

Building Software via Shared Knowledge

José R. Herrero, Juan J. Navarro

Computer Architecture Department, Universitat Politècnica de Catalunya*

Jordi Girona 1-3, Mòdul D6, 08034 Barcelona, Spain

{josepr,juanjo}@ac.upc.es

Abstract *In this paper we present a new approach to writing Makefiles and a system called maker which helps in this process. Our main goals are: ease the process of writing user Makefiles, reuse variable and rule definitions, handle common tasks automatically (dependency tracking, preparation of code and environment for testing or debugging) and provide support for software development on heterogeneous environments (automatic creation of targets in a specific build tree for each architecture while working in the source tree; use of appropriate compiler name, flags and libraries; preparation of environment variables for finding libraries and programs).*

Keywords: Build process, Makefile reuse, NFS

1 Introduction

Building large software packages is a complex task. Source code is usually scattered over many files and directories. Creation of destination files out of the source files can be eased with the help of build programs. One such program is make [3] which has become a de facto standard in the Unix world. make uses files called Makefiles to get directions on how targets have to be built [8]. There are many flavours of make [1, 4, 5, 10]. However, one of them is particularly attractive: GNU make or gmake [10]. It is freely distributed, has a largely extended functionality over conventional make versions, and it is available for many different platforms. There exists a tool called pmake which

extends gmake's utility to support distributed job execution [5].

In principle, creation of Makefiles is rather easy. However, it can become tedious work when handling projects with many directories and files. Common targets and variables are often repeated in Makefiles in different directories. For instance, a target called clean is commonly used to delete all files in the current directory that are created by building the program. A target like this will be probably found in as many Makefiles as directories in the project.

Dealing with file dependencies can become rather complicated when included header files include other files themselves. There are ways to get this dependencies from the compiler but most people either do not know this is possible or run it only at the time they create the Makefile. Unless the dependency list is updated dynamically when a change is detected, inconsistencies can occur.

Another difficulty arises when one wants to build the project in a different directory structure from where the sources are, i.e. the build tree is different from the source tree. The problem with this is the time and effort it takes to change to the directory and invoke make with the -f option followed by a possibly long path compared to the time it takes to type make in the current directory.

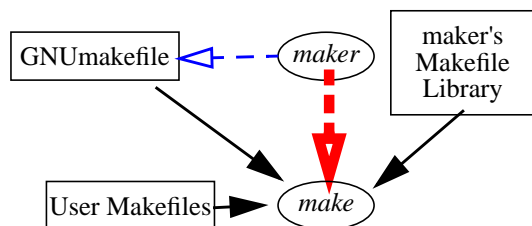
Finally, handling compilation in a heterogeneous environment with different platforms can be a real pain. The compiler or the library names, command line options or the location of files in different systems can differ substantially. Some of these problems have already been tackled by other tools:

*. This work was supported by the Ministerio de Ciencia y Tecnología of Spain and the EU FEDER funds (TIC2001-0995-C02-01)

- BSD make [1] and nmake [4] use Makefile templates which can be included from other Makefiles to allow for reuse of variable and rule definitions.
- *automake* [7] and *autoconf* [6] help in the process of writing portable code and its later distribution and installation. They also tackle the problem of dealing with a directory hierarchy and using different build trees.

In this paper we present a new tool called maker: a front end to make that solves the problems stated above and greatly simplifies the creation of user Makefiles. The main advantages to maker's users are: first, user Makefiles are remarkably simple since they can reuse rules and definitions. Second, the user does not need to deal explicitly with build directories specific to each target architecture. Instead, the user can call maker from the source tree and it will automatically use adequate compiler flags and create the targets in the appropriate build tree. This is particularly interesting in a heterogeneous system of computers using a Network File System [9]. For the time being, however, maker does not address the problem of code portability and distribution.

2 System Architecture.



Legend:

- - - > File creation
- > Input file
- - - > Program call

The system has three components: 1) a program called maker which drives the execution, 2) a set

of predefined Makefiles which we refer to as maker's Makefile library, and 3) an intermediate Makefile, named GNUmakefile, built and used by maker which acts as link between the user Makefiles and the Makefile library

maker

maker is a wrapper around make (gmake). It is a program written in Perl [11] that automatically changes to the appropriate build tree and calls gmake from there. It is responsible for passing gmake the correct parameters for operation. This includes passing the name of the Makefile in the source tree with the correct path. In order to do that, an intermediate Makefile is generated and used. This file is called GNUmakefile and needs to be generated only once, for the root of the project tree (PRJROOT). Other tasks performed by maker are the creation of certain files or directories if they do not exist and are necessary. The GNUmakefile and the build tree directories (with the same tree structure as in the source tree) are always needed. The debug build tree and debugger initialization file are only built when needed.

