNV Open Source Project, Relax NG Compact Syntax Validator. 2004.

25. Kientzle, T. TAR - Format of Tape ARchive Files. 2009.

26. Deutsch, P. GZIP file format specification version 4.3. RFC 1952 (Informational), 1996.

27. Seward, J. BZIP2 File Compressor. 2010.

28. PKWARE Inc., ZIP File Format Specification. 2012.

29. RPM Open Source Project, RPM Package Manager. 2013.

NO. OF COPIES	ORGANIZATION
1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
1 (PDF)	DIRECTOR US ARMY RESEARCH LAB RDRL CIO LL
1 (PDF)	GOVT PRINTG OFC A MALHOTRA
1 (PDF)	USARL RDRL SLE R FLORES
	ABERDEEN PROVING GROUND
1 (HC)	DIR US ARMY EVALUATION CTR HQ TEAE SV P A THOMPSON 2202 ABERDEEN BLVD 2ND FL APG MD 21005-5001
3 (2HC 1 PDF)	DIR USARL RDRL SL J BEILFUSS P TANENBAUM RDRL SLB A M PERRY (PDF only)

NO. OF COPIES	ORGANIZATION
	ABERDEEN PROVING GROUND
23 (7 HC 16 PDF)	RDRL SL G KUCINSKI (1 PDF) RDRL SLB S M PERRY (1 PDF) G SAUERBORN (1 PDF) C YAPP (1 PDF, 5 HC) E GREENWALD (1 PDF, 2 HC) W BOWMAN (1 PDF) R PARKER (1 PDF) C MORRISON (1 PDF) N REED (1 PDF) R WEISS (1 PDF) L BUTLER (1 PDF) RDRL SLB W S SNEAD (1 PDF) T MYERS (1 PDF) RDRL SLB A G MANNIX (1 PDF) RDRL SLB D R GROTE (1 PDF) RDRL SLB E M MAHAFFEY (1 PDF) RDRL SLB G P MERGLER (1 PDF)
1 (PDF)	NVL SURFC WARFARE CTR DAHLGREN D DICKINSON G24
1 (PDF)	AFRL MUNITIONS DIRCTRT N GAGNON
1 (PDF)	AFRL RWAL S STANDLEY
1 (PDF)	ASC ENDA VULNERABILITY TEAM T STALEY
1 (PDF)	QUANTUM RSRCH INTRNTL C HORTON
1 (PDF)	SURVICE ENGRG D KREGEL
1 (PDF)	APPLIED RSRCH ASSOC A ROSS
1 (PDF)	MANTECH SRS TECHLGY T BROWDER

NO. OF COPIES	ORGANIZATION
2 (PDF)	NO PRINS MAURITS LAB S PRONK W BOKKERS THE NETHERLANDS
1 (PDF)	INDUSTRIEANLAGEN BETRIEBSGESELLSCHAFT MBH IABG D ROSSBERG GERMANY