Understanding the Neutrino Build Process#

Background#

The Neutrino build process uses makefiles (GNU make) to recursively traverse through a source module picking up source at various levels and compiling them into object files. These object files are placed at the leaf nodes and then assembled into whatever their final target may be, generally an executable or a library.

This build structure allows Neutrino to maintain a common source base that can be built concurrently for multiple operating systems, CPU architectures and other variants such as debug, endian etc. This is important to keep the code base manageable while at the same time supporting 5 different CPU architectures over 2 endians.

The QNX recursive makefile structure is described in the general QNX documentation in the Neutrino Programmer's Guide in the appendix Conventions for Makefiles and Directories.

General Build Flow#

Ignoring the set-up, building the general tree takes two distinct steps, as covered in the BuildTheOSSource page:

1. Transferring header files from the source repository to a general staging area (make hinstall)
2. Build the binary that you want to target (default make target or make all)

Lets examine the details of these steps.

Transferring the header files
: This step is required because the Neutrino source tree is not self referencial with respect to its header file inclusions. This means that if I have a public header file in lib/c (say inttypes.h) and I have a module in services/foo that would like to use it, I don't do either of the following:

- Add an include path that looks like -I$(PROJECT_ROOT)/../../lib/c/public
- Add an include statement that looks like #include <../../lib/c/public/inttypes.h>

The reason is that as maintainers of a lot of software, it makes the most sense to keep the header files that expose a public API *close* to the source that implements their functionality. This means that while a particular service may be generic, the functionality for the service is mainly in a library and that is where the header file lives. If full relative paths up and down the repository were used, it would end up being a large maintenance activity and it would make it nearly impossible to move source code around once it had been established. Not a good software engineering practice at all!

Build the binary
: Performing a top level build takes a while. If you want the binaries (libraries and executables) to end up in the staging area, then you will need to run the **make install** build command. Otherwise the default command is **make all** which will perform the build but leave the resulting binaries in the source tree without copying them to the stage location.

Anatomy of a source module#

The structure of the entire source repository is covered in depth in the UnderstandingTheNeutrinoSourceTree, but it is worthwhile looking at what components of a source module are used in the build process.

A source module will generally(1) contain the following entries:

- A Makefile file that contains a stub for the QNX recursive makefiles

```
% cat Makefile
```

```
LIST=CPU
include recurse.mk
```

- A common.mk file contains the build directives for the source module

```
% cat common.mk
ifndef QCONFIG
QCONFIG=qconfig.mk
endif
include $(QCONFIG)

INSTALLDIR=usr/sbin

define PINFO
PINFO DESCRIPTION=This is a nifty module
endef

include $(MKFILES_ROOT)/qtargets.mk
```

- A module.tmpl file that contains descriptive information about the source module and information used for automated build and packaging
- The CPU or OS variant directories, for example x86, mips, sh, arm and/or ppc
- A directory named public. This is only present if the module provides any public header files. The content of this directory will be copied over entirely into the stage area when a **make install** or **make hinstall** command is performed on the module.

<hr> 1: Really, no two source modules are identical. This is pretty much all made up based on an ideal reality.

Tuning Your Build#

If you are performing blanket top level builds across broad parts of the source tree with many modules ... the building will be easy but time consuming (likely a two coffee exercise with a walk around the building in between). However, you can also build modules in a much more targetted fashion taking advantage of your knowledge of the code structure, what changes you are interested in and what parts of the tree you are working on and that have relevant modifications.

Build Environment Variables

The first thing to do is to understand what you can tune with respect to basic environment variables used by build system:

OSLIST
: In general you will always want this to be nto to identify that this is a build for the Neutrino. There are some source modules which can be compiled for other systems however and setting this will allow you to skip any additional processing.

CPULIST
: This is likely the one you will use the most often. It controls which architectures that you are building for. If you are only interested in PPC systems, then it is not really worth spending your time watching all of the ARM variants get built. Save some time and pick one processor to do most of the development on, then run a global compile when you are nearing completion to make sure you at least build for the other architectures and endians. Of course we know that you will test on those too once you get something working!

VARIANTLIST
: This one is particularly handy if you are building library source modules (**so** and **a** variants) as well as for controlling the endian's you are compiling (**be** and **le** variants)

CP_HOST_OPTIONS
: This one is sneaky! Some platforms allow you to customize the copy command so that if there is no work to be done because the source and destination file are the same (or destination is newer) then you can use this to speed up the hinstall and install stages by adding in a *-n*

In addition to these variables, the EXCLUDE_* environment variables can be used to control which items are removed if removing a variant is easier. For example if you were to say
`CPULIST=x86 arm sh mips`
in order to remove the ppc variant, it may be easier to simply say
`EXCLUDE_CPULIST=ppc`
instead.

The biggest performance win is going to come by appropriately setting the *CPULIST* option. Consider that the Neutrino build system by default will build for the 5 target architectures (PPC, MIPS, SH, ARM, x86) and that for three of these architectures there are both big and little endian versions, we will end up compiling the same source code into 8 different binaries: ppcbe, x86le, mipsle, mipsbe, shle, shbe, armle, armbe.

So if we were to revisit the build commands described in building the OS source and perform a build only targetting little endian ARM CPUs:

1. Install all of the header files in the local stage
   ```
   % cd $BUILD_ROOT/trunk
   % make CPULIST=arm VARIANTLIST=le OSLIST=nto hinstall
   ```
   ... lots of output, including copying to $BUILD_ROOT/stage
2. Build the Neutrino OS binaries
   ```
   % cd $BUILD_ROOT/trunk
   % make CPULIST=arm VARIANTLIST=le OSLIST=nto install
   ```
   ... lots of build output, binaries copied to $BUILD_ROOT/stage

---

Performing Targetted Builds

TODO: Fill in details here about how to avoid doing complete make hinstall's or make install's and just building specific source modules as required.

Magic Makefile File Locations#

If you want to dive even further into the construction and operatoin of the QNX recursive makefiles, or to debug the makefiles to understand why something is or is not happening, then you need to go to the source. The bulk of the magic of the QNX recursive makefiles takes place in the following files:

- $QNX_TARGET/usr/include/recurse.mk
- $QNX_TARGET/usr/include/qconfig.mk
- $QNX_TARGET/usr/include/buildlist.mk
- With support of the $QNX_TARGET/usr/include/mk