The GNUmakefile

The GNUmakefile is a Makefile automatically created by maker which allows for an automatic inclusion of the Makefile Library files into the user Makefiles. It defines some variables which are important for the correct operation of the system. These variables are:

- PRJROOT
The absolute name of the project source tree root. This is useful to locate per project Makefiles or header files (if stored in “\$(PRJROOT)/include” since this directory is automatically added to the search path for header files).
- PRJCWD
The absolute name of the of the project current working directory in the source tree. When make is called the current working directory is in the build tree. Since source files come from the source tree, we need a way to specify the matching direc-

tory in the source tree where source files can be found.

There are also 3 variables that gmake uses to control its behavior:

- **MAKEFLAGS**
This variable is automatically passed to a sub-make, i.e. a recursive invocation of make. Amongst data passed are directories where Makefiles can be found. This includes the system wide *maker* directory so that the Makefile Library files are found; directory “\$(PRJROOT)/Makefiles” so that per project Makefiles can be found (this usually applies to file Makefile.prj); and the current working directory in the source tree (PRJCWD).
- **MAKEFILES**
This variable keeps the name of all the Makefiles to be read on every invocation of make. These files are, in load order: Makefile.sys Makefile.prj Makefile.cfg and Makefile.lib. Thus, these files will be automatically loaded upon call to the user's Makefile in each subdirectory.
- **VPATH**
Keeps the search path for all dependencies. It contains values given at the command line, plus the PRJCWD

Once these variables are defined, make is invoked for the user's Makefile in \$(PRJCWD)/Makefile. By the time it is loaded the Makefile library is already loaded and fully available for use as if it was coded in the user file. Only one GNUmakefile is necessary for a whole project's directory hierarchy. This file must correspond to the project root directory and can be used to prepare the build process for any subdirectory under the project root directory. This allows for finding header files in “\$(PRJROOT)/include” or libraries created in some other subdirectory within the project directory hierarchy.

The GNUmakefile is created only once and reused henceforward. It is automatically created by maker if it does not exist but can also be created on demand. If the project is moved to a different directory the GNUmakefile has to be regenerated since the PRJROOT is hard coded in this file.

Makefile library

The Makefile library is a set of predefined Makefiles which contain variable initializations and/or declaration of rules and functions. These files are read before the user Makefile is read so that all their contents are available to the user. They are presented in the same order as they are loaded:

- **Makefile.sys**
This is the system-wide configuration Makefile. In this file variables controlling the build process are set to their default values. Variables could be grouped into several categories: software used, variables used as compiler or linker options, installation directories, default filenames, search paths and platform description. It assumes environment variable *ARCH* is defined and its value describes the architecture where the execution is being performed. This is the only environment variable that *maker* needs predefined. In case it is not predefined the specific platform description will simply not be used. Instead, some variables like compiler names or flags will be set to default values.

At this point, the project file Makefile.prj would be read if it exists. Next, the user file Makefile.cfg would be loaded if the user had provided one for the current directory being processed. Afterwards, Makefile.lib is automatically included:

- **Makefile.lib**
This file defines a set of common targets and rules and is meant to be used as a Makefile library. At this time this file has about 1700 lines which are mainly common targets such as clean or install, or pattern rules like “%.so: %.a” (which defines the way shared libraries can be created from archives) or “%.d: %.c” (for creation of dependency files for C programs). A rule for recursion into directories listed in variable SUBDIRS is also given in this file. Three files are included at the very beginning of Makefile.lib: *Makefile.vpath*, *Pdesc.\$(ARCH)*, and *Makefile.deps*. At the end, some optional parts of the library called *maker modules* can be loaded.
- **Makefile.vpath**
This is the Makefile used for configuring the header file search path. This path includes the

project current working directory in the source tree, the project include directory, a system dependent and a system independent include directories. The user can specify other directories for header file searching in a variable called `USRINC` which is placed before the rest of the directories in the header file search path. The order directories are added to the search path allows for file interposition: a directory or a project can use their own header files overriding the system ones with the same name.

