To Resource Manager or Not to Resource Manager

Authored by: **Mario Charest**
Updated by: **Thomas Fletcher**

This article assumes you have a general idea of what a resource manager is. But by way of a quick summary, let's define a resource manager as a program that provides an interface through the pathname space, using the standard I/O functions **read**(), **write**(), **devctl**(), **select**(), etc.

In this article, I'll be referring to two different operating systems: QNX 4 and QNX Neutrino.

QNX 4 I/O managers

Under QNX 4, the equivalent of a Neutrino resource manager is called an I/O manager, which can be somewhat cumbersome to use. There was very little documentation available on I/O managers; only in the last one to two years has a decent template/library been made available to help developers write their own I/O managers. Although this is an improvement, the template/library was provided only as a convenience and comes with no guaranty. Many developers have never written an I/O manager under QNX 4, simply because they felt it was too complex to do so.

For these reasons, the concept of a resource manager isn't well known to the QNX 4 community, and even less known to people who are new to Neutrino. I've had many discussions with fellow programmers about the benefits and drawbacks of resource managers. Most of these programmers have QNX 4 backgrounds and felt uneasy with the concept. It was difficult for them to create a design that can unleash the power of the resource manager.

Personally, I perceive the jump from QNX 4's Send()/Receive()/Reply() model to Neutrino's resource managers to be like moving from C to C++. It's not only the language that differs, but how you solve problems. If you have a QNX 4 background and have never used an I/O manager, you'll need to change the way you think. The rest of this article describes my own experience with resource managers and how I've solved problems with them. I myself have converted from the QNX 4 I/O manager mindset to the Neutrino resource manager concept. It wasn't easy, but boy am I glad I did!

Three examples

The text below describes three different resource managers I have written or helped design for various projects. We'll start with the simplest example and move on to more complex examples.

For those who aren't familiar with resource managers, you might want to read the documentation for a better understanding before you continue. (See: "Writing a resource manager" in the *Programmer's Guide*.)

GPS example

Without going into the fine details of a GPS device, in general the unit sends a stream of data every second. The stream is composed of information organized in command groups. Here's an example of the output from a GPS:

```
$GPGSA,A,3,17,16,22,31,03,18,25,,,,,,1.6,1.0,1.2*39
$GPGGA,185030.30,4532.8959,N,07344.2298,W,1,07,1.0,23.8,M,-32.0,M,,*69
$GPGLL,4532.8959,N,07344.2298,W,185030.30,A*12
$GPRMC,185030.00,A,4532.8959,N,07344.2298,W,0.9,116.9,160198,,*27
```

```
$GPVTG,116.9,T,,,0.9,N,1.7,K*2D
$GPZDA,185030.30,16,01,1998,,*65
$GPGSV,2,1,08,03,55,142,50,22,51,059,51,18,48,284,53,31,23,187,52*78
```

Each line corresponds to a data set. Here's the C structure of some of the data sets:

```
typedef struct GPSRMC_s {
double UTC;
int Status;
Degree_t Latitude;
NORTHSOUTH Northing;
Degree_t Longitude;
EASTWEST Easting;
float Speed;
float Heading;
} GPSRMC_t;
typedef struct GPSVTG_s {
float Heading;
float SpeedInKnots;
float SpeedInKMH;
} GPSVTG_t;
typedef struct GPSUTM_s {
UTM_t X;
UTM_t Y;
} GPSUTM_t;
```

When I first designed the driver for QNX 4, I chose to provide one API per GPS format command: **gps_get_rmc**(), **gps_get_vtg**(), **get_get_utm**(), etc. Internally, each function would **Send**() a message with a different command and the reply was the data last received by the GPS.

Finally, it came time to port the driver to QNX Neutrino. Although I could have kept the same architecture used in the QNX 4 design, I felt it was time to overcome my apprehensions about resource managers. I took a deep breath and decided to dive in.

