Building c++ code with JMakefile

January 22, 2026

Abstract

This note contains a brief description of the implementation of the make process within Jpp. Examples of how to use the Makefiles included in Jpp to compile a c++ project are given.

1 Jpp makefiles

The makefiles included in Jpp can be used for several purposes. Besides compiling the Jpp source code, one can build .pdf documents from .tex files, they can be used to build c++ projects including the compilation of dynamic libraries, and one can also convert libreoffice documents to .pdf files. The instructions and variabes that make will follow are distributed in several makefiles organised according to the structure shown in figure 1. If one takes a look at these

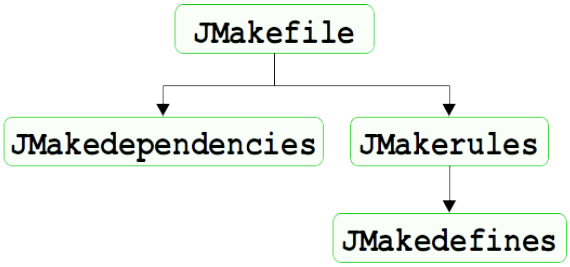

Figure 1: Structure of the jpp Makefiles. JMakefile includes JMakerules and JMakedependencies, the former of which includes JMakedefines.

makefiles, they look as follows.

```
#JMakefile
-----------------------------------
include $(JPP_DIR)/JMakerules
include $(JPP_DIR)/JMakedependencies
```

`JMakefile` is rather simple. It only includes `JMakerules` and `JMakedependencies`. `JMakerules` is structured as follows,

```
#JMakerules
-------------------------------------
include $(JPP_DIR)/JMakedefines
{
  definition of pattern rules
}
{
  definition of file lists and assignation of default values
}
```

where

```
#JMakedefines
-------------------------------------
{
  definition of auxiliary functions
}
{
  definition of global variables
}
```

Finally, `JMakedependencies` has the following structure

```
#JMakedependencies
-------------------------------------
{
  definition of phony targets and top level rules for lists of flies
}
{
  inclussion of dependency files generated by the compiler
}
```

In what follows, a description of the most relevant functions, variables and rules needed to compile a `c++` project is given.

1.1 Auxiliary functions

The functions defined in `JMakedefines` are mainly functions for the treatment of strings of characters. Among these functions one can find for instance the `insert` function, which inserts a path in a string of paths and which can be used to update environmental variables such as `LD_LIBRARY_PATH` or `PATH`. Another interesting function is the `binary` function, which given a list of `c++` source code files (with `.cc` extension) finds which files contain the `int main` method and returns a list with the corresponding binary file names (ie, the same file names but without the `.cc` extension).

1.2 Global variables

Variables related to the multiple actions that can be carried out by the `Jpp makefiles` are defined inside `JMakedefines`. Some of them are mentioned in the following sections, and a brief description of other which are relevant for the compilation of `c++` projects is given in the appendix 2.2

1.3 Pattern rules

Several pattern rules are included in `JMakerules` that implement the compilation, linking and document generation. The pattern rules for compiling `c++` source code files into object files and for linking of object files into binary files are shown below

```
#Compilation pattern rule
-------------------------------------
%.o: %.cc %.d
  $(CXX) $(CXXFLAGS) $(CPPFLAGS) -c -o $@ $<

#Linking pattern rule
-------------------------------------
%: %.o
  $(CXX) $(LDFLAGS) -o $@ $^ $(LOADLIBES)
```

`CXX = g++` is the compiler and `CXXFLAGS` and `CPPFLAGS` are lists of options that can be passed to the compiler and the preprocessor respectively. `CXXFLAGS` should include the list of paths pointing to the location of the `.hh` files included in the `c++` source code files that are compiled. `LDFLAGS` is a list of paths pointing to the location of external libraries needed to link several objects into an executable, and `LOADLIBES` is a list of external libraries. These variables are set to some default values by `JMakedefines` which are listed on 2.2.

