QNX® Neutrino® RTOS

Writing a Resource Manager

BlackBerry | QNX

QNX Software Systems Limited
1001 Farrar Road
Ottawa, Ontario
K2K 0B3
Canada

Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: info@qnx.com
Web: http://www.qnx.com/

Electronic edition published: March 06, 2020

Contents

About This Guide

This guide will help you create a *resource manager*, a process that registers a name in the filesystem name space, which other processes then use to communicate with the resource manager.

The following table may help you find information quickly:

For information about:	Go to:
What a resource manager is, and when you would—and wouldn't—use one	*What Is a Resource Manager?*
The overall structure of a resource manager	*The Bones of a Resource Manager*
Adding some "meat" to the basic structure	*Fleshing Out the Skeleton*
Data structures that the POSIX-layer routines use	*POSIX-Layer Data Structures*
Reading and writing data	*Handling Read and Write Messages*
Atomic operations	*Combine Messages*
Adding your own data to resource-manager structures	*Extending the POSIX-Layer Data Structures*
Handling other types of messages	*Handling Other Messages*
Unblocking clients because of signals, timeouts, and closed file descriptors, and handling interrupts	*Unblocking Clients and Handling Interrupts*
Handling more than one message at once	*Multithreaded Resource Managers*
Taking over a directory	*Filesystem Resource Managers*
Terms used in QNX Neutrino docs	*Glossary*

For another perspective on resource managers, see the Resource Managers chapter of *Getting Started with QNX Neutrino*. In particular, this chapter includes a summary of the handlers for the connect and I/O messages that a resource manager will receive; see "Alphabetical listing of connect and I/O functions" in it.

This book assumes that you're familiar with message passing. If you're not, see the Interprocess Communication (IPC) chapter in the *System Architecture* guide as well as the *MsgSend()*, *MsgReceivev()*, and *MsgReply()* series of calls in the QNX Neutrino *C Library Reference*.

For information about programming in the QNX Neutrino RTOS, see *Getting Started with QNX Neutrino* and the QNX Neutrino *Programmer's Guide*.

Typographical conventions

Throughout this manual, we use certain typographical conventions to distinguish technical terms. In general, the conventions we use conform to those found in IEEE POSIX publications.

The following table summarizes our conventions:

Reference	Example
Code examples	`if( stream == NULL)`
Command options	`-lR`
Commands	`make`
Constants	NULL
Data types	`unsigned short`
Environment variables	*PATH*
File and pathnames	**/dev/null**
Function names	*exit()*
Keyboard chords	**Ctrl–Alt–Delete**
Keyboard input	`Username`
Keyboard keys	**Enter**
Program output	`login:`
Variable names	*stdin*
Parameters	*parm1*
User-interface components	**Navigator**
Window title	**Options**

We use an arrow in directions for accessing menu items, like this:

You'll find the Other... menu item under **Perspective → Show View**.

We use notes, cautions, and warnings to highlight important messages:

Notes point out something important or useful.

CAUTION: Cautions tell you about commands or procedures that may have unwanted or undesirable side effects.

DANGER: Warnings tell you about commands or procedures that could be dangerous to your files, your hardware, or even yourself.

Note to Windows users

In our documentation, we typically use a forward slash (/) as a delimiter in pathnames, including those pointing to Windows files. We also generally follow POSIX/UNIX filesystem conventions.

Technical support

Technical assistance is available for all supported products.

To obtain technical support for any QNX product, visit the Support area on our website (*www.qnx.com*). You'll find a wide range of support options, including community forums.

Chapter 1
What Is a Resource Manager?

In general terms, a *resource manager* is a process that registers a name in the filesystem name space. Other processes use that path to communicate with the resource manager.

To give the QNX Neutrino RTOS a great degree of flexibility, to minimize the runtime memory requirements of the final system, and to cope with the wide variety of devices that may be found in a custom embedded system, the OS allows user-written processes to act as resource managers that can be started and stopped dynamically.

Resource managers are typically responsible for presenting an interface to various types of devices. This may involve managing actual hardware devices (like serial ports, parallel ports, network cards, and disk drives) or virtual devices (like **/dev/null**, a network filesystem, and pseudo-ttys).

In other operating systems, this functionality is traditionally associated with *device drivers*. But unlike device drivers, resource managers don't require any special arrangements with the kernel. In fact, a resource manager runs as a process that's *separate from the kernel* and looks just like any other user-level program.

The kernel (`procnto`) is itself a resource manager; it handles **/dev/null**, **/proc**, and several other resources in the same way any other process handles them.

A resource manager accepts messages from other programs and, optionally, communicates with hardware. It registers a pathname prefix in the pathname space (e.g., **/dev/ser1**), and other processes can open that name using the standard C library *open()* function, and then *read()* from, and *write()* to, the resulting file descriptor. When this happens, the resource manager receives an open request, followed by read and write requests.

A resource manager isn't restricted to handling just *open()*, *read()*, and *write()* calls; it can support any functions that are based on a file descriptor or file pointer, as well as other forms of IPC.

Adding resource managers in QNX Neutrino won't affect any other part of the OS—the drivers are developed and debugged like any other application. And since the resource managers are in their own protected address space, a bug in a device driver won't cause the entire OS to shut down.

If you've written device drivers in most UNIX variants, you're used to being restricted in what you can do within a device driver; but since a device driver in QNX Neutrino is just a regular process, you aren't restricted in what you can do (except for the restrictions that exist inside an ISR).

In order to register a prefix in the pathname space, a resource manager must have the PROCMGR_AID_PATHSPACE ability enabled. In order to create a public channel (i.e., *without* _NTO_CHF_PRIVATE set), your process must have the PROCMGR_AID_PUBLIC_CHANNEL ability enabled. For more information, see *procmgr_ability()* in the QNX Neutrino *C Library Reference*.

For example, a serial port may be managed by a resource manager called `devc-ser8250`, although the actual resource may be called **/dev/ser1** in the pathname space. When a process requests serial port services, it does so by opening a serial port (in this case **/dev/ser1**).

```
fd = open("/dev/ser1", O_RDWR);
for (packet = 0; packet < npackets; packet++)
{
    write(fd, packets[packet], PACKET_SIZE);
}
close(fd);
```

Because resource managers execute as processes, their use isn't restricted to device drivers; any server can be written as a resource manager. For example, a server that's given DVD files to display in a GUI interface wouldn't be classified as a driver, yet it could be written as a resource manager. It can register the name **/dev/dvd** and as a result, clients can do the following:

```
fd = open("/dev/dvd", O_WRONLY);
while (data = get_dvd_data(handle, &nbytes))
{
    bytes_written = write(fd, data, nbytes);
    if (bytes_written != nbytes)
    {
        perror ("Error writing the DVD data");
    }
}
close(fd);
```

Why write a resource manager?

Here are a few reasons why you'd want to write a resource manager:

- The client API is POSIX.

  The API for communicating with the resource manager is, for the most part, POSIX. All C programmers are familiar with the *open()*, *read()*, and *write()* functions. Training costs are minimized, and so is the need to document the interface to your server.

- You can reduce the number of interface types.

  If you have many server processes, writing each server as a resource manager keeps the number of different interfaces that clients need to use to a minimum.

  An example of this is if you have a team of programmers building your overall application, and each programmer is writing one or more servers for that application. These programmers may work directly for your company, or they may belong to partner companies who are developing add-on hardware for your modular platform.

  If the servers are resource managers, then the interface to all of those servers is the POSIX functions: *open()*, *read()*, *write()*, and whatever else makes sense. For control-type messages that don't fit into a read/write model, there's *devctl()* (although *devctl()* isn't POSIX).

- Command-line utilities can communicate with resource managers.

  Since the API for communicating with a resource manager is the POSIX set of functions, and since standard POSIX utilities use this API, the utilities can be used for communicating with the resource managers.

The types of resource managers

There are two types of resource managers:

- device resource managers
- filesystem resource managers

The type you use depends on what you want the resource manager to do, as well as on the amount of work you want to do yourself in order to present a proper POSIX filesystem to the client.

Device resource managers

Device resource managers create only single-file entries in the filesystem, each of which is registered with the process manager. Each name usually represents a single device. These resource managers typically rely on the resource-manager library to do most of the work in presenting a POSIX device to the user.

For example, a serial port driver registers names such as `/dev/ser1` and `/dev/ser2`. When the user does `ls -l /dev`, the library does the necessary handling to respond to the resulting _IO_STAT messages with the proper information. The person who writes the serial port driver is able to concentrate instead on the details of managing the serial port hardware.

Filesystem resource managers

Filesystem resource managers register a *mountpoint* with the process manager. A mountpoint is the portion of the path that's registered with the process manager. The remaining parts of the path are managed by the filesystem resource manager. For example, when a filesystem resource manager attaches a mountpoint at **/mount**, and the path **/mount/home/thomasf** is examined:

/mount/

Identifies the mountpoint that's managed by the process manager.

home/thomasf

Identifies the remaining part that's to be managed by the filesystem resource manager.

Examples of using filesystem resource managers are:

- flash filesystem drivers (although the source code for flash drivers takes care of these details)
- a `tar` filesystem process that presents the contents of a `tar` file as a filesystem that the user can `cd` into and `ls` from
- a mailbox-management process that registers the name **/mailboxes** and manages individual mailboxes that look like directories, and files that contain the actual messages

Communication via native IPC

Once a resource manager has established its pathname prefix, it will receive messages whenever any client program tries to do an *open()*, *read()*, *write()*, etc. on that pathname. For example, after `devc-ser*` has taken over the pathname **/dev/ser1**, and a client program executes:

```
fd = open ("/dev/ser1", O_RDONLY);
```

the client's C library will construct an _IO_CONNECT message, which it then sends to the `devc-ser*` resource manager via IPC.

Some time later, when the client program executes:

```
read (fd, buf, BUFSIZ);
```

the client's C library constructs an _IO_READ message, which is then sent to the resource manager.

A key point is that *all* communications between the client program and the resource manager are done through *native IPC messaging*. This allows for a number of unique features:

- A well-defined interface to application programs. In a development environment, this allows a very clean division of labor for the implementation of the client side and the resource manager side.
- A simple interface to the resource manager. Since all interactions with the resource manager go through native IPC, and there are no special “back door” hooks or arrangements with the OS, the writer of a resource manager can focus on the task at hand, rather than worry about all the special considerations needed in other operating systems.
- Free network transparency. Since the underlying native IPC messaging mechanism is inherently network-distributed without any additional effort required by the client or server (resource manager), programs can seamlessly access resources on other nodes in the network without even being aware that they're going over a network.

All QNX Neutrino RTOS device drivers and filesystems are implemented as resource managers. This means that everything that a “native” QNX Neutrino device driver or filesystem can do, a user-written resource manager can do as well.

Consider FTP filesystems, for instance. Here a resource manager would take over a portion of the pathname space (e.g., **/ftp**) and allow users to `cd` into FTP sites to get files. For example, `cd /ftp/rtfm.mit.edu/pub` would connect to the FTP site **rtfm.mit.edu** and change directory to **/pub**. After that point, the user could open, edit, or copy files.

Application-specific filesystems would be another example of a user-written resource manager. Given an application that makes extensive use of disk-based files, a custom tailored filesystem can be written that works with that application and delivers superior performance.

The possibilities for custom resource managers are limited only by the application developer's imagination.

Examples of resource managers

Before getting into the workings of a resource manager, let's consider some actual and possible uses.

Transparent Distributed Processing (Qnet) statistics

For instance, Transparent Distributed Processing (Qnet) — part of the `io-pkt` core networking stack — contains resource-manager code that registers the name **/proc/qnetstats**. If you open this name and read from it, the resource manager code responds with a body of text that describes the statistics for Qnet.

The `cat` utility takes the name of a file and opens the file, reads from it, and displays whatever it reads to standard output (typically the screen). As a result, you can type:

```
cat /proc/qnetstats
```

The Qnet resource manager code responds with text such as:

```
kif net_server               :         0,3
kif waiting                  :         1,2
kif net_client               :         0,1
kif buffer                   :         0,1
kif outbound_msgs            :         0,1
kif vtid                     :         0,1
kif server_msgs              :         0,1
kif nd_down                  :        42
kif nd_up                    :       132
kif nd_changed               :         3
kif send_acks                :         0
kif client_kercalls          :        14
kif server_msgs              :    202898
kif server_unblock           :         0
qos tx_begin_errors          :         0
qos tx_done_errors           :         0
qos tx_throttled             :         0
qos tx_failed                :         8
qos pkts_rxd_noL4            :         0
qos tx_conn_created          :        43
qos tx_conn_deleted          :        41
qos rx_conn_created          :        35
qos rx_conn_deleted          :        33
qos rx_seq_order             :         0
```

Robot arm

You could also use command-line utilities for a robot-arm driver. The driver could register the name, **/dev/robot/arm/angle**, and any writes to this device are interpreted as the angle to set the robot arm to. To test the driver from the command line, you'd type:

```
echo 87 >/dev/robot/arm/angle
```

The `echo` utility opens **/dev/robot/arm/angle** and writes the string ("87") to it. The driver handles the write by setting the robot arm to 87 degrees. Note that this was accomplished without writing a special tester program.

Another example would be names such as **/dev/robot/registers/r1**, **r2**, ... Reading from these names returns the contents of the corresponding registers; writing to these names set the corresponding registers to the given values.

Even if all of your other IPC is done via some non-POSIX API, it's still worth having one thread written as a resource manager for responding to reads and writes for doing things as shown above.

GPS devices

In general, a GPS device 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;
```

You could provide one API per GPS format command: *gps_get_rmc()*, *gps_get_vtg()*, *get_get_utm()*, and so on. Internally, each function would send a message with a different command, and the reply was the data last received by the GPS.

The first obstacle was that *read()* and *write()* are half-duplex operations; you can't use them to send a command and get data back. You 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. You could use *devctl()* to send a command and request specific data, but if you implement the driver as a resource manager, you can 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**

and 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);
```

The benefit of having both the **.txt** and **.bin** extensions is that data returned by the *read()* would be in ASCII format if you use the **.txt** file. If you use the **.bin** file 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'd see:

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

You now have access to the GPS data from the shell. But you can do even more:

- If you want to know all the commands the GPS supports, simply type:

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

- To do the same from a C program, use *opendir()* and *readdir()*.
- If you 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), the developers 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 API in the original design included *db_read()*, *db_write()*, *db_update_notification()*, etc., but a resource manager is an even better fit. The API was changed to the familiar *open()*, *read()*, *write()*, *select()*, *ionotify()* and so on.

Client programs were seeing only one path, **/dbase/filename**, but the files found in **/dbase/** weren't physically there; they 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 designers 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 *ionotify()* or *select()* on the file descriptor of the files. Unfortunately, that feature isn't supported natively by the filesystem, so the database program needs to be running in order for you to test the operation.

I2C (Inter-Integrated Circuit) driver

This example is a driver for an I2C bus controller. The I2C bus is simply a 2-wire bus, and the hardware is extremely cheap to implement. 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.

You think a resource manager wouldn't apply in this case. All *read()* and *write()* operations require a device address and command register. You could use *devctl()*, but a resource manager again provides a cleaner interface.

With a resource manager, each device would live under its own directory, and each register would have a filename:

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

An important thing to note is 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_CREAT.

To prevent any problems caused by a high number of possibly existing files, you could make an `ls` command of the **/dev/i2c1** directory return nothing. Alternatively, you could make `ls` list the filenames that have been opened at least once. At this point, it's important to clarify that the existence of the filenames is totally handled by the resource manager; the OS itself isn't used in that process. So it isn't a problem if filenames respond to *open()* requests, but not to `ls`.

One element is left to solve: the I2C bus has a concept of a baud rate. There are already C functions to set up baud rates, and you can make it work via the `stty` command from the shell.

People using the driver don'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, and it's unbelievably flexible, not to mention the support for separate permissions for each and every command register of every device.

One more thing: this code could be clearer:

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

It would 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, you could use `#define` directives to show meaningful information, but an even better solution is to create a configuration file that the driver could read to create an alias. The configuration file looks like this:

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

The field inside the square brackets defines the device address and name. The data that follows specifies each register of that device. The main advantages to this approach are:

- 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 need to recompile the program to recompile.

