Note: This project does NOT include source access.#

This guide is intended to help developers who want to work with the QNX code tree get up and running as quickly and as painlessly as possible.

It is important for developers to understand how the build system and source layout is put together, so this "quick start" is more like a "comprehensive start" that also explains the source layout and build mechanisms to allow you to experiment with the latest source while keeping your installed development environment clean.

Note: The instructions are equally applicable for the IDE environment as they are for the command line environment.

Source Code Organization#

Subversion (SVN) is used to manage the source repositories. Each technology project maintains its own repository, however the structure of those repositories are very similar, differing only based on the technology content. @@@ What SVN clients are available?

The QNX source repositories are laid out in a hierarchichal manner and use the QNX recursive makefile macros to configure and execute the build operations.

Due to the way in which the source is built, it is best to maintain the source hierarchy. This will facilitate the build and reduce the amount of complication you have in your life.

The code is structured in a way that facilitates partially identifying the role of the module:

```
.../trunk/
  |- lib
  |- services
  |- hardware
  |- utils
  |- apps
  |- tools
  |- ...
```

The names are designed to be fairly clear and self-explanatory:

lib
: Library code used my multiple applications, services and drivers (libc, libm, libz, etc)

services
: System services, daemons and frameworks (kernel/procnto, pipe, mqueue)

hardware
: Hardware-specific components, including the startup and board support packages.

utils
: Utility applications, sorted and structured alphabetically (e.g. 'cp' is in utils/c/cp).

apps
: General applications (generally graphical and photon based)

tools
: Core development tools (e.g. gdb, gcc)

ports
: Ports of software from other systems

Not all projects will have components from all parts of the source tree, however having entries in lib|services|utils is common.

As much as possible source that is related to a component is kept close to that component in the tree. This means that public header files do not all live in a common directory, but rather "live" with source that they are most relevant for. For example, the lib/m module contains all of the public math library headers while services/slogger contains the header file that contains the system logging codes. These public header files are all transferred to a common area as part of the build process.

Development Environment Selection#

The Momentics SDK provides the ability to build software targetting Neutrino from Windows, Linux, Solaris as well as Neutrino x86 (self-hosted). The majority of the build environment and infrastructure was designed to run under a Neutrino self-hosted environment, but any one of the development hosts should be capable of building the source.

Neutrino	Linux	Windows
Pros *Original Neutrino build environment *Rapid development & prototyping	**Pros** *Standard Unix environment	**Pros** *Standard business environment
Cons *Non-standard, but unix, environment	**Cons** *Immature Neutrino build environment *Can't build configure based entries	**Cons** *Immature Neutrino build environment *Can't build configure based entries

Ultimately the entire QNX source will be able to build under all development hosts, but depending on the modules you are working with you may need to make some build adjustments.

Building the Source#

As mentioned in the source code organization, the source uses the QNX recusive makefile structure to facilitate the building of source for multiple target architectures as well as providing multi-operating system support all from a single source base.

This type of build is required because currently Neutrino source is built for eight (8) different architectures (x86, ppc-be, mips-be, mips-le, sh-le, sh-be,arm -le, arm -be) and some of the utilities that are part of the Momentics SDK must also build natively for Windows, Linux and Solaris.

- Specialized build/compilation environments are required for non-Neutrino builds and are not discussed in this guide

Before continuing, it is worthwhile to take a quick read through:

@@@ Insert Link to Help Documentation on QNX Makefiles

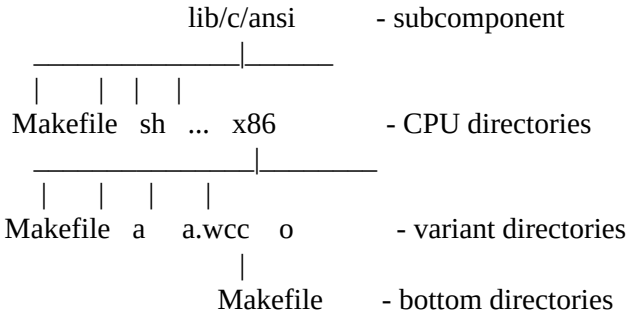
The makefile system uses a recursive structure. If "make" is executed from a directory, everything below that directory is built. Each major component has a Makefile at its top level. In order to build the entire component with all of its targets and variants, just execute "make" from the top level.

However, building all of the different variants can take a very long time...

In general, during development a developer only needs to build one particular target of a subcomponent.

The directory structure is built up in levels. Each major component (e.g. lib) contains subcomponents (e.g. lib/c/ansi) in separate directories. Each subcomponent has a variety of targets: if you "make" in the subcomponent directory, it will build the subcomponent many times: for different CPUs (e.g. x86, PPC, SH), for different OSs (e.g. Neutrino, QNX4) and for different variants (e.g. debug, little-endian vs. big endian).