1.4 Native and Public file lists

`JMakerules` includes the definition of `NATIVE_` and `PUBLIC_` variables to store file lists. The native file lists are reserved for those files existing in the directory from which `JMakefile` is called (or "current directory"). The variable `NATIVE_SRCS` will be filled with a list of all the files with `.cc` extension, the variable `NATIVE_OBJS` will contain the list of corresponding object file names, and `NATIVE_BINS` will contain a list of binary file names corresponding to every source file that contains a `main` function (this is achieved thanks to the `binary` function described above). In addition, `NATIVE_SCRIPTS` conatins a lists of `.sh` files and `NATIVE_DOCS` a list of `.tex, .docx` and `.pptx` files. If the current directory is a sub-directory of `$JPP_DIR/software`, then the lists of files are stored in the public variables (`PUBLIC_SRCS`, `PUBLIC_BINS`...etc).

1.5 Phony targets

`JMakedependencies` includes the declaration of several phony targets. Among those, the most relevant are `default, all, install` and `clean`. The default target (ie the first target appearing in the makefile structure, which is the one that will be remade if no target is specified when

launching make) is `default`, which in turn has `all` and `install` as prerequisites. The prerequisites of `all` are the `NATIVE` lists, and the prerequisites of `install` are the `PUBLIC` lists. Therefore, by default all the files in the current directory are remade according to the corresponding pattern rules.

2 Using `JMakefile` to build a `c++` project

Since `JMakefile` implements the majority of the functions and rules needed by make to compile and link `c++` code, using it to build one's own project is rather straightforward. One just have to create a `Makefile` that includes `JMakefile`, where the compilation and linking variables are accordingly updated with the desired library paths, include paths and external libraries. By default, `JMakefile` doesn't know which object files need to be linked into a binary file. If a binary file is obtained from the compilation and linking of more than one source code file, the user should specify this information in the `Makefile`. This is simply done by writing an explicit rule with the final binary file as a target, the object files to be linked as prerequisites. `JMakefile` will use the appropriate pattern rules to generate the object files and the final binary file. A typical `Makefile` would be like this:

```
#My Makefile
-------------------------------------
include $(JPP_DIR)/JMakefile

LDFLAGS += -L/path/...
CXXFLAGS += -I/path/...
LOADLIBES += -l...

main: main.o lib1.o lib2.o ...etc
```

The **first example** included with this document consists on three `c++` files plus a `Makefile`. The `c++` files consist on `JMain.cc`, where the main method calls the `lib` function that is defined in `JLib.hh` and implemented in `JLib.cc`. The `Makefile` to build the `JMain` binary file is simply

```
#My Makefile
-------------------------------------
include $(JPP_DIR)/JMakefile
JMain: JMain.o JLib.o
```

The **second example** compiles the same binary file as in the first example, but following a different strategy that better suits with the `Jpp` philosophy. This is, defining and implementing the classes and functions in a single include file. In this case the project consists only on two `c++` files plus the `Makefile`. The diference with respect to the previous example is that in this case the `JLib.cc` file doesn't exist, and both the definition and implementation of the `lib` function is done inside `JLib.hh`. In this case the rules included in `JMakefile` are enough to build the only the only file to be compiled is JMain.cc, and the `Makefile` can be simplified to a single line.

```
#My Makefile
------------------------------------
include $(JPP_DIR)/JMakefile
```

2.1 Using JMakefiles to build Jpp for ANTARES

Jpp is developed for KM3NeT, but it is also suited for ANTARES. In the case that one wants to apply Jpp to ANTARES, it is necessary to specify it at the time of building Jpp. This is done by typing

```
make Antares
```

As a result, JMakefile will assign the value ANTARES to the variable NAMESPACE (whose default vaule is KM3NET) which is passed to the preprocessor via the CXXFLAGS during the Jpp compilation. As an example of how this variable is used in the Jpp libraries one can take a look for instance, at the following lines in $JPP_DIR/examples/JSirene/JPMT.hh

```
#include "JPhysics/Antares.hh"
#include "JPhysics/KM3NeT.hh"
  ...
  ...
  using namespace NAMESPACE;
  ...
  ...
```

The c++ preprocessor will recognize and expand the macro NAMESPACE to the value chosen in the Make execution. The ANTARES and KM3NET namespaces are defined in the $JPP_DIR/software/JPhysics/Antares.hh and $JPP_DIR/software/JPhysics/KM3NeT.hh files respectively. They mainly contain information about the PMT characteristics and water properties for each instrument. Through a similar procedure, the number of PMTs is defined for each instrument in the $JPP_DIR/software/JDAQ/JDAQ.hh file.