These predefined devices would always show via the `ls` command.

When not to use a 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.

Nevertheless, you can turn almost everything into a resource manager. If your resource manager's client uses only the POSIX API, there's less documentation to write, and the code is very portable. 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 than a simple *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.

Chapter 2
The Bones of a Resource Manager

Let's start with the overall structure of a resource manager. First we'll look at what happens under the covers on both the client side and the server side. After that, we'll go into the layers in a resource manager and then look at some examples.

Under the covers

Despite the fact that you'll be using a resource manager API that hides many details from you, it's still important to understand what's going on under the covers. For example, your resource manager is a server that contains a *MsgReceive()* loop, and clients send you messages using *MsgSend*()*. This means that you must reply either to your clients in a timely fashion, or leave your clients blocked but save the *rcvid* for use in a later reply.

To help you understand, we'll discuss the events that occur under the covers for both the client and the resource manager.

Under the client's covers

When a client calls a function that requires pathname resolution (e.g., *open()*, *rename()*, *stat()*, or *unlink()*), the function sends messages to both the process manager and the resource manager to obtain a file descriptor. Once the file descriptor is obtained, the client can use it to send messages to the device associated with the pathname, via the resource manager.

In the following, the file descriptor is obtained, and then the client writes directly to the device:

```
/*
 * In this stage, the client talks
 * to the process manager and the resource manager.
 */
fd = open("/dev/ser1", O_RDWR);

/*
 * In this stage, the client talks directly to the
 * resource manager.
 */
for (packet = 0; packet < npackets; packet++)
{
    write(fd, packets[packet], PACKET_SIZE);
}
close(fd);
```

For the above example, here's the description of what happened behind the scenes. We'll assume that a serial port is managed by a resource manager called `devc-ser8250`, that's been registered with the pathname prefix **/dev/ser1**:

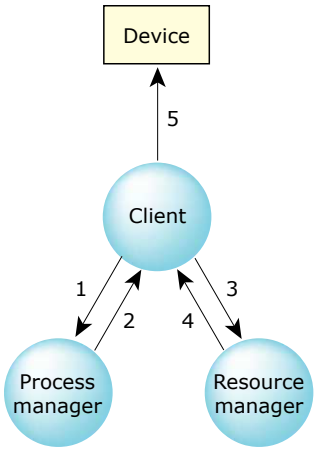

Figure 1: Under-the-cover communication between the client, the process manager, and the resource manager.

1. The client's library sends a "query" message. The *open()* in the client's library sends a message to the process manager asking it to look up a name (e.g., **/dev/ser1**).
2. The process manager indicates who's responsible and it returns the *nd*, *pid*, *chid*, and *handle* that are associated with the pathname prefix.

   Here's what went on behind the scenes...

   When the `devc-ser8250` resource manager registered its name (**/dev/ser1**) in the namespace, it called the process manager. The process manager is responsible for maintaining information about pathname prefixes. During registration, it adds an entry to its table that looks similar to this:

   ```
   0, 47167, 1, 0, 0, /dev/ser1
   ```

   The table entries represent:

   0

   Node descriptor (*nd*).

   47167

   Process ID (*pid*) of the resource manager.

   1

   Channel ID (*chid*) that the resource manager is using to receive messages with.

   0

   Handle given in case the resource manager has registered more than one name. The handle for the first name is 0, 1 for the next name, etc.

   0

   The open type passed during name registration (0 is _FTYPE_ANY).

   /dev/ser1

   The pathname prefix.

A resource manager is uniquely identified by a node descriptor, process ID, and a channel ID. The process manager's table entry associates the resource manager with a name, a handle (to distinguish multiple names when a resource manager registers more than one name), and an open type.

When the client's library issued the query call in step 1, the process manager looked through all of its tables for any registered pathname prefixes that match the name. If another resource manager had previously registered the name **/**, more than one match would be found. So, in this case, both **/** and **/dev/ser1** match. The process manager will reply to the *open()* with the list of matched servers or resource managers. The servers are queried in turn about their handling of the path, with the longest match being asked first.

3. The client's library sends a "connect" message to the resource manager. To do so, it must create a connection to the resource manager's channel:

```
fd = ConnectAttach(nd, pid, chid, 0, 0);
```

The file descriptor that's returned by *ConnectAttach()* is also a connection ID and is used for sending messages directly to the resource manager. In this case, it's used to send a connect message (_IO_CONNECT defined in **<sys/iomsg.h>**) containing the handle to the resource manager requesting that it open **/dev/ser1**.

Typically, only functions such as *open()* call *ConnectAttach()* with an *index* argument of 0. Most of the time, you should OR _NTO_SIDE_CHANNEL into this argument, so that the connection is made via a *side channel*, resulting in a connection ID that's greater than any valid file descriptor.

When the resource manager gets the connect message, it performs validation using the access modes specified in the *open()* call (e.g., are you trying to write to a read-only device?).

4. The resource manager generally responds with a pass (and *open()* returns with the file descriptor) or fail (the next server is queried).
5. When the file descriptor is obtained, the client can use it to send messages directly to the device associated with the pathname.

   In the sample code, it looks as if the client opens and writes directly to the device. In fact, the *write()* call sends an _IO_WRITE message to the resource manager requesting that the given data be written, and the resource manager responds that it either wrote some of all of the data, or that the write failed.

Eventually, the client calls *close()*, which sends an _IO_CLOSE message to the resource manager. The resource manager handles this by doing some cleanup.

Under the resource manager's covers

The resource manager is a server that uses the QNX Neutrino send/receive/reply messaging protocol to receive and reply to messages. The following is pseudo-code for a resource manager:

```
initialize the resource manager
register the name with the process manager
DO forever
    receive a message
```

```
    SWITCH on the type of message
        CASE _IO_CONNECT:
            call io_open handler
            ENDCASE
        CASE _IO_READ:
            call io_read handler
            ENDCASE
        CASE _IO_WRITE:
            call io_write handler
            ENDCASE
        .   /* etc. handle all other messages */
        .   /* that may occur, performing     */
        .   /* processing as appropriate      */
    ENDSWITCH
ENDDO
```

Many of the details in the above pseudo-code are hidden from you by a resource manager library that you'll use. For example, you won't actually call a *MsgReceive*()* function — you'll call a library function, such as *resmgr_block()* or *dispatch_block()*, that does it for you. If you're writing a single-threaded resource manager, you might provide a message handling loop, but if you're writing a multithreaded resource manager, the loop is hidden from you.

You don't need to know the format of all the possible messages, and you don't have to handle them all. Instead, you register "handler functions," and when a message of the appropriate type arrives, the library calls your handler. For example, suppose you want a client to get data from you using *read()* — you'll write a handler that's called whenever an _IO_READ message is received. Since your handler handles _IO_READ messages, we'll call it an "*io_read handler.*"

The resource manager library:

1. Receives the message.
2. Examines the message to verify that it's an _IO_READ message.
3. Calls your *io_read* handler.

However, it's still your responsibility to reply to the _IO_READ message. You can do that from within your *io_read* handler, or later on when data arrives (possibly as the result of an interrupt from some data-generating hardware).

The library does default handling for any messages that you don't want to handle. After all, most resource managers don't care about presenting proper POSIX filesystems to the clients. When writing them, you want to concentrate on the code for talking to the device you're controlling. You don't want to spend a lot of time worrying about the code for presenting a proper POSIX filesystem to the client.

Layers in a resource manager

A resource manager is composed of some of the following layers:

- thread pool layer (the top layer)
- dispatch layer
- resmgr layer
- iofunc layer (the bottom layer)

Let's look at these from the bottom up.

The iofunc layer

This layer consists of a set of functions that take care of most of the POSIX filesystem details for you—they provide a POSIX personality. If you're writing a device resource manager, you'll want to use this layer so that you don't have to worry too much about the details involved in presenting a POSIX filesystem to the world.

This layer consists of default handlers that the resource manager library uses if you don't provide a handler. For example, if you don't provide an *io_open* handler, *iofunc_open_default()* is called.

The iofunc layer also contains helper functions that the default handlers call. If you override the default handlers with your own, you can still call these helper functions. For example, if you provide your own *io_read* handler, you can call *iofunc_read_verify()* at the start of it to make sure that the client has access to the resource.

The names of the functions and structures for this layer have the form *iofunc_**. The header file is **<sys/iofunc.h>**. For more information, see the QNX Neutrino *C Library Reference*.

The resmgr layer

This layer manages most of the resource manager library details. It:

- examines incoming messages
- calls the appropriate handler to process a message

If you don't use this layer, then you'll have to parse the messages yourself. Most resource managers use this layer.

The names of the functions and structures for this layer have the form *resmgr_**. The header file is **<sys/resmgr.h>**. For more information, see the QNX Neutrino *C Library Reference*.

Be sure to `#include` **<sys/iofunc.h>** before **<sys/resmgr.h>**, or else the data structures won't be defined properly.

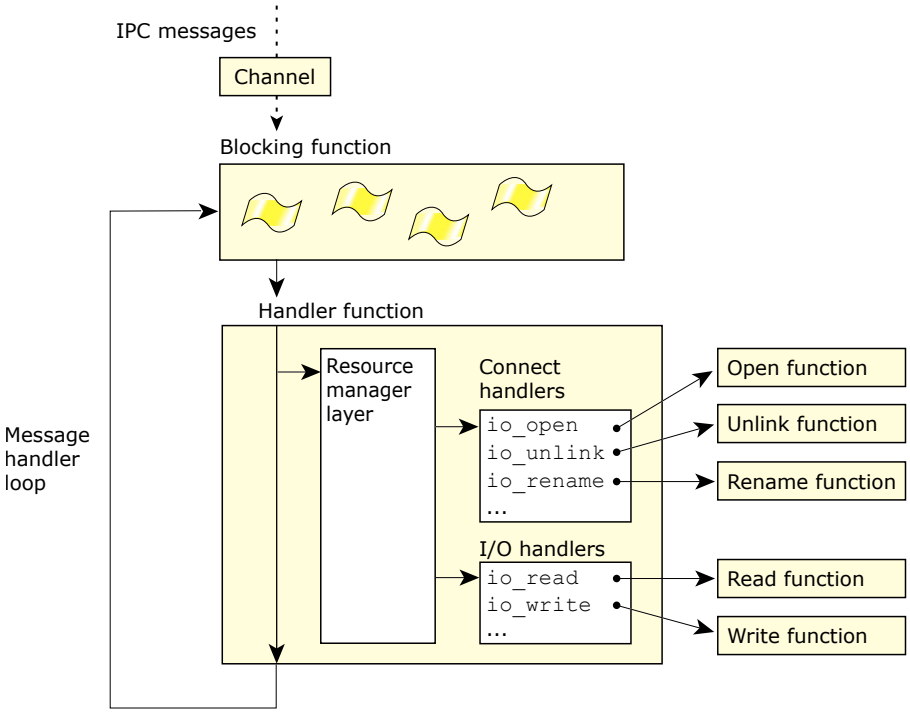

Figure 2: You can use the resmgr layer to handle _IO_* messages.

The dispatch layer

This layer acts as a single blocking point for a number of different types of things. With this layer, you can handle:

_IO_* messages

It uses the resmgr layer for this.

select()

Processes that do TCP/IP often call *select()* to block while waiting for packets to arrive, or for there to be room for writing more data. With the dispatch layer, you register a handler function that's called when a packet arrives. The functions for this are the *select_*()* functions.

Pulses

As with the other layers, you register a handler function that's called when a specific pulse arrives. The functions for this are the *pulse_*()* functions.

Other messages

You can give the dispatch layer a range of message types that you make up, and a handler. So if a message arrives and the first few bytes of the message contain a type in the given range, the dispatch layer calls your handler. The functions for this are the *message_*()* functions.

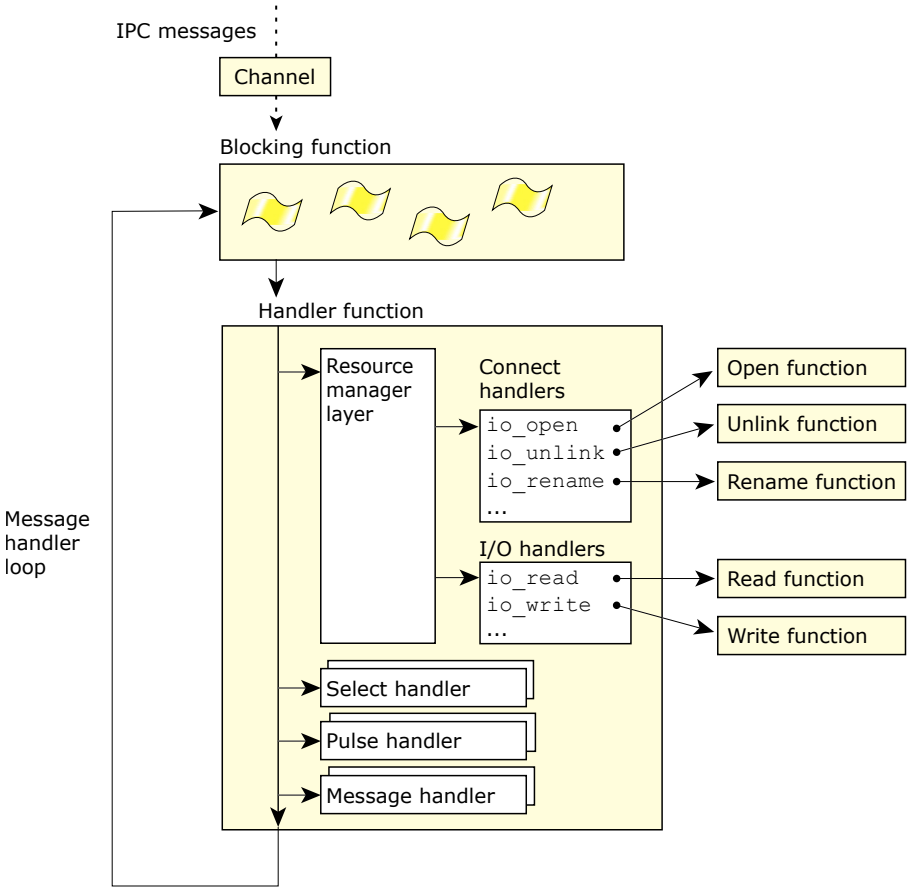

Figure 3: You can use the dispatch layer to handle _IO_* messages, select, pulses, and other messages.

Here's how messages are handled via the dispatch layer (or more precisely, through *dispatch_handler()*):

- Depending on the blocking type, the handler may call the *message_*()* subsystem. A search is made, based on the message type or pulse code, for a matching function that was attached using *message_attach()* or *pulse_attach()*. If a match is found, the attached function is called.
- If the message type is in the range handled by the resource manager (I/O messages) and pathnames were attached using *resmgr_attach()*, the resource manager subsystem is called and handles the resource manager message.
- If a pulse is received, it may be dispatched to the resource manager subsystem if it's one of the codes handled by a resource manager (UNBLOCK and DISCONNECT pulses). If a *select_attach()* is done and the pulse matches the one used by *select*, then the *select* subsystem is called and dispatches that event.
- If a message is received and no matching handler is found for that message type, *MsgError*(ENOSYS) is returned to unblock the sender.

The thread pool layer

This layer allows you to have a single- or multithreaded resource manager. This means that one thread can be handling a *write()* while another thread handles a *read()*.

You provide the blocking function for the threads to use as well as the handler function that's to be called when the blocking function returns. Most often, you give it the dispatch layer's functions. However, you can also give it the resmgr layer's functions or your own.

You can use this layer independently of the resource manager layer.

Simple examples of device resource managers

The programs that follow are complete but **simple** examples of a device resource manager.

As you read through this guide, you'll encounter many code snippets. Most of these code snippets have been written so that they can be combined with either of these simple resource managers.

The first two of these simple device resource managers model their functionality after that provided by **/dev/null** (although they use **/dev/sample** to avoid conflict with the "real" **/dev/null**):

- an *open()* always works
- *read()* returns zero bytes (indicating EOF)
- a *write()* of any size "works" (with the data being discarded)
- lots of other POSIX functions work (e.g., *chown()*, *chmod()*, *lseek()*)

The chapters that follow describe how to add more functionality to these simple resource managers.