Note: Some of these different target dimensions are optional -- for example not every component is differentiated by OS. The targets are defined by the directory structure:



The Makefile contents are usually quite simple. For example, lib/c/ansi/Makefile: `LIST=CPU include recurse.mk`

The "LIST=CPU" line tells the makefile system that the subdirectories at this level define CPU targets. The "include recurse.mk" line tells the makefile system that the normal subdirectory recursion should be done.

The "LIST=" element of the different makefiles is quite important -- it provides an important mechanism we will use to restrict the build to just the piece we need built.

Source Build Targets#

If "make" is issued without a target, then this is equivalent to typing "make all", the component of the current working directory, along with any subcomponents and targets reflected by the directory structure below the current working directory, will be built (compiled and linked).

These are make targets that will typically be used:

- all - The default build target
- clean - Cleans the component's directory structure of all build products (object files, binaries, etc)
- hinstall - Install the component's public headers into the staging area
- install - Build the component and install it (headers and binaries) into the staging area

---

Setting Up A Stage Area#

The components of the source tree do not include explicit dependency information. Since the source being built is *generally* being bootstrapped for the "next" version of the operating system, using the headers and

dependencies available as part of the local build environment (for example the headers that are installed as part of the base Momentics development environment) is not practical. Also, since public header files are distributed across the tree, it is not practical to encode a large set of dependencies into each source module as a large set of #include search paths. To top that all off, you don't want to be "installing" experimental libraries and binaries on top of a working stable installation, so you would rather install them to a distinct and separate location.

... these problems are solved by using a stage area.

A stage area is a directory location that mimics the local installation path(s) that one finds under the development hosts' QNX_TARGET environment variable. It is built up based on the content from the tree when a make hinstall (for headers) or make install (for headers & binaries) is performed.

The staging area path is selected by setting the INSTALL_ROOT_nto makefile variable to the base path where header files and libraries are to be installed. Additionally, you need to set the USE_INSTALL_ROOT macro (the value doesn't matter). This tells the makefiles to search the INSTALL_ROOT_nto tree when the compiler and linker are seaching for headers and libraries It is cumbersome to set these values each time you perform an installation, so the build environment facilitates the setting of these variables through the use of a single override makefile that is specified using the QCONF_OVERRIDE environment variable.

Clear as mud? Let's review:

1. When performing a build of the current source, you need to use current headers (and potentially libraries for linking)
2. The local source module, as extracted from SVN will be the first path referenced for include files and libraries
3. The path pointed to by the INSTALL_ROOT_nto makefile macro will then be used for include files and libraries
4. Finally the locally installed (as part of the Momentics devlopment environment) headers and libraries will be used

Here is a sample (Windows):

```
C:\>echo %QCONF_OVERRIDE%
c:/thomasf/master-qconf-override.mk
C:\>cat c:/thomasf/master-qconf-override.mk
HERE := $(shell $(PWD_HOST))

###
# This assumes that you always checkout into a directory called "trunk"
# as the root of your source tree and then creates the "stage" directory
# as a peer to the trunk directory
###
ROOTDIR := $(findstring C:/thomasf/mediaware/trunk,$(HERE))
ifeq ($(ROOTDIR),)
ROOTDIR := $(findstring C:/thomasf/630SP3/trunk,$(HERE))
endif
ifeq ($(ROOTDIR),)
ROOTDIR := $(findstring C:/thomasf/633/trunk,$(HERE))
endif
ifneq ($(ROOTDIR),)
INSTALL_ROOT_nto :=$(subst trunk,stage,$(ROOTDIR))
USE_INSTALL_ROOT=1
endif
```

Actually Performing The Build#

Once you have checked out the source, configured your environment and set-up a staging area, you are finally ready to do a build. If you are starting from an empty stage directory then you will need to execute all of these scripts. Once you have established a baseline, then you will only need to re-do the make hinstall/make install steps on components that are updated as they are updated.

From the root of the tree (unless otherwise declared in the build instructions)

1. make OSLIST=nto hinstall "(Go for coffee)"
2. make OSLIST=nto install "(Go for a coffee and a doughnut)"

The first hinstall ensures that all of the public headers are in place before any of the binaries are attempted to be created. The second installation should sweep through and then build all of the binaries. If you take a look at the Makefile for the toplevel directory, you will see that there is an explicit ordering to ensure that libraries are built and installed before services are built (that may depend on the libraries being present in the stage area).

Congratulations! After this completes you should now be fully populated with binaries and headers in your stage area.

Contributors#

- Doug Bailey (Initial Content Seeding)