MRO FTT/NAS & FLC

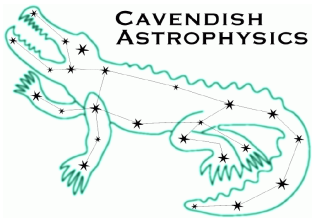
Software Build System

MRO-MAN-CAM-1160-0164

John Young

rev 1.0

19 December 2013



Cavendish Laboratory
JJ Thomson Avenue
Cambridge CB3 0HE
UK

Change Record

Revision	Date	Author(s)	Changes
1.0	2013-12-19	JSY	Initial version

Objective

To describe the build system used for the Fast Tip-Tilt and ICoNN software written in Cambridge.

Scope

This document describes the salient features of the build system implementation, to facilitate its adoption for new software projects, and to assist in integrating software that uses this build framework into the MROI build system. Familiarity with GNU Make and makefile syntax is assumed.

Reference Documents

RD1 GNU Make Manual

Applicable Documents

AD1 Miller, P.A. (1998), Recursive Make Considered Harmful, AUUGN Journal of AUUG Inc., 19(1), pp. 14-25.

AD2 Version Control with Subversion, Pilato, C.M., Collins-Sussman, B, & Fitzpatrick, B.W. (2008), O'Reilly.

Acronyms and Abbreviations

FTT Fast Tip-Tilt

FLC First Light Camera

ICD Interface Control Document

ICoNN Infrared Coherencing Nearest Neighbour fringe tracker

ISS Interferometer Supervisory System

MROI Magdalena Ridge Observatory Interferometer

NAS Narrow-field Acquisition System

NMT New Mexico Tech

TBC To be confirmed

TBD To be determined

Table of Contents

1 Introduction

The FTT and ICoNN software releases supplied by Cambridge use a build system based on recursive use of GNU Make. We expect to migrate the production delay line software to the same system in due course.

This document describes the salient features of the build system implementation, to facilitate its adoption for new software projects, and to assist in integrating software that uses this build framework into the MROI software build system. Familiarity with GNU Make and makefile syntax is assumed.

The build framework is part of a collection of the "camlibs" collection of software libraries, and includes support for linking the camlibs libraries when these are incorporated into the build tree.

1.1 Rationale

We have followed the conventional approach of using Make recursively. This means expressing dependencies at the directory level (as well as between files within directories) rather than in full. Such incomplete dependency information can result in unreliable builds as explained in AD1. Nevertheless, we believe that if the project is not too large, most of these issues can be avoided by thinking carefully when deciding how to segment the software. The recursive approach has the advantage of greatly simplifying the process of combining software from subversion repositories into a single build tree using `svn:externals` properties (see Chapter 3 of AD2).

1.2 Key Features

The noteworthy features of the build system are as follows:

- Uses standard build tools: GNU Make, GCC, and pkg-config.
- Provides rules for building C and C++ code and LaTeX documentation.
- Allows one-step building of multiple software applications and their prerequisite libraries.
- The system supports a mixture of Xenomai and standard Linux executables.
- Dependencies on header files within a directory are determined automatically by GCC.
- Static libraries that are built in order to build the applications do not need to be installed, hence several software releases containing different versions of the libraries can coexist on the same computer.
- The system can find installed libraries using the pkg-config utility, avoiding the need to hand-edit paths in makefiles.

2 Example Makefiles

The build system comprises three files that can be incorporated into makefiles using the **`include`** statement. We now describe how to use these include files by means of examples. The examples are for C code, but the framework also supports C++. We present examples of the three basic kinds of makefiles that can be written using the build framework:

- Top-level makefile: runs make recursively in a specified set of subdirectories
- Library makefile: builds one or more static libraries
- Application makefile: builds one or more software applications which depend on libraries elsewhere in the build tree

2.1 Top-level makefile

Here we present a simplified example of a makefile that runs make recursively in several subdirectories. There are dependencies between the subdirectories; these constrain the order in which the subdirectories can be processed.

```
# Top-level Makefile for mroi-ftt distribution

SUBDIRS = ext/camlibs ext/gsi/CControlSystem controlgui systems/FTTCamSystem

export CAMLIBS_DIR = $(CURDIR)/ext/camlibs
include $(CAMLIBS_DIR)/build/camlibs_recurse.mk

controlgui systems/FTTCamSystem: ext/camlibs ext/gsi/CControlSystem

# End of example
```

In the above example there are three subdirectories to build, expressed relative to the directory containing this makefile. Note that a subdirectory may be several levels down from the calling makefile, and that a makefile in a subdirectory may in turn recurse into lower-level directories.

The subdirectories to recurse into are listed in the SUBDIRS variable. The next two lines include a file camlibs_recurse.mk that contains a standard set of rules for invoking make in the directories listed in SUBDIRS. The definition of CAMLIBS_DIR is exported for use by subdirectory makefiles. Finally the dependencies between the subdirectories are expressed in the standard makefile fashion. The include statement precedes the dependency rules so that the default target is the one defined in camlibs_recurse.mk.

The file camlibs_recurse.mk defines the following (phony) targets. These are so-called “phony” targets i.e. they are not real files. Mostly these have the conventional meanings as outlined in RD1. The non-standard install-lib target installs libraries and their header files. The rationale for defining a separate target for this is explained in the next section.

- all (the default target unless the including makefile defines one)
- check
- install
- install-lib
- clean
- distclean

Lower-level makefiles should define the above targets, either directly or by including camlibs_recurse.mk or camlibs_build.mk.