The QNX Momentics Integrated Development Environment (IDE) includes a sample **/dev/sample** resource manager that's very similar to the single-threaded one given below. To get the sample in the IDE, choose **Help → Welcome**, and then click the Samples icon.

Single-threaded device resource manager

Here's the complete code for a simple single-threaded device resource manager:

```
#include <errno.h>
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>

static resmgr_connect_funcs_t    connect_funcs;
static resmgr_io_funcs_t         io_funcs;
static iofunc_attr_t             attr;

int main(int argc, char **argv)
{
    /* declare variables we'll be using */
    resmgr_attr_t        resmgr_attr;
    dispatch_t           *dpp;
    dispatch_context_t   *ctp;
    int                  id;

    /* initialize dispatch interface */
    if((dpp = dispatch_create()) == NULL) {
```

```
        fprintf(stderr,
                "%s: Unable to allocate dispatch handle.\n",
                argv[0]);
        return EXIT_FAILURE;
    }

    /* initialize resource manager attributes */
    memset(&resmgr_attr, 0, sizeof resmgr_attr);
    resmgr_attr.nparts_max = 1;
    resmgr_attr.msg_max_size = 2048;

    /* initialize functions for handling messages */
    iofunc_func_init(_RESMGR_CONNECT_NFUNCS, &connect_funcs,
                     _RESMGR_IO_NFUNCS, &io_funcs);

    /* initialize attribute structure used by the device */
    iofunc_attr_init(&attr, S_IFNAM | 0666, 0, 0);

    /* attach our device name */
    id = resmgr_attach(
            dpp,            /* dispatch handle        */
            &resmgr_attr,   /* resource manager attrs */
            "/dev/sample",  /* device name            */
            _FTYPE_ANY,     /* open type              */
            0,              /* flags                  */
            &connect_funcs, /* connect routines       */
            &io_funcs,      /* I/O routines           */
            &attr);         /* handle                 */
    if(id == -1) {
        fprintf(stderr, "%s: Unable to attach name.\n", argv[0]);
        return EXIT_FAILURE;
    }

    /* allocate a context structure */
    ctp = dispatch_context_alloc(dpp);

    /* start the resource manager message loop */
    while(1) {
        if((ctp = dispatch_block(ctp)) == NULL) {
            fprintf(stderr, "block error\n");
            return EXIT_FAILURE;
        }
        dispatch_handler(ctp);
    }
    return EXIT_SUCCESS;
}
```

Include **<sys/dispatch.h>** *after* **<sys/iofunc.h>** to avoid warnings about redefining the members of some functions.

Let's examine the sample code step-by-step.

Initialize the dispatch interface

```
/* initialize dispatch interface */
if((dpp = dispatch_create()) == NULL) {
    fprintf(stderr, "%s: Unable to allocate dispatch handle.\n",
            argv[0]);
    return EXIT_FAILURE;
}
```

We need to set up a mechanism so that clients can send messages to the resource manager. This is done via the *dispatch_create()* function which creates and returns the dispatch structure. This structure contains the channel ID. Note that the channel ID isn't actually created until you attach something, as in *resmgr_attach()*, *message_attach()*, and *pulse_attach()*.

In order to register a prefix in the pathname space, a resource manager must have the PROCMGR_AID_PATHSPACE ability enabled. In order to create a public channel (i.e., *without* _NTO_CHF_PRIVATE set), your process must have the PROCMGR_AID_PUBLIC_CHANNEL ability enabled. For more information, see *procmgr_ability()*.

The dispatch structure (of type `dispatch_t`) is opaque; you can't access its contents directly. Use *message_connect()* to create a connection using this hidden channel ID.

Initialize the resource manager attributes

When you call *resmgr_attach()*, you pass a `resmgr_attr_t` control structure to it. Our sample code initializes this structure like this:

```
/* initialize resource manager attributes */
memset(&resmgr_attr, 0, sizeof resmgr_attr);
resmgr_attr.nparts_max = 1;
resmgr_attr.msg_max_size = 2048;
```

In this case, we're configuring:

- how many IOV structures are available for server replies (*nparts_max*)
- the minimum receive buffer size (*msg_max_size*)

For more information, see *resmgr_attach()* in the QNX Neutrino *C Library Reference*.

Initialize functions used to handle messages

```
/* initialize functions for handling messages */
iofunc_func_init(_RESMGR_CONNECT_NFUNCS, &connect_funcs,
                 _RESMGR_IO_NFUNCS, &io_funcs);
```

Here we supply two tables that specify which function to call when a particular message arrives:

- *connect functions* table
- *I/O functions* table

Instead of filling in these tables manually, we call *iofunc_func_init()* to place the *iofunc_*_default()* handler functions into the appropriate spots.

Initialize the attribute structure used by the device

```
/* initialize attribute structure used by the device */
iofunc_attr_init(&attr, S_IFNAM | 0666, 0, 0);
```

The attribute structure contains information about our particular device associated with the name **/dev/sample**. It contains at least the following information:

- permissions and type of device
- owner and group ID

Effectively, this is a *per-name* data structure. In the *Extending the POSIX-Layer Data Structures* chapter, we'll see how you can extend the structure to include your own *per-device* information.

Put a name into the namespace

To register our resource manager's path, we call *resmgr_attach()* like this:

```
/* attach our device name */
id = resmgr_attach(dpp,            /* dispatch handle        */
                   &resmgr_attr,   /* resource manager attrs */
                   "/dev/sample",  /* device name            */
                   _FTYPE_ANY,     /* open type              */
                   0,              /* flags                  */
                   &connect_funcs, /* connect routines       */
                   &io_funcs,      /* I/O routines           */
                   &attr);         /* handle                 */
if(id == -1) {
    fprintf(stderr, "%s: Unable to attach name.\n", argv[0]);
    return EXIT_FAILURE;
}
```

Before a resource manager can receive messages from other programs, it needs to inform the other programs (via the process manager) that it's the one responsible for a particular pathname prefix. This is done via pathname registration. When the name is registered, other processes can find and connect to this process using the registered name.

In this example, a serial port may be managed by a resource manager called `devc-xxx`, but the actual resource is registered as **/dev/sample** in the pathname space. Therefore, when a program requests serial port services, it opens the **/dev/sample** serial port.

We'll look at the parameters in turn, skipping the ones we've already discussed.

device name

Name associated with our device (i.e., **/dev/sample**).

open type

Specifies the constant value of _FTYPE_ANY. This tells the process manager that our resource manager will accept *any* type of open request; we're not limiting the kinds of connections we're going to be handling.

Some resource managers legitimately limit the types of open requests they handle. For instance, the POSIX message queue resource manager accepts only open messages of type _FTYPE_MQUEUE.

flags

Controls the process manager's pathname resolution behavior. By specifying a value of zero, we indicate that we'll accept only requests for the name "**/dev/sample**".

The bits that you use in this argument are the _RESMGR_FLAG_* flags (e.g., _RESMGR_FLAG_BEFORE) defined in **<sys/resmgr.h>**. We'll discuss some of these flags in this guide, but you can find a full list in the entry for *resmgr_attach()* in the QNX Neutrino *C Library Reference*.

There are some other flags whose names *don't* start with an underscore, but they're for the *flags* member of the `resmgr_attr_t` structure, which we'll look at in more detail in "*Setting resource manager attributes*" in the Fleshing Out the Skeleton chapter.

If you want to use the same dispatch handler for other types of notifications, you can also call *message_attach()*, *pulse_attach()*, and *select_attach()* at this point.

Allocate the context structure

```
/* allocate a context structure */
ctp = dispatch_context_alloc(dpp);
```

The context structure contains a buffer where messages will be received. The size of the buffer was set when we initialized the resource manager attribute structure. The context structure also contains a buffer of IOVs that the library can use for replying to messages. The number of IOVs was set when we initialized the resource manager attribute structure.

For more information, see *dispatch_context_alloc()* in the QNX Neutrino *C Library Reference*.

Once you've called *dispatch_context_alloc()*, don't call *message_attach()* or *resmgr_attach()* specifying a larger maximum message size or a larger number of message parts for the same dispatch handle. In QNX Neutrino 7.0 or later, these functions indicate an error of EINVAL if this happens. (This doesn't apply to *pulse_attach()* or *select_attach()* because you can't specify the sizes with these functions.)

Start the resource manager message loop

```
/* start the resource manager message loop */
while(1) {
    if((ctp = dispatch_block(ctp)) == NULL) {
        fprintf(stderr, "block error\n");
        return EXIT_FAILURE;
    }
    dispatch_handler(ctp);
}
```

Once the resource manager establishes its name, it receives messages when any client program tries to perform an operation (e.g., *open()*, *read()*, *write()*) on that name.

In our example, once **/dev/sample** is registered, and a client program executes:

```
fd = open ("/dev/sample", O_RDONLY);
```

the client's C library constructs an _IO_CONNECT message and sends it to our resource manager. Our resource manager receives the message within the *dispatch_block()* function. We then call *dispatch_handler()*, which decodes the message and calls the appropriate handler function based on

the connect and I/O function tables that we passed in previously. After *dispatch_handler()* returns, we go back to the *dispatch_block()* function to wait for another message.

Note that *dispatch_block()* returns a pointer to a dispatch context (`dispatch_context_t`) structure—the same type of pointer you pass to the routine:

- If *dispatch_block()* returns a non-NULL context pointer, it could be different from the one passed in, as it's possible for the *ctp* to be reallocated to a larger size. In this case, the old *ctp* is no longer valid.
- If *dispatch_block()* returns NULL (for example, because a signal interrupted the *MsgReceive()*), the old context pointer is still valid. Typically, a resource manager targets any signals to a thread that's dedicated to handling signals. However, if a signal can be targeted to the thread doing *dispatch_block()*, you could use the following code:

```
dispatch_context_t   *ctp, *new_ctp;

ctp = dispatch_context_alloc( … );
while (1) {
   new_ctp = dispatch_block( ctp );
   if ( new_ctp ) {
      ctp = new_ctp
      }
   else {
      /* handle the error condition */
      …
      }
}
```

At some later time, when the client program executes:

```
read (fd, buf, BUFSIZ);
```

the client's C library constructs an _IO_READ message, which is then sent directly to our resource manager, and the decoding cycle repeats.

Multithreaded device resource manager

Here's the complete code for a simple multithreaded device resource manager:

```
#include <errno.h>
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

/*
 * Define THREAD_POOL_PARAM_T such that we can avoid a compiler
 * warning when we use the dispatch_*() functions below
 */
#define THREAD_POOL_PARAM_T dispatch_context_t

#include <sys/iofunc.h>
```

```
#include <sys/dispatch.h>

static resmgr_connect_funcs_t    connect_funcs;
static resmgr_io_funcs_t         io_funcs;
static iofunc_attr_t             attr;

int main(int argc, char **argv)
{
    /* declare variables we'll be using */
    thread_pool_attr_t   pool_attr;
    resmgr_attr_t        resmgr_attr;
    dispatch_t           *dpp;
    thread_pool_t        *tpp;
    int                  id;

    /* initialize dispatch interface */
    if((dpp = dispatch_create()) == NULL) {
        fprintf(stderr, "%s: Unable to allocate dispatch handle.\n",
                argv[0]);
        return EXIT_FAILURE;
    }

    /* initialize resource manager attributes */
    memset(&resmgr_attr, 0, sizeof resmgr_attr);
    resmgr_attr.nparts_max = 1;
    resmgr_attr.msg_max_size = 2048;

    /* initialize functions for handling messages */
    iofunc_func_init(_RESMGR_CONNECT_NFUNCS, &connect_funcs,
                     _RESMGR_IO_NFUNCS, &io_funcs);

    /* initialize attribute structure used by the device */
    iofunc_attr_init(&attr, S_IFNAM | 0666, 0, 0);

    /* attach our device name */
    id = resmgr_attach(dpp,            /* dispatch handle        */
                       &resmgr_attr,   /* resource manager attrs */
                       "/dev/sample",  /* device name            */
                       _FTYPE_ANY,     /* open type              */
                       0,              /* flags                  */
                       &connect_funcs, /* connect routines       */
                       &io_funcs,      /* I/O routines           */
                       &attr);         /* handle                 */
    if(id == -1) {
        fprintf(stderr, "%s: Unable to attach name.\n", argv[0]);
        return EXIT_FAILURE;
    }

    /* initialize thread pool attributes */
    memset(&pool_attr, 0, sizeof pool_attr);
    pool_attr.handle = dpp;
    pool_attr.context_alloc = dispatch_context_alloc;
    pool_attr.block_func = dispatch_block;
    pool_attr.unblock_func = dispatch_unblock;
```

```
    pool_attr.handler_func = dispatch_handler;
    pool_attr.context_free = dispatch_context_free;
    pool_attr.lo_water = 2;
    pool_attr.hi_water = 4;
    pool_attr.increment = 1;
    pool_attr.maximum = 50;

    /* allocate a thread pool handle */
    if((tpp = thread_pool_create(&pool_attr,
                                 POOL_FLAG_EXIT_SELF)) == NULL) {
        fprintf(stderr, "%s: Unable to initialize thread pool.\n",
                argv[0]);
        return EXIT_FAILURE;
    }

    /* Start the threads. This function doesn't return. */
    thread_pool_start(tpp);
    return EXIT_SUCCESS;
}
```

Most of the code is the same as in the single-threaded example, so we'll cover only those parts that aren't described above. Also, we'll go into more detail on multithreaded resource managers later in this guide, so we'll keep the details here to a minimum.

For this code sample, the threads are using the *dispatch_*()* functions (i.e., the dispatch layer) for their blocking loops.

Define THREAD_POOL_PARAM_T

```
/*
 * Define THREAD_POOL_PARAM_T such that we can avoid a compiler
 * warning when we use the dispatch_*() functions below
 */
#define THREAD_POOL_PARAM_T dispatch_context_t

#include <sys/iofunc.h>
#include <sys/dispatch.h>
```

The THREAD_POOL_PARAM_T manifest tells the compiler what type of parameter is passed between the various blocking/handling functions that the threads will be using. This parameter should be the context structure used for passing context information between the functions. By default it's defined as a `resmgr_context_t`, but since this sample is using the dispatch layer, we need it to be a `dispatch_context_t`. We define it prior to the `include` directives above, since the header files refer to it.

Initialize thread pool attributes

```
/* initialize thread pool attributes */
memset(&pool_attr, 0, sizeof pool_attr);
pool_attr.handle = dpp;
pool_attr.context_alloc = dispatch_context_alloc;
pool_attr.block_func = dispatch_block;
pool_attr.unblock_func = dispatch_unblock;
pool_attr.handler_func = dispatch_handler;
pool_attr.context_free = dispatch_context_free;
```

```
pool_attr.lo_water = 2;
pool_attr.hi_water = 4;
pool_attr.increment = 1;
pool_attr.maximum = 50;
```

The thread pool attributes tell the threads which functions to use for their blocking loop and control how many threads should be in existence at any time. We'll go into more detail on these attributes when we talk about multithreaded resource managers in more detail later in this guide.

Allocate a thread pool handle

```
/* allocate a thread pool handle */
if((tpp = thread_pool_create(&pool_attr,
                             POOL_FLAG_EXIT_SELF)) == NULL) {
    fprintf(stderr, "%s: Unable to initialize thread pool.\n",
            argv[0]);
    return EXIT_FAILURE;
}
```

The thread pool handle is used to control the thread pool. Among other things, it contains the given attributes and flags. The *thread_pool_create()* function allocates and fills in this handle.

Start the threads

```
/* start the threads; will not return */
thread_pool_start(tpp);
```

The *thread_pool_start()* function starts up the thread pool. Each newly created thread allocates a context structure of the type defined by THREAD_POOL_PARAM_T using the *context_alloc* function we gave above in the attribute structure. They'll then block on the *block_func* and when the *block_func* returns, they'll call the *handler_func*, both of which were also given through the attributes structure. Each thread essentially does the same thing that the single-threaded resource manager above does for its message loop.