The first obstacle was that **read**() and **write**() are half-duplex operations -- they can't be used to send a command and get data back. Actually, I guess one could do this:

```
GPSUTM_t utm;
Int cmd = GPS_GET_GSA;
fd = open( "/dev/gps1", O_RDWR );
write( fd, &cmd, sizeof( cmd) );
read( fd, &data, sizeof(data) );
close(fd);
```

But this code looks unnatural. Nobody would expect **read**() and **write**() to be used in that way, so this approach was a no go. I thought about using **devctl**() to send a command and request specific data, but my gut feeling was telling me it wasn't the "resource managerish" way of doing it. Then it hit me. Why not have a different path for every command set? The driver would create the following pathnames:

```
/dev/gps1/gsa.txt
/dev/gps1/gsa.bin
```

```
/dev/gps1/gga.bin
/dev/gps1/gga.txt
So a program wanting to get GSA information would do this:
gps_gsa_t gsa;
int fd;
fd = open ( "/dev/gps1/gsa.bin", O_RDONLY );
read( fd, &gsa, sizeof( gsa ) );
close ( fd);
```

Now, I don't know about you, but this looks darn clean to me! The logic behind the .txt and .bin extensions is that data returned by the **read**() would be in ASCII format if the *.txt* file is used. If the *.bin* file is used instead, the data is returned in a C structure for easier use. But why support **.txt* if most programs would prefer to use the binary representation? The reason is simple.

From the shell you could type:

```
# cat /dev/gps1/rmc.txt
```

and you would see:

```
# GPRMC,185030.00,A,4532.8959,N,07344.2298,W,0.9,116.9,160198,,*27
```

Can you feel the power?! You now have access to the GPS data from the shell. Wait, there's more! You want to know all the commands the GPS supports.

From the shell, type:

```
# ls /dev/gps1
```

Or from a C program use **opendir**()/**readdir**(). You also want your program to be informed when there's new data available. Simply use **select**() or **ionotify**() instead of polling on the data. Contrary to what you might think, it's quite simple to support these features from within the resource manager.

Database example

This particular design asked for a program to centralize file access. Instead of having each program handle the specific location of the data files (on the hard disk or in FLASH or RAM), we decided that one program would handle it. Furthermore, file updates done by one program required that all programs using that file be notified. So instead of having each program notify each other, only one program would take care of the notification. That program was cleverly named "database". ;-)

The original design included the following API: **db_read**(), **db_write**(), **db_update_notification**(), etc. When I was brought into the project, it became obvious to me this was a nice fit for a resource manager. The API was changed to **open**(), **read**(), **write**(), **select**(), **ionotify**(), etc. Sound familiar? ;-)

Client programs were seeing one path only:

```
/dbase/filename
```

Of course, much like the QNX Neutrino Package Filesystem manager, files found in /dbase/" were not physically there. Files could be scattered all over the place.

The database resource manager program looked at the filename during **open**() and decided where the file needed to go or to read from. This, of course, depended on the specific field in the filename. For example, if the file had a *.tmp* extension, it would go to the RAM disk.

The real beauty of this design is that the designer of the client program could test their application without having the database program running. The **open**() would be handled directly by the filesystem.

To support notification of a file change, a client would use **ionotifiy**() or **select**() on the file descriptor of the files. Unfortunately, that feature isn't supported natively by the filesystem.

I2C example

I am currently writing a driver for an I2C bus controller of the PowerPC MPC860 (lovely CPU, by the way). The I2C bus is simply a 2-wire bus. It's extremely cheap to implement, hardware-wise. The bus supports up to 127 devices on the bus and each device can handle 256 commands. When devices want to read or write information to or from another device, they must first be set up as a master to own the bus. Then the device sends out the device address and command register number. The slave then acts upon the command received.

Now that you know everything there is to know about I2C bus, let me take you through my design phases. I want to cover this example last because it's the one I found the most exciting.

At first, I thought a resource manager wouldn't apply in this case. All **read**() and **write**() operations require a device address and command register. I thought about using **devctl**(), but I'm allergic to **devctl**()! I can't say exactly why, but their usage looks ugly to me.

Since I couldn't find a way to not use **devctl**, I was sticking with sending custom messages and hiding the **MsgSend**() in an API. I presented the API to the customer and they agreed on it.