2.2 Library makefile

Here is a simple example of a makefile that builds a static library and its unit test suite:

```
# Makefile for xdgdirs library

INCFILES = xdgdirs.h
LIBRARIES = libxdgdirs.a
TEST_EXES = utest_xdgdirs

# Link with the GLib library using pkg-config:
PKG_LIBRARIES = glib-2.0
CPPFLAGS =
LDLIBS =
```

```
CAMLIBS_DIR = $(CURDIR)/../..
include $(CAMLIBS_DIR)/build/camlibs_build.mk


check: utest_xdgdirs
	./utest_xdgdirs

libxdgdirs.a: xdgdirs.o

utest_xdgdirs: utest_xdgdirs.o xdgdirs.o

# End of example
```

In this case the makefile builds a static library libxdgdirs.a and a unit test executable utest_xdgdirs. Such libraries are normally only needed to build an executable elsewhere in the build tree, hence the install rule does *not* install the files specified by the INCFILES or LIBRARIES variables. If necessary, the install-lib rule can be used to install the libraries so they are available for building external software.

As in the previous example, there are three sections to the makefile:

- Variable assignments to define the targets to be built and options that control how they are built
- Include statement for a file camlibs_build.mk containing standard rules that build these targets
- Dependency rules. Note the absence of dependencies on C header files. These rules are generated automatically using GCC

The include file defines the following same (phony) targets as camlibs_recurse.mk. The following variables should be used at the top of the makefile to define the files built by these targets:

module	Defines install path for MODULE_DATA and MODULE_DOCS. A project could define any number of distinct values that apply to different subdirectories
EXES	Executables to be made and installed (by the install rule)
TEST_EXES	Executables to be made but not installed
SCRIPTS	Executables to be installed (by the install rule) but do not delete under the clean rule
LIBRARIES	Libraries to be made and installed by the install-lib rule
INCFILES	Include files to be (optionally made and) installed by the install-lib rule. If made, you should add them to REMOVE_TARGETS
MODULE_DATA	Module-specific data (architecture-independent), to be installed in $(prefix)/share/$(module) by the install rule
MODULE_DOCS	Module-specific documentation, to be installed in $(prefix)/share/doc/$(module) by the install rule
REMOVE_TARGETS	Extra files to be deleted by the clean rule

The following variables control compilation and linking:

PKG_LIBRARIES	Libraries to compile and link against. The **pkg-config** utility must have been installed together with meta-information for the specified libraries. If you are unable to use pkg-config for a particular library, use CPPFLAGS and LDLIBS instead.
XENO_SKINS	Xenomai skins to compile and link against. Xenomai, including the **xeno-config** utility, must have been installed. Possible skins include 'native', 'posix' and 'rtdm'.
CPPFLAGS	Arguments for the C/C++ preprocessor, e.g. specifying extra include directories for cpp to search
LDLIBS	Library arguments for the link command

Further documentation is available in the comments at the top of camlibs_build.mk.

2.3 Application makefile

The following subdirectory makefile builds an executable which is linked against several libraries that are built in other directories (again this is a simplified example of a real makefile).

```
# Makefile for flcgui

# Depends on camlibs libraries: fileutil, gui, msgbase, msgevent

objects = FlcEngGui.o DataHistory.o FlcSysGui.o FttEnvGui.o fitsimage.o

module = flcgui
EXES = flcgui

PKG_LIBRARIES = gtk+-2.0 gmodule-2.0 gtkimageview
CPPFLAGS = $(CPPFLAGS_msgevent) $(CPPFLAGS_msgbase) $(CPPFLAGS_gui) \
 $(CPPFLAGS_fileutil)
LDLIBS =
LDFLAGS = -export-dynamic

CFLAGS = -g -Wall -std=c99


CAMLIBS_DIR = $(CURDIR)/../../ext/camlibs
include $(CAMLIBS_DIR)/build/camlibs_libs.mk
include $(CAMLIBS_DIR)/build/camlibs_build.mk
vpath %.a $(VPATH_msgevent) $(VPATH_msgbase) $(VPATH_gui) $(VPATH_fileutil)


flcgui: flcgui.o $(objects) $(LIBS_msgevent) $(LIBS_msgbase) $(LIBS_gui) \
 $(LIBS_fileutil)

# End of example
```

The principal difference from the library makefile is the way that the prerequisite header files and libraries are specified. This is done by including a further file, camlibs_libs.mk. This include file defines a triplet of variables CPPFLAGS_lib, VPATH_lib and LIBS_lib for each of a number of library "groups". Each library group comprises a set of header files and static libraries which might be used independently of the others.

For example the variables for the "fileutil" library group (which only contains one library) expand to:

- CPPFLAGS_fileutil is `-I$(CAMLIBS_DIR)/fileutil/xdgdirs`
- VPATH_fileutil is `$(CAMLIBS_DIR)/fileutil/xdgdirs`

- LIBS_fileutil is `-lxdgdirs`

These sets of variables are used with GNU Make's **`vpath`** statement as follows. The libraries, as specified by LIBS_lib , are given as explicit prequisites of the executable (here flcgui). The **`vpath`** statement is used to tell Make where to find these libraries, making use of the VPATH_lib variables. The search paths for the header files are set in the CPPFLAGS variable, making use of CPPFLAGS_lib variables. CPPFLAGS is used by the build rules defined in camlibs_build.mk that compile both C and C++ code.