- `Pdesc.$(ARCH)`
This is the platform description. This file defines architecture/system dependent environment variables. Environment variable `ARCH` should be set accordingly before `maker` is used. Otherwise default values are used.
- `Makefile.deps`
This file defines a way to include `.d` dependency files in Makefiles. This inclusion will be automatically done for files listed in variables `CSRC` and `FSRC` (for C and Fortran source files).
- `maker modules`
Optionally, some parts of the library which are only used in some cases can be loaded. We call these parts `maker modules`. They extend the Makefile Library functionality while preserving performance. They are only read on user demand as specified on variable `USE_MAKER_MODULES`.

At this point, user file `Makefile` would be loaded and the build process would start.

User Makefiles

Per project:

- `Makefile.prj`
This is the project-wide configuration Makefile and should be stored in directory `$(PRJROOT)/Makefiles`. Since this file is included for every subdirectory in the project, common definitions can be placed here.

Per directory:

- `Makefile.cfg`
If present, this file is read by `make` immediately

before loading file `Makefile.lib` and can thus be used to customize some behavior of the library. For instance, defining variable `CSRC` to list all C files used in that directory will trigger the automatic inclusion of dependency files which can itself trigger the automatic dependency generation in case dependency files are inexistent or need to be rebuilt.

- `Makefile`
It should specify the targets for the directory as well as the source file dependencies for each target she wants to build and the way it has to be built (unless a rule already defined by `gmake` or our Makefile Library applies).

An equivalent solution could be achieved with a single user file per directory which had a first part with the contents of the `.cfg` file, followed by an explicit inclusion of file `Makefile.lib`, and a final part with the targets in `Makefile`. This solution is possible but has not been used to avoid the explicit inclusion of `Makefile.lib` by the user.

3 Available Operations

For the time being `maker` extends `make` basic functionality in many ways. The following list shows briefly some aspects for which `maker` either does it automatically, or supports it with a minimum effort from the user, like specifying a simple option from the command line.

- change current working directory to the appropriate directory in the build tree before calling `gmake`
- generation of the project GNUmakefile
- inclusion of makefiles: system, project, configuration, library and platform description
 - initialization of variables with default values
 - definition of common targets
 - definition of new rules
- dynamic dependence generation
 - invoke `cpp` to create `.d` files listing all dependencies with header files for each C or Fortran files listed in variables `CSRC` and `FSRC`; recompute the `.d` files whenever necessary

- include the corresponding *.d* files for all files listed in CSRC and FSRC
- definition of VPATH for file searches
- search of header files and generation of proper compiler option for adding header file directories
- recursion into subdirectories listed in variable SUBDIRS
- Treatment of shells or interpreters in sharpbang (!) lines:
e.g. create an executable file *namefile* using the contents of *namefile.pl* and replacing some pre-defined strings with adequate values (like replace *#YOUR_PERL_INTERPRETER#* with the absolute name of *Perl* in the system)
- Creation of a testing environment (in the same window or in a new *xterm*) defining environment variables:
 - PRJROOT: if set to the project root directory then *maker* can be successfully executed from any subdirectory
 - PATH and LD_LIBRARY_PATH: extended with the proper directories in the build tree, they allow for using new executable files and dynamic libraries created in the build tree
- Creation of tag or debugger initialization files for a whole project tree
- Use compiler options for debugging and a different build tree when *maker --dbg* is issued.

Example

Figure 1 presents the Makefiles for a simple directory structure under directory “simulator” with two subdirectories called lib and include. Figure 2 shows the important directories used for a build performed on an Alpha ev6 processor (\$ARCH=ev6), maker variables, and files generated. Makefiles and files other than *prefetch.** have been omitted to simplify the graph. The other subdirectories under “.BuildTree” would be used on builds on other systems

Using maker a number of actions would be performed for the Makefiles presented above:

- Create directory *.BuildTree/\$ARCH* with a GNU-makefile inside
- Create directory hierarchy mirroring the source tree hierarchy under *.BuildTree/\$ARCH* (in this case it only creates one subdirectory called lib since the Makefile in directory simulator has SUBDIRS=lib and the ones in lib have no SUBDIRS
- Set *maker* variables: PRJROOT, PRJCWD, ...
- Change directory: *chdir* to \$PRJBT. Nothing to be done in there other than doing recursion into SUBDIRS
- *chdir* into \$PRJBT/lib, set PRJCWD to simulator/lib and call make with the user makefile \$PRJCWD/Makefile (the Makefile library and the