From this point on, your resource manager is ready to handle messages. Since we gave the POOL_FLAG_EXIT_SELF flag to *thread_pool_create()*, once the threads have been started up, *pthread_exit()* will be called and this calling thread will exit.

Chapter 3
Fleshing Out the Skeleton

It's time now to start adding some flesh to the basic bones of the resource manager. We'll look at the types of messages you might have to handle, how to set the resource manager's attributes, how to add functionality, and some security issues you should consider.

Message types

As we saw in the *Bones of a Resource Manager* chapter, your resource manager may need to handle these types of messages:

- *connect messages*
- *I/O messages*

We'll examine them in the sections and chapters that follow.

The *Getting Started with QNX Neutrino* guide includes a summary of the handlers for these messages; see "Alphabetical listing of connect and I/O functions" in its Resource Managers chapter.

Connect messages

A connect message is issued by the client to perform an operation based on a pathname. This may be a message that establishes a longer term relationship between the client and the resource manager (e.g., *open()*), or it may be a message that is a "one-shot" event (e.g., *rename()*).

When you call *resmgr_attach()*, you pass it a pointer to a `resmgr_connect_funcs_t` structure that defines your connect functions. This structure is defined in **<sys/resmgr.h>** as follows:

```
typedef struct _resmgr_connect_funcs {

    unsigned nfuncs;

    int (*open)      (resmgr_context_t *ctp, io_open_t *msg,
                      RESMGR_HANDLE_T *handle, void *extra);

    int (*unlink)    (resmgr_context_t *ctp, io_unlink_t *msg,
                      RESMGR_HANDLE_T *handle, void *reserved);

    int (*rename)    (resmgr_context_t *ctp, io_rename_t *msg,
                      RESMGR_HANDLE_T *handle,
                      io_rename_extra_t *extra);

    int (*mknod)     (resmgr_context_t *ctp, io_mknod_t *msg,
                      RESMGR_HANDLE_T *handle, void *reserved);

    int (*readlink)  (resmgr_context_t *ctp, io_readlink_t *msg,
                      RESMGR_HANDLE_T *handle, void *reserved);

    int (*link)      (resmgr_context_t *ctp, io_link_t *msg,
                      RESMGR_HANDLE_T *handle,
                      io_link_extra_t *extra);

    int (*unblock)   (resmgr_context_t *ctp, io_pulse_t *msg,
                      RESMGR_HANDLE_T *handle, void *reserved);
```

```
    int (*mount)     (resmgr_context_t *ctp, io_mount_t *msg,
                      RESMGR_HANDLE_T *handle,
                      io_mount_extra_t *extra);
} resmgr_connect_funcs_t;
```

To initialize this structure, call *iofunc_func_init()* to fill it with pointers to the default handlers, and then override any that your resource manager needs to handle specifically.

- In order to correctly define `RESMGR_HANDLE_T`, `#include` **<sys/iofunc.h>** before **<sys/resmgr.h>**.
- The *resmgr_attach()* function copies the *pointers* to the `resmgr_connect_funcs_t` and `resmgr_io_funcs_t` structures, not the structures themselves. You should allocate the structures, declare them to be static, or make them global variables. If your resource manager is for more than one device with different handlers, create separate structures that define the handlers.

The connect messages all have a type of _IO_CONNECT; the subtype further indicates what's happening. The entries are as follows:

nfuncs

The number of functions in the structure. This allows for future expansion.

open

Handles client calls to *open()*, *fopen()*, *sopen()*, and so on. The message subtype is _IO_CONNECT_COMBINE, _IO_CONNECT_COMBINE_CLOSE, or _IO_CONNECT_OPEN.

For more information about the *io_open* handler, see "*Ways of adding functionality to the resource manager*," later in this chapter.

unlink

Handles client calls to *unlink()*. The message subtype is _IO_CONNECT_UNLINK.

rename

Handles client calls to *rename()*. The message subtype is _IO_CONNECT_RENAME.

mknod

Handles client calls to *mkdir()*, *mkfifo()*, and *mknod()*. The message subtype is _IO_CONNECT_MKNOD.

readlink

Handles client calls to *readlink()*. The message subtype is _IO_CONNECT_READLINK.

link

Handles client calls to *link()*. The message subtype is _IO_CONNECT_LINK.

unblock

Handles requests from the kernel to unblock a client during the connect message phase. There's no corresponding message; the call is synthesized by the library.

For more information about the *io_unblock* handler, see "*Handling client unblocking due to signals or timeouts*" in the Unblocking Clients and Handling Interrupts chapter.

mount

Handles client calls to *mount()*. The message subtype is _IO_CONNECT_MOUNT.

For more information about the *io_mount* handler, see "*Handling mount()*" in the Handling Other Messages chapter.

If the message is the _IO_CONNECT message (and variants) corresponding with the *open()* outcall, then a *context* needs to be established for further I/O messages that will be processed later. This context is referred to as an *OCB (Open Control Block)*; it holds any information required between the connect message and subsequent I/O messages.

Basically, the OCB is a good place to keep information that needs to be stored on a per-open basis. An example of this would be the current position within a file. Each open file descriptor would have its own file position. The OCB is allocated on a per-open basis. During the open handling, you'd initialize the file position; during read and write handling, you'd advance the file position. For more information, see the section "*The open control block (OCB) structure*" in the POSIX-Layer Data Structures chapter of this guide.

I/O messages

An I/O message is one that relies on an existing binding (e.g., OCB) between the client and the resource manager.

As an example, an _IO_READ (from the client's *read()* function) message depends on the client's having previously established an association (or *context*) with the resource manager by issuing an *open()* and getting back a file descriptor. This context, created by the *open()* call, is then used to process the subsequent I/O messages, like the _IO_READ.

There are good reasons for this design. It would be inefficient to pass the full pathname for each and every *read()* request, for example. The *open()* handler can also perform tasks that we want done only once (e.g., permission checks), rather than with each I/O message. Also, when the *read()* has read 4096 bytes from a disk file, there may be another 20 megabytes still waiting to be read. Therefore, the *read()* function would need to have some context information telling it the position within the file it's reading from, how much has been read, and so on.

The `resmgr_io_funcs_t` structure (which you pass to *resmgr_attach()* along with the connect functions) defines the functions to call for the I/O messages. The `resmgr_io_funcs_t` structure is defined in **<sys/resmgr.h>** as shown below.

In order to correctly define `RESMGR_OCB_T`, `#include` **<sys/iofunc.h>** before **<sys/resmgr.h>**.

```
typedef struct _resmgr_io_funcs {
   unsigned   nfuncs;
   int (*read)       (resmgr_context_t *ctp, io_read_t *msg, RESMGR_OCB_T *ocb);
   int (*write)      (resmgr_context_t *ctp, io_write_t *msg, RESMGR_OCB_T *ocb);
   int (*close_ocb)  (resmgr_context_t *ctp, void *reserved, RESMGR_OCB_T *ocb);
```

```
    int (*stat)       (resmgr_context_t *ctp, io_stat_t *msg, RESMGR_OCB_T *ocb);
    int (*notify)     (resmgr_context_t *ctp, io_notify_t *msg, RESMGR_OCB_T *ocb);
    int (*devctl)     (resmgr_context_t *ctp, io_devctl_t *msg, RESMGR_OCB_T *ocb);
    int (*unblock)    (resmgr_context_t *ctp, io_pulse_t *msg, RESMGR_OCB_T *ocb);
    int (*pathconf)   (resmgr_context_t *ctp, io_pathconf_t *msg, RESMGR_OCB_T *ocb);
    int (*lseek)      (resmgr_context_t *ctp, io_lseek_t *msg, RESMGR_OCB_T *ocb);
    int (*chmod)      (resmgr_context_t *ctp, io_chmod_t *msg, RESMGR_OCB_T *ocb);
    int (*chown)      (resmgr_context_t *ctp, io_chown_t *msg, RESMGR_OCB_T *ocb);
    int (*utime)      (resmgr_context_t *ctp, io_utime_t *msg, RESMGR_OCB_T *ocb);
    int (*openfd)     (resmgr_context_t *ctp, io_openfd_t *msg, RESMGR_OCB_T *ocb);
    int (*fdinfo)     (resmgr_context_t *ctp, io_fdinfo_t *msg, RESMGR_OCB_T *ocb);
    int (*lock)       (resmgr_context_t *ctp, io_lock_t *msg, RESMGR_OCB_T *ocb);
    int (*space)      (resmgr_context_t *ctp, io_space_t *msg, RESMGR_OCB_T *ocb);
    int (*shutdown)   (resmgr_context_t *ctp, io_shutdown_t *msg, RESMGR_OCB_T *ocb);
    int (*mmap)       (resmgr_context_t *ctp, io_mmap_t *msg, RESMGR_OCB_T *ocb);
    int (*msg)        (resmgr_context_t *ctp, io_msg_t *msg, RESMGR_OCB_T *ocb);
    int (*reserved)   (resmgr_context_t *ctp, void *msg, RESMGR_OCB_T *ocb);
    int (*dup)        (resmgr_context_t *ctp, io_dup_t *msg, RESMGR_OCB_T *ocb);
    int (*close_dup)  (resmgr_context_t *ctp, io_close_t *msg, RESMGR_OCB_T *ocb);
    int (*lock_ocb)   (resmgr_context_t *ctp, void *reserved, RESMGR_OCB_T *ocb);
    int (*unlock_ocb) (resmgr_context_t *ctp, void *reserved, RESMGR_OCB_T *ocb);
    int (*sync)       (resmgr_context_t *ctp, io_sync_t *msg, RESMGR_OCB_T *ocb);
    int (*power)      (resmgr_context_t *ctp, io_power_t *msg, RESMGR_OCB_T *ocb);
    int (*acl)        (resmgr_context_t *ctp, io_acl_t *msg, RESMGR_OCB_T *ocb);
    int (*pause)      (resmgr_context_t *ctp, void *reserved, RESMGR_OCB_T *ocb);
    int (*unpause)    (resmgr_context_t *ctp, io_pulse_t *msg, RESMGR_OCB_T *ocb);
    int (*read64)     (resmgr_context_t *ctp, io_read_t *msg, RESMGR_OCB_T *ocb);
    int (*write64)    (resmgr_context_t *ctp, io_write_t *msg, RESMGR_OCB_T *ocb);
    int (*notify64)   (resmgr_context_t *ctp, io_notify_t *msg, RESMGR_OCB_T *ocb);
    int (*utime64)    (resmgr_context_t *ctp, io_utime_t *msg, RESMGR_OCB_T *ocb);
} resmgr_io_funcs_t;
```

You initialize this structure in the same way as the `resmgr_connect_funcs_t` structure: call *iofunc_func_init()* to fill it with pointers to the default handlers, and then override any that your resource manager needs to handle specifically. This structure also begins with an *nfuncs* member that indicates how many functions are in the structure, to allow for future expansion.

The *resmgr_attach()* function copies the *pointers* to the `resmgr_connect_funcs_t` and `resmgr_io_funcs_t` structures, not the structures themselves. You should allocate the structures, declare them to be static, or make them global variables. If your resource manager is for more than one device with different handlers, create separate structures that define the handlers.

Notice that the I/O functions all have a common parameter list. The first entry is a resource manager context structure, the second is a message (the type of which matches the message being handled and contains parameters sent from the client), and the last is an OCB (containing what we bound when we handled the client's *open()* function).

You usually have to provide a handler for the following entries:

read*, *read64

Handles client calls to *read()* and *readdir()*. The message type is _IO_READ. For more information about the *io_read* handler, see "*Handling the* _IO_READ *message*" in the Handling Read and Write Messages chapter.

The *read64* handler is for communication with older servers. The message type is _IO_READ64, and the message uses a previously unused member to specify the high 32 bits of length.

write*, *write64

Handles client calls to *write()*, *fwrite()*, and so on. The message type is _IO_WRITE. For more information about the *io_write* handler, see "*Handling the* _IO_WRITE *message*" in the Handling Read and Write Messages chapter.

The *write64* handler is for communication with older servers. The message type is _IO_WRITE64, and the message uses a previously unused member to specify the high 32 bits of length.

devctl

Handles client calls to *devctl()* and *ioctl()*. The message type is _IO_DEVCTL. For more information about the *io_devctl* handler, see "*Handling devctl() messages*" in the Handling Other Messages chapter.

You typically use the default entry for the following:

close_ocb

Called by the library when the last *close()* has been received by a particular OCB. You can use this handler to clean up anything associated with the OCB.

stat

Handles client calls to *stat()*, *lstat()*, and *fstat()*. The message type is _IO_STAT. For more information about the *io_stat* handler, see "*Handling stat()*" in the Handling Other Messages chapter.

notify

Handles client calls to *select()* and *ionotify()*. The message type is _IO_NOTIFY. For more information about the *io_notify* handler, see "*Handling ionotify(), poll(), and select()*" in the Handling Other Messages chapter.

unblock

Handles requests from the kernel to unblock the client during the I/O message phase. There's no message associated with this. For more information about the *io_unblock* handler, see "*Handling client unblocking due to signals or timeouts*" in the Unblocking Clients and Handling Interrupts chapter.

pathconf

Handles client calls to *fpathconf()* and *pathconf()*. The message type is _IO_PATHCONF.

lseek

Handles client calls to *lseek()*, *fseek()*, and *rewinddir()*. The message type is _IO_LSEEK. For more information about the *io_lseek* handler, see "*Handling lseek()*" in the Handling Other Messages chapter.

chmod

Handles client calls to *chmod()* and *fchmod()*. The message type is _IO_CHMOD.

chown

Handles client calls to *chown()* and *fchown()*. The message type is _IO_CHOWN.

utime

Handles client calls to *utime()*. The message type is _IO_UTIME.

openfd

Handles client calls to *openfd()*. The message type is _IO_OPENFD.

fdinfo

Handles client calls to *iofdinfo()*. The message type is _IO_FDINFO.

lock

Handles client calls to *fcntl()*, *lockf()*, and *flock()*. The message type is _IO_LOCK.

space

Handles client calls to *fcntl()*, *ftruncate()*, and *ltrunc()*. The message type is _IO_SPACE.

shutdown

Reserved for future use.

mmap

Handles client calls to *mmap()*, *munmap()*, *mmap_device_io()*, and *mmap_device_memory()*. The message type is _IO_MMAP.

msg

Handles messages that are manually assembled and sent via *MsgSend()*. The message type is _IO_MSG. For more information about the *io_msg* handler, see "*Handling out-of-band (_IO_MSG) messages*" in the Handling Other Messages chapter.

reserved

Reserved for future use.

dup

Handles client calls to *dup()*, *dup2()*, *fcntl()*, *fork()*, *spawn*()*, and *vfork()*. The message type is _IO_DUP. For more information about the *io_dup* handler, see "*Handling open(), dup(), and close() messages*" in the Handling Other Messages chapter.

close_dup

Handles client calls to *close()* and *fclose()*. The message type is _IO_CLOSE.

You'll almost never replace the default *close_dup* handler because the library keeps track of multiple *open()*, *dup()*, and *close()* calls for an OCB. For more information, see "*open(), dup(), and close()*," below.

lock_ocb

Locks the attributes structure pointed to by the OCB. This is done to ensure that only one thread at a time is operating on both the OCB and the corresponding attributes structure. The lock (and corresponding unlock) functions are synthesized by the resource manager library before and after completion of message handling.

unlock_ocb

Unlocks the attributes structure pointed to by the OCB.

sync

Handles client calls to *fsync()* and *fdatasync()*. The message type is _IO_SYNC.

power

Reserved for future use.

acl

Handles messages about access control lists.

pause

Reserved for future use.

unpause

Reserved for future use.

notify64

Similar to *notify*, but for communication with older servers. The message type is _IO_NOTIFY64, and the message extends the structure to handle 64-bit `sigevent`s. For more information about the *io_notify64* handler, see "*Handling ionotify(), poll(), and select()*" in the Handling Other Messages chapter.

utime64

Similar to *utime*, but for communication with older servers. The message type is _IO_UTIME64, and the message uses a different structure to support 64-bit `time_t`s.

