Welcome to the QNX Foundation Classes Project#

The QNX Foundation Classes (QFC) is a library of fundamental C++ classes for the QNX™ operating system. It provides the following :

· Complete Resource Manager framework

· QNX IPC template class (Channels and Message Vectors)

· Input Output Selector class

· Timer services

· Adaptive Partitioning services

· System Analysis Toolkit services

· Threads

· Thread pool

· Iostreams logging

· Completely documented using Doxygen

· And more…

What's in it for developers#

C++ developers looking to create QNX specific applications (some classes are portable, but most are QNX specific), can quickly build applications around the most common QNX models (for example, resource managers and message based IPC).

The Resource Manager Framework provides a fully implemented resource manager with default methods that (by simply instantiating an instance of the class) creates a fully functional ramdisk (with symbolic link support). By overriding methods within this framework, the resource manager can take on any type of behavior (examples provided).

The IPC template class provides a form of compile time type safety for messages. A server provides an interface definition which details the classes that it can receive. A client application includes this interface, and instantiates a "type safe" instance of the Connection class, any attempt to send an unsupported class to the server, will be caught at compile time (and with the advent of concepts in C++09x, they will even be understandable :-). At this time an invalid sequence of valid classes can not be caught at compile time.

The Input Output Selector class provides a single point dispatch for applications to receive messages/pulses/ select notifications. This helps produce single threaded applications, when single threaded applications would be more desirable.

Timer services provides the capability to have thousands of process local timers driven from a single O/S timer. This reduces timer overhead for timer intensive applications.

Threads and Thread pools are lightweight classes to cover these constructs. The anticipation is that C++09x will have standard support for these features, and that the library will be adapted to work with the standard based classes at this time.

Iostreams logging allows a single interface to any logging subsystem (current adapters for syslogd and slogger). It also has a manipulator that is the answer to the question "I love iostreams, but I sure miss printf style formatting for numbers, can anything be done?". Hint, the answer is yes...

How do I get it?#

QFC will check out into a Momentics IDE 4.01 directly from the source repository; however, since Momentics does not ship with a subversion plug-in pre-installed, you'll need to download one. I happen to like subversive (http://www.polarion.org/projects/subversive/download/1.1/update-site/) myself.

Get Subversion plug-in#

If you are familiar with Eclipse, then you'll recognize this URL as an update site; if you are new to Eclipse, then these instructions should get you set up with subversive.

Connect to the source repository#

Once you have a subversion plug-in installed, you need to connect to the source repository and check out the source. If you feel you know your way around subversion then the only think you need to know is the source repository URL (http://community.qnx.com/svn/repos/qfc), otherwise you can follow these instructions

Check-out and build the source#

In order to check-out and build QFC, you may find the following instructions useful.

Options#

Get the Eclipse Doxygen plug-in#

You can get Eclox (EClipse dOXygen) from its update site at: http://download.gna.org/eclox/update

QFC is completely self documented using doxygen and Eclox, so you may find this a handy tool to have!

Licensing#

Source modules in the QFC library are distributed under one of three licenses.

1. Version 2.0 of the apache license Apache License

2. The following "attribution" license:

Copyright (c) 2003 Eugene Gladyshev

Permission to copy, use, modify, sell and distribute this software
is granted provided this copyright notice appears in all copies.
This software is provided "as is" without express or implied
warranty, and with no claim as to its suitability for any purpose.

3. A public domain license (no restrictions)

Resources#

Not included in the main library (due to a restrictive license), but available for download is a derivative of a co-routine class, that has been adapted to allow classes developed with the QFC library as threaded classes, to be re-hosted as co-routines. Experiments in hosting threaded classes as co-routines during development showed that they can be used to isolate concurrency issues, since executing as a co-routine eliminates these issues, without any substantial code changes.

Planning#

The framework will be re-engineered for C++09x to integrate the standardized threading, and concepts extensions. Following this, support for compile time, sequence type safety for channels will be investigated. Until then the plan is simply to add more useful QNX specific classes, and to remove bugs...

Currently under development is integration of Reflex. Once Reflex is fully integrated, then a new plugin framework will be released which is implemented using transparent dynamic class loading (i.e. a plug-in subscriber application need only attempt to instantiate a class, and if it is not loaded, it will automatically be loaded from a plug-in dll as determined by the run-time environment). The plug-in framework will scale from a effective zero overhead model requiring the application architect to define the interfaces that plug-ins will supply (and for plug-in provider writers to meet those interfaces), to a higher overhead, but an extremely loose interface definition based on reflection (i.e. the plug-in provider writer simply needs to provide some interface, in some class; somewhere in the dll that has an interface that is usable by the plug-in subscriber). In the reflection model, the plug-in subscriber can receive a detailed account of all the recognized interfaces, and can thus enable/disable or provide default features dynamically).

An example application using reflective plug-ins will be the "class" file system. This will consist of a plug-in subscriber that registers a root prefix name space, and then instantiates a class with no interface (other than a constructor) which reflects on it's own dll, and then (based on what it finds in there) populates the name space (conforming methods found in the dll override the default resmgr methods). This completely relieves the developer of any knowledge whatsoever about how resource managers work, and essentially enables one to simply create a bunch of classes and throw them into the "class" file system and let it figure out what to do with them (hint the classes will show up as directory entires, and descending down them each of the classes of which that class is composed will have an entry - if a class has a conforming method - say one that matches the Read signature - then that class can be "read" and will return whatever the provider developer decided it would). Should the provider wish to create a "file system" of class instantiations, a constructor of a class will be able to instantiate n instances of a class and invoke the reflector on each instance (thus making them children entries of the calling class). In order to provide such a "root class" the only requirement is that it have a constructor that takes a reflector as an argument (so that reflection can be dynamically applied as instantiation occurs). An example plug-in provider dll that implements an ftp file system will be provided (and it will be shockingly small).