QNX Neutrino Resource Managers: Using MsgSend() and MsgReply()

Authored by: **Chris McKillop**
Updated by: **Thomas Fletcher**

This article will show, by example, how to write a QNX Neutrino resource manager skeleton.

If you were previously developing using QNX 4, then this article you will get a comfortable feeling of still being able to use a single numeric ID (like a process ID) as the basis for establishing communication.

If you are a first time user, then welcome aboard!

The skeleton sample code is also a good base to use for their expansion into other areas (beyond what QNX 4 was capable of) when they have the opportunity.

Note: If you haven't read the QNX Neutrino (Neutrino) System Architecture guide, you should do so before going on, it has some good stuff!

IPC under the two OSs

Since many of the people reading this will not have written a line of code for QNX 4, they may need to be brought up to speed with the differences between IPC (inter-process communication) on QNX Neutrino and QNX 4. First, both OSs use a notion of Send/Receive/Reply for messaging. This IPC mechanism is (generally) used in a synchronous manner - the sending process waits for a reply from the receiver and a receiver waits for a message to be sent. This provides a very easy call-response synchronization.

QNX 4 approach

Under QNX 4, the *Send*() function call needed only the process ID of the receiving process. QNX 4 also provided a very simple API for giving a process name and, in turn, looking up that name to get a process ID. So you could name your server process, and then your client process could look up that name, get a process ID (**pid**), and then Send the server data and wait for a reply. This model worked well in a non-threaded environment.

QNX Neutrino approach

Since QNX Neutrino included proper thread support, the notion of having a single conduit into a process didn't make a lot of sense, so a more flexible system was designed. To perform a **MsgSend**() under QNX Neutrino, you no longer need a **pid**, but rather a connection ID (**coid**). This **coid** is obtained from opening a connection to a channel. Processes can now create multiple channels and can have different threads service any (or all) of them. The issue now becomes: How does a client get a **coid** in the first place so it can open a connection to get the **coid** it needs to perform the **MsgSend**()?

There are many different ways this kind of information sharing can occur, but the method that falls in line with the QNX Neutrino design ideals is for the server to also be a *Resource Manager*.

A Resource Manager is a term for a process that handles resources in the filesystem. The kernel (**procnto**) is itself a Resource Manager - **/dev/null**, **/proc**, and several other resources are handled by **procnto** in the same way any other process handle them. Under QNX Neutrino, and other POSIX systems, when you call **open**() you get back a file descriptor (**fd**). But this **fd** is also a **coid**!!! So instead of registering a name, as in QNX 4, your server process registers *a path in the filesystem* and the client opens that path to get the **coid** to talk to the server.

A client/server example

The best way to learn is by example. We'll now go over a simple client/server that can be used as the starting point for any similar project. There are two source files - I recommend that you right-click on each one and open them in a separate browser windows. This way you can keep the code in one window while you read this article in another.

server.c
client.c

The two source files are easily built by invoking:

```
qcc -o client client.c
qcc -o server server.c
```

from the command line. Note that you must run **server** as **root** - a requirement in order to use the **resmgr_attach**() function.

The server

Let's begin with the server. If you've had a chance to read over the chapter on “Writing a Resource Manager” in the *Programmer's Guide*, you'll see some similarities between this example and that one, but also some differences. If you haven't read this guide, I recommend that you do.

The first action the server will take is to create a dispatch handle (**dpp**) using **dispatch_create**(). This handle will be used later when making other calls into the dispatch portion of the library. This is important - the bottom layer of a resource manager is the dispatch layer. This layer takes care of receiving incoming messages and routing them to the right layer above (**resmgr**, message, pulse).

After the dispatch handle is created, the server sets up the variables needed to make a call into **resmgr_attach***()*. But since we're not using the **resmgr** functionality for anything more then getting a connection ID to use with **MsgSend**(), the server sets up everything to defaults. We don't need (or want) to worry about I/O and connection messages right now (like the messages that **open**/**close**/**read**/**write**/... generate); we just want them to work and do the right thing. Luckily, there are defaults built into the C library to handle these types of messages for you, and **iofunc_func_init***()* sets up these defaults. The call to **iofunc_attr_init** sets up the attribute structure so that the entry in the filesystem has the specified attributes.