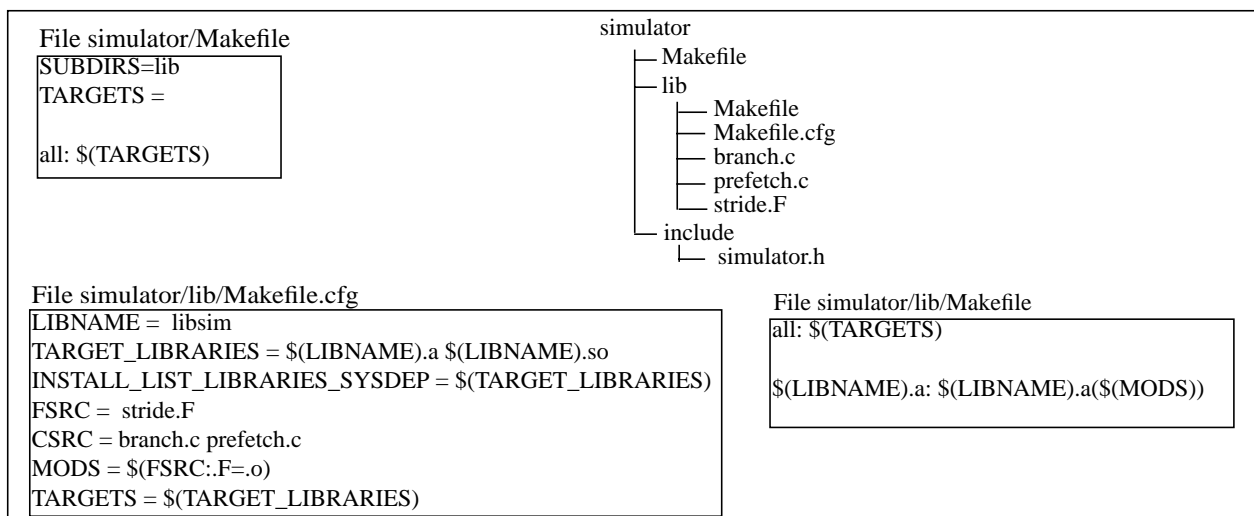


Figure 1: Makefiles for a simple directory structure

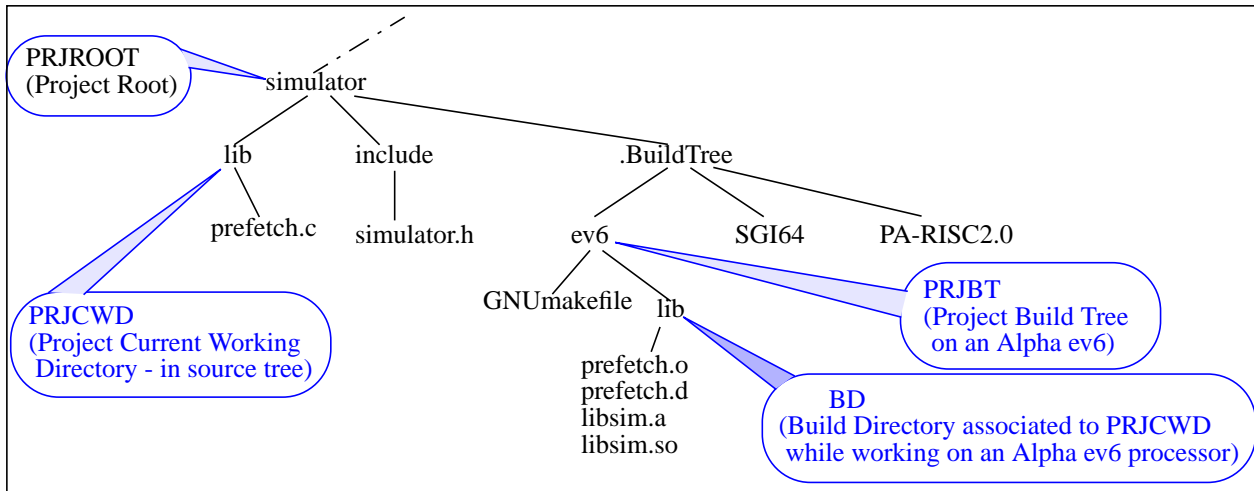


Figure 2: Directory structure

other user file Makefile.cfg will be loaded automatically)

- Directory \$PRJROOT/includes is automatically used for header files search
- Create dependency files for all files in CSRC and FSRC (branch.d, prefetch.d and stride.d) and include them
- Compile code into object files: branch.o, prefetch.o and stride.o
- Create libsिम.a
- Create libsिम.so
- If the user typed “*maker install*” files libsिम.a and libsिम.so would be installed in a system dependent directory for libraries. This directory is specified in variable LIBDIR_SYSDEP, whose value is set in Makefile.sys (unless it is overridden by the user).