2.2 Parallel processing

JMakefile is able to execute several recipes at the same time through the use of the -j option:

```
make -j
```

In the make language, the amount of recipes executed simultaneously is called the number of *job slots*. This number can be chosen by writing an integer number after the -j option. If no number is specified, the amount of job slots is not limited. The operative system will take care of multitasking the number of requested job slots, and the gain in speed with respect to sequential execution depends on the number of cores in the available in the CPU.

To see the effect of the -j option on the Jpp make procedure one can do the following test in the compilation of a project,

```
time make
make clean
time make -j
```

and compare the needed time in both cases.

Appendix A: relevant variables defined in `JMakefile`

Here, a list of variables defined or updated in `JMakedefines` that one may need to use in an owns `Makefile` is given.

Environmental variables

The envionrmental variables `LD_LIBRARY_PATH` and `PATH` are updated by `JMakedefines`:

- `PATH`: The path `$ROOTSYS/bin` is appended to the `PATH`
- `LD_LIBRARY_PATH`: The paths `$ROOTSYS/lib , JPP_LIB` and `AANET_LIB` are appended to the `LD_LIBRARY_PATH`

Default g++ options

The default values of the compiler variables mentioned above include the following libraries and paths

- `CXXFLAGS` include the following paths pointing to include files: `JPP_INCLUDE, ROOTINCL, AANET_INCLUDE(/evt /util), ANTARES_INCLUDE`.
- `LDFLAGS` include the following paths pointing to external libraries: `JPP_LIB, AANET_LIB, ANTARES_LIB`.
- `LOADLIBES` includes by default `JPP_LIBS` and `ROOTLIBS`

see below for a description of these variables.

`Jpp` variables

- `JPP_INCLUDE = $JPP_DIR/software`: Path to the location of `Jpp` include files.
- `JPP_LIB = $JPP_DIR/$SYSTEM/lib/`: Path to `Jpp` libraries. Here, the `SYSTEM = Linux` is another variable. The variables for different of these libraries are the following:
  - `JPP_LIBS = -llang`
  - `JDAQ_LIBS = -lKM3NeTDAQROOT`
  - `JDAQ_CHSM = -lDAQ_CHSM`
  - `JTRIGGER_LIBS = -ltriggerROOT`
  - `JEVT_LIBS = -levtROOT`

  - JAANET_LIBS = -ljaanetROOT
  - JCOMPAREHISTOGRAMS_LIBS = -lcomparehistogramsROOT
  - JMARKOV_LIBS = -lmarkovROOT

aanet variables

- AANET_INCLUDE = $JPP_DIR/externals/aanet/: Path to the location of aanet .hh files.
- AANET_LIB = $JPP_DIR/externals/aanet/: Path to the aanet external libraries. The variable containing the libraries is
  - AANET_LIBS = -laa

Database

- JDB_INCLUDE = $JPP_DIR/externals/dbclient/include
- JDB_LIB = $JPP_DIR/externals/dbclient/lib
  - JDB_LIBS = -lKM3NeTDBClient

Antares

- ANTARES_INCLUDE = $JPP_DIR/externals/
- JDB_LIB = $JPP_DIR/externals/antares-dataformat/
  - ANTARES_LIBS = -lAntaresDAQROOT

ROOT

- ROOTLIBS = $ROOTSYS/bin/root-config --libs
- ROOTGLIBS = $ROOTSYS/bin/root-config --glibs
- ROOTINCL = $ROOTSYS/bin/root-config --cflags
- ROOTCINTFLAGS = -I$(JPP_INCLUDE) -DNAMESPACE=ANTARES / KM3NeT