Default message handling

Since a large number of the messages received by a resource manager deal with a common set of attributes, the OS provides an *iofunc_*()* shared library that lets a resource manager handle functions like *stat()*, *chmod()*, *chown()*, *lseek()*, and so on *automatically*, without your having to write additional code. As an added benefit, these *iofunc_*()* default handlers implement the POSIX semantics for the messages, offloading some work from you.

The library contains *iofunc_*()* default handlers for these client functions:

- *chmod()*
- *chown()*
- *close()*
- *devctl()*
- *fpathconf()*
- *fseek()*
- *fstat()*
- *lockf()*
- *lseek()*
- *mmap()*
- *open()*
- *pathconf()*
- *stat()*
- *utime()*

open(), *dup()*, and *close()*

The resource manager shared library automatically handles *dup()* messages.

Suppose that the client program executed code that eventually ended up performing:

```
fd = open ("/dev/device", O_RDONLY);
...
fd2 = dup (fd);
...
fd3 = dup (fd);
...
close (fd3);
...
close (fd2);
...
close (fd);
```

The client generates an open connect message for the first *open()*, and then two _IO_DUP messages for the two *dup()* calls. Then, when the client executes the *close()* calls, it generates three close messages.

Since the *dup()* functions generate duplicates of the file descriptors, new context information shouldn't be allocated for each one. When the close messages arrive, because no new context has been allocated for each *dup()*, no *release* of the memory by each close message should occur either! (If it did, the first close would wipe out the context.)

The resource manager shared library provides default handlers that keep track of the *open()*, *dup()*, and *close()* messages and perform work only for the last close (i.e., the third `io_close` message in the example above).

Setting resource manager attributes

In addition to the structures that define the connect and I/O functions, you pass a `resmgr_attr_t` structure to *resmgr_attach()* to specify the attributes of the resource manager.

The `resmgr_attr_t` structure is defined as follows:

```
typedef struct _resmgr_attr {
    unsigned            flags;
    unsigned            nparts_max;
    size_t              msg_max_size;
    int                 (*other_func)(resmgr_context_t *,
                                      void *msg);
    unsigned            reserved[4];
} resmgr_attr_t;
```

The members include:

flags

Lets you change the behavior of the resource manager interface. Set this to 0, or a combination of the following bits (defined in **<sys/dispatch.h>**):

- RESMGR_FLAG_ATTACH_LOCAL — set up the resource manager, but don't register its path with `procnto`. You can send messages to the resource manager's channel (if you know where to find it).
- RESMGR_FLAG_CROSS_ENDIAN — the server handles cross-endian support. The framework handles all necessary conversions on the server's side; the client doesn't have to do anything.

  If necessary, your resource manager can determine that a message came from a client of a different endian-ness by checking to see if the _NTO_MI_ENDIAN_DIFF bit is set in the *flags* member of the `_msg_info` structure that's included in the `resmgr_context_t` structure that's passed to the handler functions.
- RESMGR_FLAG_NO_DEFAULT_FUNC — not implemented.
- RESMGR_FLAG_RCM (QNX Neutrino 6.6 or later) — automatically adopt the client's resource constraint mode when handling a request.

There are also some _RESMGR_FLAG_* bits (with a leading underscore), but you use them in the *flags* argument to *resmgr_attach()*.

nparts_max

The number of components that should be allocated to the IOV array.

msg_max_size

The size of the message buffer.

These members will be important when you start writing your own handler functions.

If you specify a value of zero for *nparts_max*, the resource manager library will bump the value to the minimum usable by the library itself. Why would you want to set the size of the IOV array? As we'll see in the "*Getting the resource manager library to do the reply*" section of the Handling Read and Write Messages chapter, you can tell the resource manager library to do our replying for us. We may want to give it an IOV array that points to *N* buffers containing the reply data. But, since we'll ask the library to do the reply for us, we need to use its IOV array, which of course would need to be big enough to point to our *N* buffers.

other_func

Lets you specify a routine to call in cases where the resource manager gets an I/O message that it doesn't understand.

In general, we don't recommend that you use this member. For private or custom messages, you should use _IO_DEVCTL or _IO_MSG handlers, as described in the *Handling Other Messages* chapter. If you want to receive pulses, use *pulse_attach()*.

If the resource manager library gets an I/O message that it doesn't know how to handle, it'll call the routine specified by the *other_func* member, if non-NULL. (If it's NULL, the resource manager library will return an ENOSYS to the client, effectively stating that it doesn't know what this message means.)

You might specify a non-NULL value for *other_func* in the case where you've specified some form of custom messaging between clients and your resource manager, although the recommended approach for this is the *devctl()* function call (client) and the _IO_DEVCTL message handler (server) or a *MsgSend*()* function call (client) and the _IO_MSG message handler (server).

For non-I/O message types, you should use the *message_attach()* function, which attaches a message range for the dispatch handle. When a message with a type in that range is received, the *dispatch_block()* function calls a user-supplied function that's responsible for doing any specific work, such as replying to the client.

Ways of adding functionality to the resource manager

You can add functionality to the resource manager you're writing in these fundamental ways:

- use the *default functions* encapsulated within your own
- use the *helper functions* within your own
- write the *entire function* yourself

The first two are almost identical, because the default functions really don't do that much by themselves—they rely on the POSIX helper functions. The third approach has advantages and disadvantages.

Using the default functions

Since the default functions (e.g., *iofunc_open_default()*) can be installed in the jump table directly, there's no reason you couldn't embed them within your own functions.

Here's an example of how you would do that with your own *io_open* handler:

```
int main (int argc, char **argv)
{
    …

    /* install all of the default functions */
    iofunc_func_init (_RESMGR_CONNECT_NFUNCS, &connect_funcs,
                      _RESMGR_IO_NFUNCS, &io_funcs);

    /* take over the open function */
    connect_funcs.open = io_open;
    …
}

int
io_open (resmgr_context_t *ctp, io_open_t *msg,
         RESMGR_HANDLE_T *handle, void *extra)
{
    return (iofunc_open_default (ctp, msg, handle, extra));
}
```

Obviously, this is just an incremental step that lets you gain control in your *io_open* handler when the message arrives from the client. You may wish to do something before or after the default function does its thing:

```
/* example of doing something before */

extern int accepting_opens_now;

int
io_open (resmgr_context_t *ctp, io_open_t *msg,
         RESMGR_HANDLE_T *handle, void *extra)
{
    if (!accepting_opens_now) {
```

```
        return (EBUSY);
    }

    /*
     *  at this point, we're okay to let the open happen,
     *  so let the default function do the "work".
     */

    return (iofunc_open_default (ctp, msg, handle, extra));
}
```

Or:

```
/* example of doing something after */

int
io_open (resmgr_context_t *ctp, io_open_t *msg,
         RESMGR_HANDLE_T *handle, void *extra)
{
    int     sts;

    /*
     * have the default function do the checking
     * and the work for us
     */

    sts = iofunc_open_default (ctp, msg, handle, extra);

    /*
     *  if the default function says it's okay to let the open
     *  happen, we want to log the request
     */

    if (sts == EOK) {
        log_open_request (ctp, msg);
    }
    return (sts);
}
```

It goes without saying that you can do something before *and* after the standard default POSIX handler.

The principal advantage of this approach is that you can add to the functionality of the standard default POSIX handlers with very little effort.

Using the helper functions

The default functions make use of *helper* functions. These functions can't be placed directly into the connect or I/O jump tables, but they do perform the bulk of the work.

Here's the source for the *iofunc_chmod_default()* function:

```
int
iofunc_chmod_default (resmgr_context_t *ctp, io_chmod_t *msg,
                      iofunc_ocb_t *ocb)
{
```

```
    return (iofunc_chmod (ctp, msg, ocb, ocb -> attr));
}
```

Notice how the *iofunc_chmod()* helper function performs all the work for the *iofunc_chmod_default()* default handler. This is typical for the simple functions.

The *iofunc_stat_default()* default handler is a more interesting case:

```
int iofunc_stat_default(resmgr_context_t *ctp, io_stat_t *msg, iofunc_ocb_t *ocb) {
    unsigned        len;

    /*  Update stale time fields (ctime, mtime, atime) this OCB. */
    (void)iofunc_time_update(ocb->attr);

    unsigned format = msg->i.format;
    int const status = iofunc_stat_format(ctp, ocb->attr, &format, &msg->o, &len);
    if(status != EOK) {
        return(status);
    }
    _RESMGR_STATUS(ctp, (long)format);
    return _RESMGR_PTR(ctp, &msg->o, len);
}
```

This handler calls two helper routines. First it calls *iofunc_time_update()* to ensure that all of the time fields (*atime*, *ctime* and *mtime*) are up to date. Then it calls *iofunc_stat_format()*, which builds the reply. The default handler then sets the status (to be returned via *MsgReply*) to the form of the status information. Finally, the default function builds a pointer in the *ctp* structure and returns -1, to indicate to the resource manager library that it should return one part from the *ctp->iov* structure to the client.

The most complicated handling is done by the *iofunc_open_default()* handler:

```
int
iofunc_open_default (resmgr_context_t *ctp, io_open_t *msg,
                     iofunc_attr_t *attr, void *extra)
{
    int     status;

    iofunc_attr_lock (attr);

    if ((status = iofunc_open (ctp, msg, attr, 0, 0)) != EOK) {
        iofunc_attr_unlock (attr);
        return (status);
    }

    if ((status = iofunc_ocb_attach (ctp, msg, 0, attr, 0))
        != EOK) {
        iofunc_attr_unlock (attr);
        return (status);
    }

    iofunc_attr_unlock (attr);
    return (EOK);
}
```

This handler calls four helper functions:

1. It calls *iofunc_attr_lock()* to lock the attribute structure so that it has exclusive access to it (it's going to be updating things like the counters, so we need to make sure no one else is doing that at the same time).
2. It then calls the helper function *iofunc_open()*, which does the actual verification of the permissions.
3. Next it calls *iofunc_ocb_attach()* to bind an OCB to this request, so that it will get automatically passed to all of the I/O functions later.
4. Finally, it calls *iofunc_attr_unlock()* to release the lock on the attribute structure.

Writing the entire function yourself

Sometimes a default function will be of no help for your particular resource manager.

For example, *iofunc_read_default()* and *iofunc_write_default()* functions implement **/dev/null**; they do all the work of returning 0 bytes (EOF) or swallowing all the message bytes (respectively). You'll want to do something in those handlers (unless your resource manager doesn't support the _IO_READ or _IO_WRITE messages).

Note that even in such cases, there are still helper functions you can use, such as *iofunc_read_verify()* and *iofunc_write_verify()*.

Here's a sample skeleton for a typical filesystem, in pseudo-code, to illustrate the steps that need to be taken to handle an open request for a file:

```
if the open request is for a path (i.e., multiple
  directory levels)
    call iofunc_client_info_ext to get information
      about client
    for each directory component
        call iofunc_check_access to check execute
          permission for access
        /*
         recall that execute permission on a
         directory is really the "search"
         permission for that directory
        */
    next
    /*
     at this point you have verified access
     to the target
    */
endif

if O_CREAT is set and the file doesn't exist
    call iofunc_open, passing the attribute of the
      parent as dattr
    if the iofunc_open succeeds,
        do the work to create the new inode,
          or whatever
    endif
else
    call iofunc_open, passing the attr of the file
     and NULL for dattr
```

```
endif

/*
 at this point, check for things like o_trunc,
 etc. -- things that you have to do for the attr
*/

call iofunc_ocb_attach
return EOK
```

For a device (i.e., *resmgr_attach()* didn't specify that the managed resource is a directory), the following steps apply:

```
/*
 at startup time (i.e., in the main() of the
 resource manager)
*/
call iofunc_attr_init to initialize an attribute
  structure

/* in the io_open message handler: */
call iofunc_open, passing in the attribute of the
  device and NULL for dattr

call iofunc_ocb_attach
return EOK
```

A resource manager's response to an *open()* request isn't always a yes-or-no answer. It's possible to return a connect message indicating that the server would like some other action taken. For example, if the open occurs on a path that represents a symbolic link to some other path, the server could respond using the *_IO_SET_CONNECT_RET()* macro and the _IO_CONNECT_RET_LINK value.

For example, an open handler that only redirects pathnames might look something like:

```
io_open(resmgr_context_t *ctp, io_open_t *msg,
        iofunc_attr_t *dattr, void *extra) {
    char *newpath;

    /* Do all the error/access checking ... */

    /* Lookup the redirected path and store
       the new path in 'newpath' */
    newpath = get_a_new_path(msg->connect.path);

    _IO_SET_CONNECT_RET(ctp, _IO_CONNECT_RET_LINK);
    len = strlen(newpath) + 1;

    msg->link_reply.eflag = msg->connect.eflag;
    msg->link_reply.nentries = 0;
    msg->link_reply.path_len = len;
    strcpy((char *)(msg->link_reply + 1), newpath);

    len += sizeof(msg->link_reply);
```

```
    return(_RESMGR_PTR(ctp, &msg->link_reply, len));
}
```

In this example, we use the macro *_IO_SET_CONNECT_RET()* (defined in **<sys/iomsg.h>**) to set the *ctp->status* field to _IO_CONNECT_RET_LINK. This value indicates to the resource-manager framework that the return value isn't actually a simple return code, but a new request to be processed.

The path for this new request follows directly after the *link_reply* structure and is *path_len* bytes long. The final few lines of the code just stuff an IOV with the reply message (and the new path to be queried) and return to the resource-manager framework.

Security

A resource manager is usually a privileged process, so you should be careful not to let a client coerce it into exhausting resources or compromising the system.

When you're designing your resource manager, you should consider the following:

The permissions on the resource manager's entry in the pathname space

You specify these permissions as an argument to *iofunc_attr_init()*. In general, there isn't a "correct" set of permissions to use; you should restrict them according to what you want other processes and users to be able to do with your resource manager.

Running as root

A resource manager typically needs to be started by **root** in order to attach to the pathname space, but it's a good idea to use *procmgr_ability()* to retain the abilities that the resource manager needs, and then run as a non-**root** user. For more information, see "Process privileges" in the QNX Neutrino *Programmer's Guide*.

Requests to allocate resources on behalf of a client

If the resource manager isn't a critical process that needs to be free of any resource constraint thresholds, it can simply run in constrained mode (see "Resource constraint thresholds" in the QNX Neutrino *Programmer's Guide*).

If the resource manager is a critical process, it should keep the PROCMGR_AID_RCONSTRAINT ability (see *procmgr_ability()*), but it then needs to ensure that constrained clients don't use it to allocate a resource in excess of the currently defined threshold. Unless the resource manager is managing the resource itself, compliance generally means adopting the client's constraint mode when handling a request, in one of the following ways:

- Automatically, by setting RESMGR_FLAG_RCM in the `resmgr_attr_t` structure (see the entry for *resmgr_attach()* in the QNX Neutrino *C Library Reference*):

```
resmgr_attr.flags |= RESMGR_FLAG_RCM;
resmgr_attach(dpp, &resmgr_attr, name, _FTYPE_ANY, 0, &connect_funcs,
              &io_funcs, &io_attr))
```

- Manually, by checking for _NTO_MI_CONSTRAINED in the *flags* member of the `_msg_info` structure, available from a call to *MsgReceive()* or *MsgInfo()*. If this bit is set, the message was received from a constrained client; when the resource manager allocates resources on behalf of such a client, it should constrain itself using *ThreadCtl()* with the _NTO_TCTL_RCM_GET_AND_SET command:

```
int value = 1; // 1 to constrain, 0 to remove constraint
ThreadCtl(_NTO_TCTL_RCM_GET_AND_SET, &value); /* swaps current state with value */

/* Handle the request... */

ThreadCtl(_NTO_TCTL_RCM_GET_AND_SET, &value); /* restores original state */
```

When a resource manager runs as a constrained process or constrains one of its threads, resource allocation requests fail when there are still resources available. It should handle these failures in the same way it would handle a failure caused by complete exhaustion of resources, generally by returning an error to the client. If the resource manager can continue to process messages, it should do so, for the sake of overall system stability.