Then at 7:45am the next morning, I stubbed my little toe on a chair leg. That must have stimulated some part of my brain, because during breakfast a solution popped into my head. (Isn't it weird that one minute you're thinking about throwing a chair out the window and the next you have a solution to a problem you thought you already solved?)

Here's what I came up with:

Each device would live under its own directory and each register would have a filename.

```
/dev/i2c1//
```

Actually, I had already thought about that before the toe accident, but had rejected it. What was new this time is the idea that each filename (127 devices * 256 registers = 32512 filenames) doesn't really need to exist. Each device would be created live, as it's required. Therefore, each **open**() is actually an O_CREATE.

To prevent any problems caused by a high number of possibly existing files, I decided an **ls** command of the /dev/i2c1 would return nothing! Slightly unintuitive I know, but I think it's worth the benefit.

Then another idea came to me. I decided to add an option allowing filenames that were open at least once to be made visible if **ls** is used. At this point, it's important to clarify that the existence of the filenames is totally handled by the resource manager. The OS itself is not used in that process. So it's not a problem if filenames respond to open requests but not to **ls**.

One element was left to solve: the I2C bus has a concept of baud rate. Guess what? There are already C functions to set up baud rates. And I could make it work via the **stty** command from the shell. Very cool!

At this stage I was happy with the new design. Anybody using the driver doesn't have to worry about libraries or include files because there are none! The resource manager, at no cost, allows each command register to be accessed via the shell, or for that matter, though SAMBA from a Linux or Windows machine. Access via the shell makes debugging so easy -- there's no need to write custom test tools, it's unbelievably flexible, not to mention the support for separate permissions for each and every command register of every device!

Now I'm all set. I'm ready to send the new spec to the customer, when BANG! -- another set of neurons fired. The following code lacks clarity in my opinion:

```
fd = open ( "/dev/i2c1/34/45" );
read( fd, &variable, sizeof( variable ) );
close(fd);
```

Would it be much better to have this instead?:

```
fd = open ( "/dev/i2c1/flash/page_number" );
read( fd, &page_number, sizeof( page_number ) );
close (fd );
```

Of course, one could use **#define** to show meaningful information, but that wasn't satisfactory. To solve this problem, I created a configuration file that the driver could read to create an alias. The configuration file looked like this:

```
[4=temperature_sensor]
10=max
11=min
12=temperature
13=alarm
[5=flash]
211=page_number
```

The field inside **[ ]** defines the device address and name. The data that follows specifies each register of that device. I see two main advantages to this approach:

- The configuration file's format helps document the program.
- If the hardware is changed and devices are assigned new addresses, you simply change the configuration file -- there's no program to recompile. That you got to love!

These predefined devices would always show via the **ls** command. Time to give myself a pat on the back. ;-)

When not to resource manager

There are times when a resource manager isn't required.

The most obvious case would be if a program doesn't need to receive messages. But still, if your program is event-driven, you might want to look at using the dispatch library to handle internal events. In the future, if your program ever needs to receive messages, you can easily turn it into a resource manager.

Another case might be if your program interfaces only with its children. The parent has access to all the children's information required to interface with them.

In any case, I think I'm biased; I'd actually make every attempt to turn everything into a resource manager.

I really like it when I can get my resource manager client to use only the POSIX API. That means less documentation to write, not to mention very portable code. Of course, in many cases, providing an API on top of the POSIX API is often very desirable. You could hide the details of **devctl**() or custom messages. The internals of the API could then be changed if you have to port to a different OS.

If you must transfer high volumes of data at a very high rate, a resource manager can be a problem. Since resource managers basically interface via IPC kernel primitives, they're slower then **memcpy**(). ;-) But nothing prevents you from using a mix of shared memory, POSIX API, and custom messages, all to be hidden in your own API. It wouldn't be too difficult to have the API detect whether or not the resource manager is local and to use shared memory when local and IPC when remote -- a way to get the best of both worlds.

Conclusion

Hopefully, this document will help you get a grasp of some of the power of resource managers. They may be unsettling at first, but in my honest opinion, they're definitely worth the learning curve.

Please feel free to contact me about your experience with resource managers. I'm curious to see how creative people can get with their own resource managers. There are plenty of possibilities; I only scratched the surface.