Finally, the call to **resmgr_attach**() is made. For our purposes, the most important parameter is the third. In this case we're registering the filesystem entry **serv**. Since an absolute path wasn't given, the entry will appear in the same directory where the server was run. All of this will give us

a filesystem entry that can be **open**()'d and **close**()'d, but generally behaves the same as */dev/null*. But that's fine, since we want to be able to **MsgSend**() data to our server, not **write**() data to the server.

Now that the **resmgr** portion of the setup is complete, we need to tell the dispatch layer that we'll be handling our own messages in addition to the standard I/O and connection messages handled by the **resmgr** layer. In order for the dispatch layer to know the general attributes of the messages we'll be receiving, the **message_attr** structure is filled with information. In this case we're telling it that the number of message parts we're going to receive is 1 with a max message size of 4096 bytes.

Once we have these attributes defined, we can register our intent to handle messages with the dispatch layer by invoking **message_attach**(). With this call we're setting up **message_callback** to be the handler of messages of type **_IO_MAX + 1** up to and including messages of **_IO_MAX + 2**. There's even the option of having a pointer to arbitrary data passed into the callback, but we don't need that so we're setting it to NULL.

Some people might now be asking, "**Message type _IO_MAX + 1**?!? I don't see anything in the **MsgSend**() docs for setting a message type!". This is true. However, in order to play nice with the dispatch later, all incoming messages should have a 32-bit integer at the start of the message indicating the message type. Although this may seem restrictive to a new QNX developer, the reason it's in place is that most designs will end up using some sort of message identification anyway, and this just forces you into a particular style. This will become clearer when we look at the client. But now let's finish the server.

Now that we've registered both the **resmgr** and message handlers with the dispatch layer, we simply create a context for the dispatch layer to use while processing messages by calling **dispatch_context_alloc** and then start receiving and processing data. This is a two-step process:

1. The server calls **dispatch_block**(), which will wait for incoming messages and pulses.
2. Once there's data available, we call into **dispatch_handler**() to do the right thing based on the message data. It's inside the **dispatch_handler** call that our **message_callback** will be invoked, when messages of the proper type are received.

Finally, let's look at what the **message_callback** actually does when a proper message is received. When a message of type **_IO_MAX + 1** or **_IO_MAX + 2** is received, our callback will be invoked. We get the message *type* passed in via the type parameter. The actual message data can be found in **ctp->msg**. When the message comes in, the server will print out the message type and the string that was sent from the client. It then prints out the offset from **_IO_MAX** of the message type and then finally formats a reply string and sends the reply back to the client via **ctp->rcvid** using **MsgReply**(). The server walk-through is now complete!

The client

The client is much simpler. It uses the **open**() function to get a **coid** (the server's default **resmgr** setup takes care of all of this on the server side), and performs a **MsgSend**() to the server based on this **coid** and waits for the reply. When the reply comes back, the client prints out the reply data. The client can be optionally given the command-line option **-n#** (where # is the offset from **_IO_MAX**) to use for the message. If you give anything over 2 as the offset, the **MsgSend**() will fail, since the server hasn't set up handlers for those messages! Neat, eh? Also, remember that since the server registers a relative pathname, the client must be run from the same directory as the server.

These examples are very basic, but still cover a lot of ground. There are many other things you can do using this same basic framework:

- Register different **message_callbacks** based on different message types.
- Register to receive pulses in addition to messages using **pulse_attach**().
- Override the default I/O message handlers so that clients can also use **read**() and **write**() to interact with your server.
- Use thread-pools to make your server multi-threaded.

Many of these topics are covered in the Neutrino Programmer's Guide in the section on Writing a Resource Manager which is a great place to expand your knowledge using this client/server as a sample to expand on.