RMR Building From Source

If you wish to build from source, the repo can be cloned with: git clone https://gerrit.oran-osc.org/r/ric-plt/lib/rmr After cloning, follow the instructions in the BUILD file.

To align with standard practice regarding package names and contents, the build process will generate one of two possible package types:

- A *runtime* package containing only the RMR shared object (.so) files
- A *development* package containing only the RMR archive (.a) and necessary header (.h) files, and manual pages

The build process is described in the `BUILD` file at the root of the RMR repo. In short, supplying `-DDEV_PKG=1` when CMake is used to configure the project, will cause the ensuing make package command to generate a development package.

Some developers may need to also include the underlying, external, transport libraries (e.g. libnng) in the package in order to make unit testing of wrappers (bindings) stand-alone. To do this, the CMake option `-DPACK_EXTERNALS=1` should also be added to the CMake command line. (See the CMake examples paragraph later on this page.)

Build Requirements

RMR requires the following utilities/tools in order to build

- cmake
- gcc
- g++ (for NNG)
- git
- make

NNG is referenced from within the repo as an external resource, and if not already installed on the system will be pulled and built.

Configure and Build RMR

RMR is built via CMake which must first be executed to set the configuration options for the intended build. Specifically this allows the user to select the type of package to be produced (development or production), and to specify options such as where to install library and header files.

To build:

1. Clone the repo and switch to the top level directory
2. Create the .build directory (mkdir .build)
3. Switch to the .build directory (cd .build)
4. Configure with CMake (see note on options below)
5. Build (See examples below)

CMake Configuration Options

For developers, it is likely that configuration options will be useful:

- the option to install the underlying transport libraries (e.g. NNG) along with the RMR libraries eliminates the need to install the transport libraries:
  `-DPACK_EXTERNALS=1`
- the option to install RMR, and transport libraries, into an alternate directory:
  -DCMAKE_INSTALL_PREFIX=/tmp/$LOGNAME/
  This will place libraries and header files into a directory based on the user name in /tmp (makes cleanup and/or complete removal much easier.
- the option to generate headers and a development archive:
  `-DDEV_PKG=1`
  This option will likely be required for C and Go developers as this causes the RMR header file (rmr.h) to be placed into the include directory. For Python programmers, it is likely that this option is **not** needed as the RMR bindings need to reference the shared object library (.so) which is not generated with this option.
- the option -DBUILD_DOC=1 will cause the RMR man pages to also be built. If creating a package they will be installed in the usual system location, or will be left in the build directory in three forms: postscript, TROFF input, and restructured text (rst).

Setting the environment variable LD_LIBRARY_PATH to reference either the standard library installation location, or the alternate path provided as in the previous example, will likely be necessary. For C progammers, the C_INCLUDE_PATH environment variable will also need to be set if an alternate install location was used (this may also apply to Go programmers).

CMake Commands

These should be useful regardless of the development environment. These commands assume that the current working directory is .build which was created at the top of the repo directory tree.

CMake command examples

```
# configure and generate a production package
cmake ..
make package

# configure and generate a development package
cmake .. -DDEV_PKG=1
make package

# configure and install in an alternate directory, and install transport (nng) libraries
cmake .. -DEV_PKG=1 -DCMAKE_INSTALL_PREFIX=/tmp/$LOGNAME/ -DPACK_EXTERNALS
make install

# configure and install in the preferred system lib/include directory path; builds both package types
# (using sudo or switching to the root user might be needed if the install prefix was not changed to
# a user writable directory.)
cmake .. -DDEV_PKG=1
make install
cmake ..
make install
```

Verification

Before building a package, or installing, the unit tests can be executed to verify that the repo clone is intact and working on the system. Unit tests are located in the test directory and can be executed by running the unit_test.ksh script; originally the script required Kshell, and thus then suffix of .ksh. The script has been changed (March 2020) to remove the Ksh features which were used making it less efficient, but more generic with respect to shell choice.

NNG Installation

When using RMR on top of NNG, the NNG libraries must also be installed on the system. This can be accomplished by building and installing NNG independently, or configuring the RMR build process to install NNG in addition to RMR. Adding the CMake option -DPACKAGE_EXTERNALS=1 to the configuration command line causes the NNG libraries to be packaged with RMR or installed when 'make install' is executed.