Checking a client's abilities

You can make sure that a client has the appropriate abilities by calling *ConnectClientInfoAble()* or *iofunc_client_info_able()*. Both of these take as an argument a list of abilities; if the client doesn't have all the required abilities, these functions set _NTO_CI_UNABLE in the *flags* member of the `_client_info` structure.

If you've called one of these functions, *iofunc_check_access()* returns EACCES if _NTO_CI_UNABLE is set.

Your resource manager can create custom abilities by calling *procmgr_ability_create()*; a client can get identifiers for them by calling *procmgr_ability_lookup()*, and then call *procmgr_ability()* to retain them before it switches to a non-**root** user ID. For more information, see "Creating abilities" in the QNX Neutrino *Programmer's Guide*. When you check a client's abilities, you can include a combination of PROCMGR_AID_* abilities and custom ones.

The resource manager library creates the following custom abilities:

Ability ID	Ability name	Description
IOFUNC_ABILITYID_CHOWN	`iofunc/chown` (IOFUNC_ABILITY_CHOWN)	Allow the client to set the ownership of files, even if not **root**
IOFUNC_ABILITYID_DUP	`iofunc/dup` (IOFUNC_ABILITY_DUP)	Allow the client to duplicate another process's handle
IOFUNC_ABILITYID_EXEC	`iofunc/exec` (IOFUNC_ABILITY_EXEC)	Grant execute access to files and directories that the client wouldn't normally have access to
IOFUNC_ABILITYID_READ	`iofunc/read` (IOFUNC_ABILITY_READ)	Allow the client to access files for reading, even if it doesn't have the required permissions

Chapter 4
POSIX-Layer Data Structures

The resource manager library defines (in **<sys/iofunc.h>**) several key structures that are related to the POSIX-layer support routines:

iofunc_ocb_t (Open Control Block) **structure**

Contains per-*open* data, such as the current position into a file (the *lseek()* offset).

iofunc_attr_t (attribute) **structure**

Since a resource manager may be responsible for more than one device (e.g., `devc-ser*` may be responsible for **/dev/ser1**, **/dev/ser2**, **/dev/ser3**, etc.), the *attributes* structure holds data on a per-*name* basis. This structure contains such items as the user and group ID of the owner of the device, the last modification time, etc.

iofunc_mount_t (mount) **structure**

Contains per-*mountpoint* data items that are global to the entire mount device. Filesystem (block I/O device) managers use this structure; a resource manager for a device typically won't have a mount structure.

Be sure to `#include` **<sys/iofunc.h>** before **<sys/resmgr.h>**, or else the data structures won't be defined properly.

This picture may help explain their interrelationships:

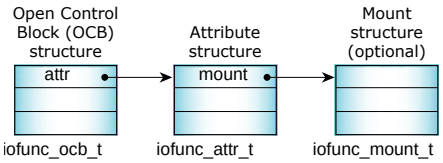

Figure 4: A resource manager is responsible for three data structures.

If three clients open two paths associated with a resource manager, the data structures are linked like this:

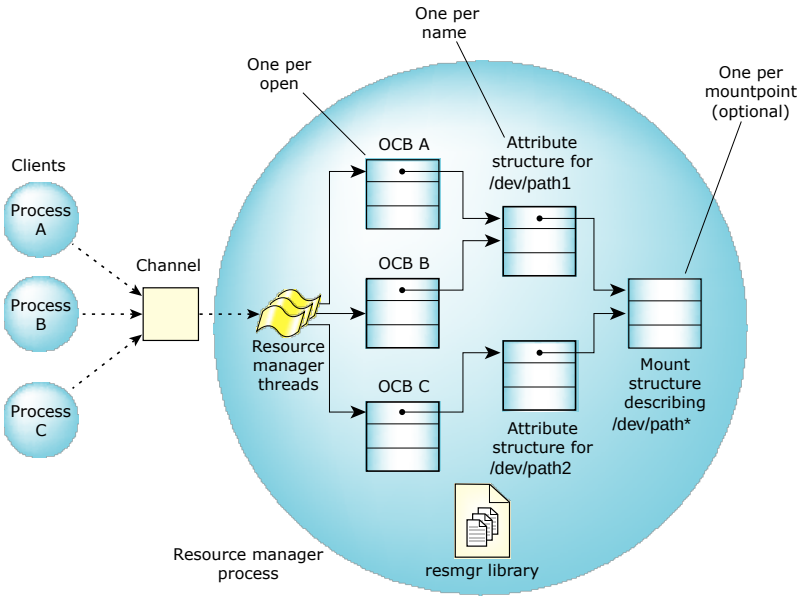

Figure 5: Multiple clients with multiple OCBs, all linked to one mount structure.

The `iofunc_ocb_t` (Open Control Block) structure

The Open Control Block (OCB) maintains the state information about a particular session involving a client and a resource manager. It's created during *open()* handling and exists until a *close()* is performed.

This structure is used by the iofunc layer helper functions. (In the *Extending the POSIX-Layer Data Structures* chapter, we'll show you how to extend this to include your own data).

The OCB structure contains at least the following:

```
typedef struct _iofunc_ocb {
    IOFUNC_ATTR_T   *attr;
    int32_t         ioflag;
    off_t           offset;
    uint16_t        sflag;
    uint16_t        flags;
} iofunc_ocb_t;
```

where the values represent:

attr

A pointer to the *attribute structure* (see below).

ioflag

Contains the mode (e.g., reading, writing, blocking) that the resource was opened with. This information is inherited from the `io_connect_t` structure that's available in the message passed to the *io_open* handler. The open modes (as passed to *open()* on the client side) are converted to the *ioflag* values as follows:

Open mode	*ioflag* value
O_RDONLY	_IO_FLAG_RD
O_RDWR	_IO_FLAG_RD | _IO_FLAG_WR
O_WRONLY	_IO_FLAG_WR

offset

The read/write offset into the resource (e.g., our current *lseek()* position within a file). Your resource manager can modify this member.

sflag

Defines the sharing mode. This information is inherited from the `io_connect_t` structure that's available in the message passed to the *io_open* handler.

flags

A combination of zero or more of the following bits:

- IOFUNC_OCB_PRIVILEGED — a privileged process (i.e., **root**) performed the *open()*

- IOFUNC_OCB_MMAP — this OCB is in use by a *mmap()* call on the client side
- IOFUNC_OCB_MMAP_UNIQUE — a hint for the *mmap()* handler to provide a unique mapping

Additionally, you can use flags in the range defined by IOFUNC_OCB_FLAGS_PRIVATE (see **<sys/iofunc.h>**) for your own purposes. Your resource manager can modify these flags.

The `iofunc_attr_t` (attribute) structure

The `iofunc_attr_t` structure defines the characteristics of the device that you're supplying the resource manager for. This is used in conjunction with the OCB structure.

The attribute structure contains at least the following:

```
typedef struct _iofunc_attr {
    IOFUNC_MOUNT_T            *mount;
    uint32_t                  flags;
    int32_t                   lock_tid;
    uint16_t                  lock_count;
    uint16_t                  count;
    uint16_t                  rcount;
    uint16_t                  wcount;
    uint16_t                  rlocks;
    uint16_t                  wlocks;
    struct _iofunc_mmap_list  *mmap_list;
    struct _iofunc_lock_list  *lock_list;
    void                      *list;
    uint32_t                  list_size;
    off_t                     nbytes;
    ino_t                     inode;
    uid_t                     uid;
    gid_t                     gid;
    time_t                    mtime;
    time_t                    atime;
    time_t                    ctime;
    mode_t                    mode;
    nlink_t                   nlink;
    dev_t                     rdev;
    unsigned                  mtime_ns;
    unsigned                  atime_ns;
    unsigned                  ctime_ns;
} iofunc_attr_t;
```

The fields include:

mount

A pointer to the *mount structure* (see below).

flags

The bit-mapped *flags* member can contain the following flags:

IOFUNC_ATTR_ATIME

The access time is no longer valid. Typically set on a read from the resource.

IOFUNC_ATTR_CTIME

The change of status time is no longer valid. Typically set on a file info change.

IOFUNC_ATTR_DIRTY_NLINK

The number of links has changed.

IOFUNC_ATTR_DIRTY_MODE

The mode has changed.

IOFUNC_ATTR_DIRTY_OWNER

The uid or the gid has changed.

IOFUNC_ATTR_DIRTY_RDEV

The *rdev* member has changed, e.g., *mknod()*.

IOFUNC_ATTR_DIRTY_SIZE

The size has changed.

IOFUNC_ATTR_DIRTY_TIME

One or more of *mtime*, *atime*, or *ctime* has changed.

IOFUNC_ATTR_MTIME

The modification time is no longer valid. Typically set on a write to the resource.

IOFUNC_ATTR_NS_TIMESTAMPS

(QNX Neutrino 7.0 or later) The attributes structure includes the fields used for nanosecond-resolution timestamps, *mtime_ns*, *atime_ns*, and *ctime_ns*.

Since your resource manager uses these flags, you can tell right away which fields of the attribute structure have been modified by the various iofunc-layer helper routines. That way, if you need to write the entries to some medium, you can write just those that have changed. The user-defined area for flags is IOFUNC_ATTR_PRIVATE (see **<sys/iofunc.h>**).

For details on updating your attribute structure, see the section on "*Updating the time for reads and writes*" in the Handling Read and Write Messages chapter.

lock_tid* and *lock_count

To support multiple threads in your resource manager, you'll need to lock the attribute structure so that only one thread at a time is allowed to change it. The resource manager layer automatically locks the attribute (using *iofunc_attr_lock()*) for you when certain handler functions are called (i.e., IO_*).

The *lock_tid* member holds the thread ID; the *lock_count* member holds the number of times the thread has locked the attribute structure. For more information, see the *iofunc_attr_lock()* and *iofunc_attr_unlock()* functions in the QNX Neutrino *C Library Reference*.)

count*, *rcount*, *wcount*, *rlocks* and *wlocks

Several counters are stored in the attribute structure and are incremented/decremented by some of the iofunc layer helper functions. Both the functionality and the actual contents of the message received from the client determine which specific members are affected.

This counter:	Tracks the number of:
count	OCBs using this attribute in any manner. When this count goes to zero, it means that no one is using this attribute.
rcount	OCBs using this attribute for reading
wcount	OCBs using this attribute for writing
rlocks	Read locks currently registered on the attribute
wlocks	Write locks currently registered on the attribute

These counts aren't exclusive. For example, if an OCB has specified that the resource is opened for reading and writing, then *count*, *rcount*, and *wcount* are *all* incremented. (See the *iofunc_attr_init()*, *iofunc_lock_default()*, *iofunc_lock()*, *iofunc_ocb_attach()*, and *iofunc_ocb_detach()* functions.)

mmap_list* and *lock_list

To manage their particular functionality on the resource, the *mmap_list* member is used by the *iofunc_mmap()* and *iofunc_mmap_default()* functions; the *lock_list* member is used by the *iofunc_lock_default()* function. Generally, you shouldn't need to modify or examine these members.

list

Reserved for future use.

list_size

The size of the reserved *list* area; reserved for future use.

nbytes

The number of bytes in the resource. Your resource manager can modify this member. For a file, this would contain the file's size. For special devices (e.g., **/dev/null**) that don't support *lseek()* or have a radically different interpretation for *lseek()*, this field isn't used (because you wouldn't use any of the helper functions, but would supply your own instead.) In these cases, we recommend that you set this field to zero, unless there's a meaningful interpretation that you care to put to it.

inode

This is a mountpoint-specific inode that must be unique per mountpoint. You can specify your own value, or 0 to have the process manager fill it in for you. For filesystem resource managers, this may correspond to some on-disk structure. In any case, the interpretation of this field is up to you.

uid* and *gid

The user ID and group ID of the owner of this resource. These fields are updated automatically by the *chown()* helper functions (e.g., *iofunc_chown_default()*) and are referenced in

conjunction with the *mode* member for access-granting purposes by the *open()* help functions (e.g., *iofunc_open_default()*).

mtime*, *atime*, and *ctime

The three POSIX time members:

- *mtime* — modification time (*write()* updates this)
- *atime* — access time (*read()* updates this)
- *ctime* — change of status time (*write()*, *chmod()*, and *chown()* update this)

One or more of the time members may be *invalidated* as a result of calling an iofunc-layer function. This is to avoid having each and every I/O message handler go to the kernel and request the current time of day, just to fill in the attribute structure's time member(s).

POSIX states that these times must be *valid* when the *fstat()* is performed, but they don't have to reflect the *actual* time that the associated change occurred. Also, the times must change between *fstat()* invocations *if* the associated change occurred between *fstat()* invocations. If the associated change never occurred between *fstat()* invocations, then the time returned should be the same as returned last time. Furthermore, if the associated change occurred multiple times between *fstat()* invocations, then the time need only be different from the previously returned time.

There's a helper function that fills the members with the correct time; you may wish to call it in the appropriate handlers to keep the time up-to-date on the device—see the *iofunc_time_update()* function.

mode

Contains the resource's mode (e.g., type, permissions). Valid modes may be selected from the S_* series of constants in **<sys/stat.h>**; see `struct stat`.

nlink

The number of links to this particular name. For names that represent a directory, this value must be at least 2 (one for the directory itself, one for the **./** entry in it). Your resource manager can modify this member.

rdev

Contains the device number for a character special device and the **rdev** number for a named special device.

mtime_ns*, *atime_ns*, and *ctime_ns

(QNX Neutrino 7.0 or later) The nanosecond values for the POSIX time members, *mtime*, *atime*, and *ctime*.

These fields are included if you compile for a 64-bit architecture, or if you define IOFUNC_NS_TIMESTAMP_SUPPORT before including **<sys/iofunc.h>** when you compile for a 32-bit architecture.

The optional `iofunc_mount_t` (mount) structure

The members of the mount structure, specifically the *conf* and *flags* members, modify the behavior of some of the iofunc layer functions.

If you need to modify any members of the mount structure, use *iofunc_mount_init()* to initialize it.

For example, your program saves files to a file system. You can use *iofunc_mount_init()* and *iofunc_mount_set_time()* to set a timestamp resolution for open files that matches the resolution the file system uses (a POSIX requirement).

The iofunc_mount_t structure is optional and contains at least the following:

```
typedef struct _iofunc_mount {
    uint32_t            flags;
    uint32_t            conf;
    dev_t               dev;
    int32_t             blocksize;
    iofunc_funcs_t      *funcs;
    void                *power;

    size_t              size;
    uint32_t            ex_flags;
    uint32_t            timeres;
    uint64_t            reserved[4];
} iofunc_mount_t;
```

The variables are:

flags

Contains one relevant bit (manifest constant IOFUNC_MOUNT_32BIT), which indicates that the offsets used by this resource manager are 32-bit (as opposed to the extended 64-bit offsets). The user-modifiable mount flags are defined as IOFUNC_MOUNT_FLAGS_PRIVATE (see **<sys/iofunc.h>**).

conf

Contains several bits:

IOFUNC_PC_CHOWN_RESTRICTED

Causes the default handler for the _IO_CHOWN message to behave in a manner defined by POSIX as "`chown`-restricted".

IOFUNC_PC_NO_TRUNC

Has no effect on the iofunc layer libraries, but is returned by the iofunc layer's default _IO_PATHCONF handler.

IOFUNC_PC_SYNC_IO

Indicates that the filesystem supports synchronous I/O operations. If this bit isn't set, the following may occur:

- The default iofunc layer _IO_OPEN handler, *iofunc_open_default()*, fails if the client specifies O_DSYNC, O_RSYNC, or O_SYNC.
- The *iofunc_sync_verify()* function returns EINVAL.
- Attempts to set O_DSYNC, O_RSYNC, or O_SYNC with *fcntl()* or the DCMD_ALL_SETFLAGS *devctl()* command fail.

IOFUNC_PC_LINK_DIR

Controls whether or not **root** is allowed to link and unlink directories.

IOFUNC_PC_ACL

Indicates whether or not the resource manager supports access control lists. For more information about ACLs, see Working with Access Control Lists (ACLs) in the QNX Neutrino *Programmer's Guide*.

IOFUNC_PC_EXT

Set automatically when *iofunc_mount_init()* is called to set the `iofunc_mount_t` structure.