Drawbacks

There a number of issues that can limit the use of maker by new users:

- It requires a Perl interpreter and the definition of an environment variable \$ARCH to work
- Debugging the Makefiles can become hard
- The *-n* option of make prints a large amount of information due to the complexity of the rule for automatic recursion into subdirectories.

4 Related Work

A number of tools exist which provide a front-end to make while extending its functionality. Amongst them there are two which are particularly relevant: imake [2] and automake + autoconf [6, 7]. It should be clear that maker was not thought as a replacement to these tools. Our main goal was not worldwide distribution of code as is their case.

Compared to these two systems Makefiles controlled by maker can be much shorter since they share many commonalities. This is particularly important when several build trees for different architectures are present in the same system. This could happen in a heterogeneous network of computers with a transparent file system access like the one provided by NFS [9]. Using automake and autoconf a whole new set of Makefiles would be created for each architecture. Also, the user would have to be aware of building the project in a different directory each time if she wanted them to coexist.

Using maker, the only requirement is that environment variable ARCH has to be set to the name of the current platform. Taken this into account, the rest is left to maker. Using it, there is only one version of each Makefile, which is stored in the source tree. The user does not need to change to

different directories for compilation since maker automatically does it. The default build tree root name is “.BuildTree/\$ARCH”. Consequently maker would create and use a new build tree each time a new machine was used for building the package. The proper platform description would be automatically used by maker and no changes in the Makefiles nor creation of new ones would be required. Separation into different trees automatically is very convenient: it makes things easier for the user and less error prone.

The definition of ARCH as an environment variable can be easily done in the shell initialization files. Actually, some shells already define variables like MACHTYPE or HOSTTYPE which can be helpful. Some systems provide a command which offers useful information: psrinfo, hinv, arch, mach. In an environment with heterogeneous systems sharing files with a Network File System [9] a script can be used to automate the process.

5 Conclusions

When software is created, the process of building the final executable can be eased if a program like make is used for directing recompilation. make is directed by the information written in Makefiles. Preparation of Makefiles can become tedious or difficult as the project grows or different platforms have to be supported.

We have presented a new approach to software development based on sharing Makefiles. An architecture has been defined which is flexible and extensible. Data used for directing builds is spread over several files in a logical way. Then, the system manages these files in a way that common data is automatically shared amongst Makefiles. The system architecture presented in this paper has been used to develop maker, a tool which has proved extremely useful for the authors when developing large software projects.

- Usual targets and rules can be used without having to write them in user Makefiles

- User makefiles are cleaner, shorter and easier to maintain
- Developing software in a heterogeneous system with a Network File System becomes much easier

We believe maker can be very useful for people interested in developing software, not Makefiles.

References

- [1] Adam de Boor, PMake - A Tutorial, University of California, Berkeley, CA, USA (Jul. 1988).
- [2] P. DuBois, “Software Portability with imake”, *O’Reilly & Associates*, 1st Edition, 1993.
- [3] Stuart I. Feldman, “Make - A Program for Maintaining Computer Programs,” *Software - Practice and Experience*, Vol. 9 No. 4, pp. 256-265, April 1979.
- [4] G. Fowler, The Fourth Generation Make, *In Proc. of the USENIX 1985 Summer Conference*, pp. 159-174, 1985.
- [5] A. Lih and E. Zadok, PGMAKE: A Portable Distributed Make System, *Tech. Report CUCS-035-94*, Computer Science Department, Columbia University.
- [6] D. MacKenzie and B. Elliston, “Autoconf: Creating Automatic Configuration Scripts.”, *User Manual, Edition 2.12, for Autoconf version 2.12. Free Software Foundation*, December 1998.
- [7] D. MacKenzie and T. Tromeu, “GNU Automake”, *User Manual, for Automake version 1.4, Free Software Foundation*, April 1999.
- [8] A. Oram and S. Talbott, “Managing Projects with make”, *O’Reilly & Associates*, 2nd. Edition, 1991.
- [9] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh and B. Lyon, “Design and Implementation of the Sun Network File System”, *USENIX Proceedings*, pp. 119-130, June 1989.
- [10] R. Stallman and R. McGrath, “GNU Make: A Program for Directing Recompilation”, *User Manual, Edition 0.51, for make version 3.76. Free Software Foundation*, August 1997.
- [11] L. Wall, T. Christiansen, J. Orwant, “Programming Perl”, *O’Reilly & Associates*; 3rd edition, July 2000.