Note that the options mentioned above for the *conf* member are returned by the iofunc layer _IO_PATHCONF default handler.

dev

Contains the device number for the filesystem. This number is returned to the client's *stat()* function in the `struct stat` *st_dev* member.

blocksize

Contains the block size of the device. On filesystem types of resource managers, this indicates the native blocksize of the disk, e.g., 512 bytes.

funcs

Contains the following structure:

```
struct _iofunc_funcs {
   unsigned     nfuncs;
   IOFUNC_OCB_T *(*ocb_calloc) (resmgr_context_t *ctp,
                                IOFUNC_ATTR_T *attr);
   void         (*ocb_free) (IOFUNC_OCB_T *ocb);
   int          (*attr_lock)(IOFUNC_ATTR_T *attr);
   int          (*attr_unlock)(IOFUNC_ATTR_T *attr);
   int          (*attr_trylock)(IOFUNC_ATTR_T *attr);

};
```

where:

nfuncs

Indicates the number of functions present in the structure; you should fill it with the manifest constant _IOFUNC_NFUNCS.

ocb_calloc()* and *ocb_free()

Allows you to override the OCBs on a per-mountpoint basis (see "*Extending the OCB and attribute structures*" in the Extending the POSIX-Layer Data Structures chapter). If these members are NULL, then the default library versions (*iofunc_ocb_calloc()* and *iofunc_ocb_free()*) are used. You must specify either *both* or *neither* of these functions; they operate as a matched pair.

attr_lock()*, *attr_unlock()* and *attr_trylock()

Allows you to create customized versions of the functions that lock, unlock, and attempt to lock the attribute structure.

power

Reserved for future use.

size

Contains the size of the `iofunc_mount_t` function. Set by *iofunc_mount_init()*.

ext_flags

Extends the range of possible flags values.

timeres

Contains the timestamp resolution for iofunc functions and structures. Set by *iofunc_mount_set_time()*.

reserved

Reserved for future use.

Chapter 5
Handling Read and Write Messages

Handling the _IO_READ message

The *io_read* handler is responsible for returning data bytes to the client after receiving an _IO_READ message. Examples of functions that send this message are *read()*, *readdir()*, *fread()*, and *fgetc()*. Let's start by looking at the format of the message itself:

```
struct _io_read {
    uint16_t            type;
    uint16_t            combine_len;
    uint32_t            nbytes;
    uint32_t            xtype;
    uint32_t            zero;
};

struct _io_read64 {
    uint16_t            type;
    uint16_t            combine_len;
    uint32_t            nbytes;
    uint32_t            xtype;
    uint32_t            nbytes_hi;
};

typedef union {
    struct _io_read     i;
    struct _io_read     i64;
    /* unsigned char    data[nbytes];    */
    /* nbytes is returned with MsgReply  */
} io_read_t;
```

As with all resource manager messages, we've defined a `union` that contains the input (coming into the resource manager) structure and a reply or output (going back to the client) structure. The *io_read* handler is prototyped with an argument of `io_read_t *`*msg*—that's the pointer to the union containing the message.

Since this is a *read()*, the *type* member has the value _IO_READ or _IO_READ64. The client library uses the _IO_READ64 form only when the length is greater than 4 GB. The items of interest in the input structure are:

combine_len

This field has meaning for a combine message—see the *Combine Messages* chapter.

nbytes

How many bytes the client is expecting. For an _IO_READ64 message, the high 32 bits of the length are in *nbytes_hi*.

xtype

A per-message override, if your resource manager supports it. Even if your resource manager doesn't support it, you should still examine this member. More on the *xtype* later (see the "*Handling the xtype member*" section).

nbytes_hi

(_IO_READ64 only) The high 32 bits of the length.

You can use the *_IO_READ_GET_NBYTES()* macro (defined in **<sys/iofunc.h>**) to determine the number of bytes. You pass it a pointer to the message:

```
num_bytes = _IO_READ_GET_NBYTES(msg);
```

We'll create an *io_read* handler that actually returns some data (the fixed string "`Hello, world\n`"). We'll use the OCB to keep track of our position within the buffer that we're returning to the client.

When we get the _IO_READ message, the *nbytes* member tells us exactly how many bytes the client wants to read. Suppose that the client issues:

```
read (fd, buf, 4096);
```

In this case, it's a simple matter to return our entire "`Hello, world\n`" string in the output buffer and tell the client that we're returning 13 bytes, i.e., the size of the string.

However, consider the case where the client is performing the following:

```
while (read (fd, &character, 1) != EOF) {
    printf ("Got a character \"%c\"\n", character);
}
```

Granted, this isn't a terribly efficient way for the client to perform reads! In this case, we would get `msg->i.nbytes` set to 1 (the size of the buffer that the client wants to get). We can't simply return the entire string all at once to the client—we have to hand it out one character at a time. This is where the OCB's *offset* member comes into play.

Sample code for handling _IO_READ messages

Here's a complete *io_read* handler that correctly handles these cases:

```
#include <errno.h>
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int io_read (resmgr_context_t *ctp, io_read_t *msg, RESMGR_OCB_T *ocb);

static char                     *buffer = "Hello world\n";

static resmgr_connect_funcs_t   connect_funcs;
static resmgr_io_funcs_t        io_funcs;
static iofunc_attr_t            attr;

int main(int argc, char **argv)
{
    /* declare variables we'll be using */
    resmgr_attr_t        resmgr_attr;
    dispatch_t           *dpp;
    dispatch_context_t   *ctp;
    int                  id;
```

```
    /* initialize dispatch interface */
    if((dpp = dispatch_create()) == NULL) {
        fprintf(stderr, "%s: Unable to allocate dispatch handle.\n",
                argv[0]);
        return EXIT_FAILURE;
    }

    /* initialize resource manager attributes */
    memset(&resmgr_attr, 0, sizeof resmgr_attr);
    resmgr_attr.nparts_max = 1;
    resmgr_attr.msg_max_size = 2048;

    /* initialize functions for handling messages, including
       our read handlers */
    iofunc_func_init(_RESMGR_CONNECT_NFUNCS, &connect_funcs,
                     _RESMGR_IO_NFUNCS, &io_funcs);
    io_funcs.read = io_read;
    io_funcs.read64 = io_read;

    /* initialize attribute structure used by the device */
    iofunc_attr_init(&attr, S_IFNAM | 0666, 0, 0);
    attr.nbytes = strlen(buffer)+1;

    /* attach our device name */
    if((id = resmgr_attach(dpp, &resmgr_attr, "/dev/sample", _FTYPE_ANY, 0,
                &connect_funcs, &io_funcs, &attr)) == -1) {
        fprintf(stderr, "%s: Unable to attach name.\n", argv[0]);
        return EXIT_FAILURE;
    }

    /* allocate a context structure */
    ctp = dispatch_context_alloc(dpp);

    /* start the resource manager message loop */
    while(1) {
        if((ctp = dispatch_block(ctp)) == NULL) {
            fprintf(stderr, "block error\n");
            return EXIT_FAILURE;
        }
        dispatch_handler(ctp);
    }
    return EXIT_SUCCESS;
}

int
io_read (resmgr_context_t *ctp, io_read_t *msg, RESMGR_OCB_T *ocb)
{
    size_t      nleft;
    size_t      nbytes;
    int         nparts;
    int         status;

    if ((status = iofunc_read_verify (ctp, msg, ocb, NULL)) != EOK)
        return (status);

    if ((msg->i.xtype & _IO_XTYPE_MASK) != _IO_XTYPE_NONE)
        return (ENOSYS);

    /*
     *  On all reads (first and subsequent), calculate
     *  how many bytes we can return to the client,
     *  based upon the number of bytes available (nleft)
     *  and the client's buffer size
     */

    nleft = ocb->attr->nbytes - ocb->offset;
    nbytes = min (_IO_READ_GET_NBYTES(msg), nleft);

    if (nbytes > 0) {
        /* set up the return data IOV */
        SETIOV (ctp->iov, buffer + ocb->offset, nbytes);
```

```
        /* set up the number of bytes (returned by client's read()) */
        _IO_SET_READ_NBYTES (ctp, nbytes);

        /*
         * advance the offset by the number of bytes
         * returned to the client.
         */

        ocb->offset += nbytes;

        nparts = 1;
    } else {
        /*
         * they've asked for zero bytes or they've already previously
         * read everything
         */

        _IO_SET_READ_NBYTES (ctp, 0);

        nparts = 0;
    }

    /* mark the access time as invalid (we just accessed it) */

    if (msg->i.nbytes > 0)
        ocb->attr->flags |= IOFUNC_ATTR_ATIME;

    return (_RESMGR_NPARTS (nparts));
}
```

The *ocb* maintains our context for us by storing the *offset* field, which gives us the position within the *buffer*, and by having a pointer to the attribute structure *attr*, which tells us how big the buffer actually is via its *nbytes* member.

Of course, we had to give the resource manager library the address of our *io_read* handler so that it knew to call it. So the code in *main()* where we had called *iofunc_func_init()* became:

```
/* initialize functions for handling messages */
iofunc_func_init(_RESMGR_CONNECT_NFUNCS, &connect_funcs,
                 _RESMGR_IO_NFUNCS, &io_funcs);
io_funcs.read = io_read;
```

We also needed to add the following to the area above *main()*:

```
#include <errno.h>
#include <unistd.h>

int io_read (resmgr_context_t *ctp, io_read_t *msg, RESMGR_OCB_T *ocb);

static char *buffer = "Hello world\n";"
```

Where did the attribute structure's *nbytes* member get filled in? In *main()*, just after we did the *iofunc_attr_init()*. We modified *main()* slightly:

After this line:

```
iofunc_attr_init (&attr, S_IFNAM | 0666, 0, 0);
```

we added this one:

```
attr.nbytes = strlen (buffer)+1;
```

At this point, if you were to run the resource manager (our simple resource manager used the name **/dev/sample**), you could do:

```
# cat /dev/sample
Hello, world
```

The return line `(_RESMGR_NPARTS(nparts))` tells the resource manager library to:

- reply to the client for us
- reply with *nparts* IOVs

Where does it get the IOV array? It's using `ctp->iov`. That's why we first used the *SETIOV()* macro to make `ctp->iov` point to the data to reply with.

If we had no data, as would be the case of a read of zero bytes, then we'd do a return `(_RESMGR_NPARTS(0))`. But *read()* returns with the number of bytes successfully read. Where did we give it this information? That's what the *_IO_SET_READ_NBYTES()* macro was for. It takes the *nbytes* that we give it and stores it in the context structure (*ctp*). Then when we return to the library, the library takes this *nbytes* and passes it as the second parameter to the *MsgReplyv()*. The second parameter tells the kernel what the *MsgSend()* should return. And since the *read()* function is calling *MsgSend()*, that's where it finds out how many bytes were read.

We also update the access time for this device in the read handler. For details on updating the access time, see the section on "*Updating the time for reads and writes*" below.

Handling the _IO_WRITE message

The *io_write* handler is responsible for writing data bytes to the media after receiving a client's _IO_WRITE message. Examples of functions that send this message are *write()* and *fflush()*. Here's the message:

```
struct _io_write {
    uint16_t            type;
    uint16_t            combine_len;
    uint32_t            nbytes;
    uint32_t            xtype;
    uint32_t            zero;
    /* unsigned char    data[nbytes]; */
};

struct _io_write64 {
    uint16_t            type;
    uint16_t            combine_len;
    uint32_t            nbytes;
    uint32_t            xtype;
    uint32_t            nbytes_hi;
    /* unsigned char    data[nbytes]; */
};

typedef union {
    struct _io_write    i;
    struct _io_write    i64;
    /*  nbytes is returned with MsgReply  */
} io_write_t;
```

As with the `io_read_t`, we have a union of an input and an output message, with the output message being empty (the number of bytes actually written is returned by the resource manager library directly to the client's *MsgSend()*).

You can use the *_IO_WRITE_GET_NBYTES()* macro (defined in **<sys/iofunc.h>**) to determine the number of bytes. You pass it a pointer to the message:

```
num_bytes = _IO_WRITE_GET_NBYTES(msg);
```

The data being written by the client almost always follows the header message stored in `struct _io_write`. The exception is if the write was done using *pwrite()* or *pwrite64()*. More on this when we discuss the *xtype* member.

To access the data, we recommend that you reread it into your own buffer. Let's say you had a buffer called *inbuf* that was “big enough” to hold all the data you expected to read from the client (if it isn't big enough, you'll have to read the data piecemeal).

Sample code for handling _IO_WRITE messages

The following is a code snippet that can be added to one of the simple resource manager examples. It prints out whatever it's given (making the assumption that it's given only character text):

```
int
io_write (resmgr_context_t *ctp, io_write_t *msg, RESMGR_OCB_T *ocb)
{
    int     status;
    char    *buf;
    size_t  nbytes;

    if ((status = iofunc_write_verify(ctp, msg, ocb, NULL)) != EOK)
        return (status);

    if ((msg->i.xtype & _IO_XTYPE_MASK) != _IO_XTYPE_NONE)
        return(ENOSYS);

    /* Extract the length of the client's message. */
    nbytes = _IO_WRITE_GET_NBYTES(msg);

    /* Filter out malicious write requests that attempt to write more
       data than they provide in the message. */
    if(nbytes > (size_t)ctp->info.srcmsglen - (size_t)ctp->offset - sizeof(io_write_t)) {
        return EBADMSG;
    }

    /* set up the number of bytes (returned by client's write()) */
    _IO_SET_WRITE_NBYTES (ctp, nbytes);

    buf = (char *) malloc(nbytes + 1);
    if (buf == NULL)
        return(ENOMEM);

    /*
     *  Reread the data from the sender's message buffer.
     *  We're not assuming that all of the data fit into the
     *  resource manager library's receive buffer.
     */

    resmgr_msgread(ctp, buf, nbytes, sizeof(msg->i));
    buf [nbytes] = '\0'; /* just in case the text is not NULL terminated */
    printf ("Received %zu bytes = '%s'\n", nbytes, buf);
    free(buf);

    if (nbytes > 0)
        ocb->attr->flags |= IOFUNC_ATTR_MTIME | IOFUNC_ATTR_CTIME;

    return (_RESMGR_NPARTS (0));
}
```

Of course, we'll have to give the resource manager library the address of our *io_write* handler so that it'll know to call it. In the code for *main()* where we called *iofunc_func_init()*, we'll add a line to register our *io_write* handler:

```
/* initialize functions for handling messages */
iofunc_func_init(_RESMGR_CONNECT_NFUNCS, &connect_funcs,
                 _RESMGR_IO_NFUNCS, &io_funcs);
io_funcs.write = io_write;
```

You may also need to add the following prototype:

```
int io_write (resmgr_context_t *ctp, io_write_t *msg,
              RESMGR_OCB_T *ocb);
```

At this point, if you were to run the resource manager (our simple resource manager used the name **/dev/sample**), you could write to it by doing `echo Hello > /dev/sample` as follows:

```
# echo Hello > /dev/sample
Received 6 bytes = 'Hello'
```

Notice how we passed the last argument to *resmgr_msgread()* (the *offset* argument) as the size of the input message buffer. This effectively skips over the header and gets to the data component.

If the buffer you supplied wasn't big enough to contain the entire message from the client (e.g., you had a 4 KB buffer and the client wanted to write 1 megabyte), you'd have to read the buffer in stages, using a `for` loop, advancing the offset passed to *resmgr_msgread()* by the amount read each time.

Unlike the *io_read* handler sample, this time we didn't do anything with `ocb->offset`. In this case there's no reason to. The `ocb->offset` would make more sense if we were managing things that had advancing positions such as a file position.

The reply is simpler than with the *io_read* handler, since a *write()* call doesn't expect any data back. Instead, it just wants to know if the write succeeded and if so, how many bytes were written. To tell it how many bytes were written we used the *_IO_SET_WRITE_NBYTES()* macro. It takes the *nbytes* that we give it and stores it in the context structure (*ctp*). Then when we return to the library, the library takes this *nbytes* and passes it as the second parameter to the *MsgReplyv()*. The second parameter tells the kernel what the *MsgSend()* should return. And since the *write()* function is calling *MsgSend()*, that's where it finds out how many bytes were written.

Since we're writing to the device, we should also update the modification, and potentially, the creation time. For details on updating the modification and change of file status times, see the section on "*Updating the time for reads and writes*" below.

Methods of returning and replying

You can return to the resource manager library from your handler functions in various ways. This is complicated by the fact that the resource manager library can reply for you if you want it to, but you must tell it to do so and put the information that it'll use in all the right places.

In this section, we'll discuss the following ways of returning to the resource manager library.

Returning with an error

To reply to the client such that the function the client is calling (e.g., *read()*) will return with an error, you simply return with an appropriate *errno* value (from **<errno.h>**).

```
return (ENOMEM);
```

In the case of a *read()*, this causes the read to return -1 with *errno* set to ENOMEM.

You might sometimes see this in the code for a resource manager:

```
_RESMGR_ERRNO (error_code)
```

but this is the same as simply returning the *error_code* directly. The *_RESMGR_ERRNO()* macro is deprecated.

Returning using an IOV array that points to your data

Sometimes you'll want to reply with a header followed by one of *N* buffers, where the buffer used will differ each time you reply. To do this, you can set up an IOV array whose elements point to the header and to a buffer.

The context structure already has an IOV array. If you want the resource manager library to do your reply for you, then you must use this array. But the array must contain enough elements for your needs. To ensure that this is the case, set the *nparts_max* member of the `resmgr_attr_t` structure that you passed to *resmgr_attach()* when you registered your name in the pathname space.

The following example assumes that the variable *i* contains the offset into the array of buffers of the desired buffer to reply with. The `2` in `_RESMGR_NPARTS(2)` tells the library how many elements in `ctp->iov` to reply with.

```
my_header_t     header;
a_buffer_t      buffers[N];

...

SETIOV(&ctp->iov[0], &header, sizeof(header));
SETIOV(&ctp->iov[1], &buffers[i], sizeof(buffers[i]));
return (_RESMGR_NPARTS(2));
```

Returning with a single buffer containing data

An example of this would be replying to a *read()* where all the data existed in a single buffer. You'll typically see this done in two ways:

```
return (_RESMGR_PTR(ctp, buffer, nbytes));
```

And:

```
SETIOV (ctp->iov, buffer, nbytes);
return (_RESMGR_NPARTS(1));
```

The first method, using the *_RESMGR_PTR()* macro, is just a convenience for the second method where a single IOV is returned.

Returning success but with no data

This can be done in a few ways. The most simple would be:

```
return (EOK);
```

But you'll often see:

```
return (_RESMGR_NPARTS(0));
```

Note that in neither case are you causing the *MsgSend()* to return with a 0. The value that the *MsgSend()* returns is the value passed to the *_IO_SET_READ_NBYTES()*, *_IO_SET_WRITE_NBYTES()*, and other similar macros. These two were used in the read and write samples above.

Getting the resource manager library to do the reply

In this case, you give the client the data and get the resource manager library to do the reply for you. However, the reply data won't be valid by that time. For example, if the reply data was in a buffer that you wanted to free before returning, you could use the following:

```
resmgr_msgwrite (ctp, buffer, nbytes, 0);
free (buffer);
return (EOK);
```

The *resmgr_msgwrite()* function copies the contents of buffer into the client's reply buffer immediately. Note that a reply is still required in order to unblock the client so it can examine the data. Next we free the buffer. Finally, we return to the resource manager library such that it does a reply with zero-length data. Since the reply is of zero length, it doesn't overwrite the data already written into the client's reply buffer. When the client returns from its send call, the data is there waiting for it.

Performing the reply in the server

In all of the previous examples, it's the resource manager library that calls *MsgReply*()* or *MsgError()* to unblock the client. In some cases, you may not want the library to reply for you. For instance, you might have already done the reply yourself, or you'll reply later. In either case, you'd return as follows:

```
return (_RESMGR_NOREPLY);
```

Leaving the client blocked, replying later

An example of a resource manager that would reply to clients later is a pipe resource manager. If the client is doing a read of your pipe but you have no data for the client, then you have a choice:

- You can reply with an error (EAGAIN).

  Or:

- You can leave the client blocked and later, when your write handler function is called, you can reply to the client with the new data.

Another example might be if the client wants you to write to some device but doesn't want to get a reply until the data has been fully written out. Here's the sequence of events that might follow:

1. Your resource manager does some I/O to the hardware to tell it that data is available.
2. The hardware generates an interrupt when it's ready for a packet of data.
3. You handle the interrupt by writing data out to the hardware.
4. Many interrupts may occur before all the data is written—only then would you reply to the client.

The first issue, though, is whether the client wants to be left blocked. If the client doesn't want to be left blocked, then it opens with the O_NONBLOCK flag:

```
fd = open("/dev/sample", O_RDWR | O_NONBLOCK);
```

The default is to allow you to block it.

One of the first things done in the read and write samples above was to call some POSIX verification functions: *iofunc_read_verify()* and *iofunc_write_verify()*. If we pass the address of an `int` as the last parameter, then on return the functions will stuff that `int` with nonzero if the client doesn't want to be blocked (O_NONBLOCK flag was set) or with zero if the client wants to be blocked.

```
int    nonblock;

if ((status = iofunc_read_verify (ctp, msg, ocb,
                                  &nonblock)) != EOK)
    return (status);

...

int    nonblock;

if ((status = iofunc_write_verify (ctp, msg, ocb,
                                   &nonblock)) != EOK)
    return (status);
```

When it then comes time to decide if we should reply with an error or reply later, we do:

```
if (nonblock) {
    /* client doesn't want to be blocked */
    return (EAGAIN);
} else {
    /*
    *  The client is willing to be blocked.
    *  Save at least the ctp->rcvid so that you can
    *  reply to it later.
    */
    ...
    return (_RESMGR_NOREPLY);
}
```

The question remains: How do you do the reply yourself? The only detail to be aware of is that the *rcvid* to reply to is `ctp->rcvid`. If you're replying later, then you'd save `ctp->rcvid` and use the saved value in your reply:

```
MsgReply(saved_rcvid, 0, buffer, nbytes);
```

Or:

```
iov_t    iov[2];

SETIOV(&iov[0], &header, sizeof(header));
SETIOV(&iov[1], &buffers[i], sizeof(buffers[i]));
MsgReplyv(saved_rcvid, 0, iov, 2);
```

At this point, it's not safe to use *resmgr_msgreply()* or *resmgr_msgreplyv()*, as explained in the references for those functions—you must use *MsgReply()* or *MsgReplyv()*.

Note that you can fill up the client's reply buffer as data becomes available by using *resmgr_msgwrite()* and *resmgr_msgwritev()*. Just remember to do the *MsgReply*()* at some time to unblock the client.

If you're replying to an _IO_READ or _IO_WRITE message, the *status* argument for *MsgReply*()* must be the number of bytes read or written.

There's another way to resume the blocked operation, but it isn't as efficient as the other methods: you can call *resmgr_msg_again()*. This function restores the `resmgr_context_t` structure to the way it was when your resource manager received the message associated with the *rcvid*, and then processes the message again as if it had just been received.

If your resource manager leaves clients blocked, you'll need to keep track of which clients are blocked, so that you can unblock them if necessary. For more information, see "*Unblocking if someone closes a file descriptor*" in the Unblocking Clients and Handling Interrupts chapter.

Returning and telling the library to do the default action

The default action in most cases is for the library to cause the client's function to fail with ENOSYS:

```
return (_RESMGR_DEFAULT);
```

Handling other read/write details

Handling the *xtype* member

The message structures passed to the *io_read*, *io_write*, and *io_openfd* handlers contain a member called *xtype*. From `struct _io_read`:

```
struct _io_read {
    ...
    uint32_t            xtype;
    ...
}
```

Basically, the *xtype* contains extended information that can be used to adjust the behavior of a standard I/O function. This information includes a type and optionally some flags:

- For _IO_READ and _IO_WRITE messages, the names are in the form _IO_XTYPE_* or _IO_XFLAG_*.
- For _IO_OPENFD messages, the names are in the form _IO_OPENFD_*. The only one that you'll likely have to worry about (assuming you even want to write your own handler) is _IO_OPENFD_NONE.

To isolate the type from the flags in _IO_READ and _IO_WRITE messages, use a bitwise AND of the *xtype* member with _IO_XTYPE_MASK:

```
if ((msg->i.xtype & _IO_XTYPE_MASK) == ...)
```

Most resource managers care about only a few types:

_IO_XTYPE_NONE

No extended type information is being provided.

_IO_XTYPE_OFFSET

If clients are calling *pread()*, *pread64()*, *pwrite()*, or *pwrite64()*, then they don't want you to use the offset in the OCB. Instead, they're providing a one-shot offset. That offset follows the `struct _io_read` or `struct _io_write` headers that reside at the beginning of the message buffers.

For example:

```
struct myread_offset {
    struct _io_read        read;
    struct _xtype_offset   offset;
}
```

Some resource managers can be sure that their clients will never call *pread*()* or *pwrite*()*. (For example, a resource manager that's controlling a robot arm probably wouldn't care.) In this case, you can treat this type of message as an error.

_IO_XTYPE_READCOND

If a client is calling *readcond()*, they want to impose timing and return buffer size constraints on the read. Those constraints follow the `struct _io_read` or `struct _io_write` headers at the beginning of the message buffers. For example:

```
struct myreadcond {
    struct _io_read        read;
    struct _xtype_readcond cond;
}
```

As with _IO_XTYPE_OFFSET, if your resource manager isn't prepared to handle *readcond()*, you can treat this type of message as an error.

_IO_XTYPE_READDIR

The *readdir()* function sets this flag in an _IO_READ message to indicate that the client is reading a directory. In your handler for this message, you could give an error of EISDIR if a client does a *read()* on a directory.

The following types are used for special purposes, so your resource manager probably doesn't need to handle them:

- _IO_XTYPE_MQUEUE
- _IO_XTYPE_REGISTRY
- _IO_XTYPE_TCPIP
- _IO_XTYPE_TCPIP_MMSG (QNX Neutrino 7.0 or later)
- _IO_XTYPE_TCPIP_MSG
- _IO_XTYPE_TCPIP_MSG2

The *xtype* member may also include some flags. Your resource manager might be interested in the following:

_IO_XFLAG_DIR_EXTRA_HINT

This flag is valid only when you're reading from a directory. The filesystem should normally return extra directory information when it's easy to get. If this flag is set, it is a hint to the filesystem to try harder (possibly causing media lookups) to return the extra information. The most common use is to return _DTYPE_LSTAT information.

The *readdir()* function sets this flag in an _IO_READ message if you've used *dircntl()* to set D_FLAG_STAT for the directory.

_IO_XFLAG_DIR_STAT_FORM_*

(QNX Neutrino 7.0 or later) These bits specify which form of the `stat` structure your resource manager should return when a client reads a directory. The client can specify the form via *dircntl()*, and *readdir()* sets the flag accordingly:

dircntl() command	*xtype* flag	Structure
D_FLAG_STAT_FORM_UNSET	_IO_XFLAG_DIR_STAT_FORM_UNSET	The default for the 32- or 64-bit architecture that you're using
D_FLAG_STAT_FORM_T32_2001	_IO_XFLAG_DIR_STAT_FORM_T32_2001	`struct __stat_t32_2001`
D_FLAG_STAT_FORM_T32_2008	_IO_XFLAG_DIR_STAT_FORM_T32_2008	`struct __stat_t32_2008`
D_FLAG_STAT_FORM_T64_2008	_IO_XFLAG_DIR_STAT_FORM_T64_2008	`struct __stat_t64_2008`

You can use _IO_XFLAG_DIR_STAT_FORM_MASK to isolate these bits from the rest of the *xtype*.

The other defined flags are used for special purposes, so your resource manager probably doesn't need to handle them:

- _IO_XFLAG_BLOCK
- _IO_XFLAG_NONBLOCK

If you aren't expecting extended types (*xtype*)

The following code sample demonstrates how to handle the case where you're not expecting any extended types. In this case, if you get a message that contains an *xtype*, you should reply with ENOSYS. The example can be used in either an *io_read* or *io_write* handler.

```
int
io_read (resmgr_context_t *ctp, io_read_t *msg,
         RESMGR_OCB_T *ocb)
{
    int    status;

    if ((status = iofunc_read_verify(ctp, msg, ocb, NULL))
         != EOK) {
        return (status);
    }

    /* No special xtypes */
    if ((msg->i.xtype & _IO_XTYPE_MASK) != _IO_XTYPE_NONE)
        return (ENOSYS);

    ...
}
```

Handling *pread*()* and *pwrite*()*

Here are code examples that demonstrate how to handle an _IO_READ or _IO_WRITE message when a client calls.

Sample code for handling _IO_READ messages in *pread*()*

The following sample code demonstrates how to handle _IO_READ for the case where the client calls one of the *pread*()* functions.

```
/* we are defining io_pread_t here to make the code below
   simple */
typedef struct {
    struct _io_read         read;
    struct _xtype_offset    offset;
} io_pread_t;

int
io_read (resmgr_context_t *ctp, io_read_t *msg,
         RESMGR_OCB_T *ocb)
{
    off64_t offset; /* where to read from */
    int     status;

    if ((status = iofunc_read_verify(ctp, msg, ocb, NULL))
         != EOK) {
        return(status);
    }

    switch(msg->i.xtype & _IO_XTYPE_MASK) {
    case _IO_XTYPE_NONE:
        offset = ocb->offset;
        break;
    case _IO_XTYPE_OFFSET:
        /*
         *  io_pread_t is defined above.
         *  Client is doing a one-shot read to this offset by
         *  calling one of the pread*() functions
         */
        offset = ((io_pread_t *) msg)->offset.offset;
        break;
    default:
        return(ENOSYS);
    }

    ...
}
```

Sample code for handling _IO_WRITE messages in *pwrite*()*

The following sample code demonstrates how to handle _IO_WRITE for the case where the client calls one of the *pwrite*()* functions. Keep in mind that the `struct _xtype_offset` information follows the `struct _io_write` in the sender's message buffer. This means that the data to be written

follows the `struct _xtype_offset` information (instead of the normal case where it follows the `struct _io_write`). So, you must take this into account when doing the *resmgr_msgread()* call in order to get the data from the sender's message buffer.

```
/* we are defining io_pwrite_t here to make the code below
   simple */
typedef struct {
    struct _io_write        write;
    struct _xtype_offset    offset;
} io_pwrite_t;

int
io_write (resmgr_context_t *ctp, io_write_t *msg,
          RESMGR_OCB_T *ocb)
{
    off64_t offset; /* where to write */
    int     status;
    size_t  skip;   /* offset into msg to where the data
                       resides */

    if ((status = iofunc_write_verify(ctp, msg, ocb, NULL))
         != EOK) {
        return(status);
    }

    switch(msg->i.xtype & _IO_XTYPE_MASK) {
    case _IO_XTYPE_NONE:
        offset = ocb->offset;
        skip = sizeof(io_write_t);
        break;
    case _IO_XTYPE_OFFSET:
        /*
         *  io_pwrite_t is defined above
         *  client is doing a one-shot write to this offset by
         *  calling one of the pwrite*() functions
         */
        offset = ((io_pwrite_t *) msg)->offset.offset;
        skip = sizeof(io_pwrite_t);
        break;
    default:
        return(ENOSYS);
    }

    ...

    /*
     *  get the data from the sender's message buffer,
     *  skipping all possible header information
     */
    resmgr_msgreadv(ctp, iovs, niovs, skip);

    ...
}
```

Handling *readcond()*

The same type of operation that was done to handle the *pread()*/_IO_XTYPE_OFFSET case can be used for handling the client's *readcond()* call:

```
typedef struct {
    struct _io_read        read;
    struct _xtype_readcond cond;
} io_readcond_t
```

Then:

```
struct _xtype_readcond *cond
...
    CASE _IO_XTYPE_READCOND:
        cond = &((io_readcond_t *)msg)->cond
        break;
}
```

Then your manager has to properly interpret and deal with the arguments to *readcond()*. For more information, see the QNX Neutrino *C Library Reference*.

Updating the time for reads and writes

In the read sample above we did:

```
if (msg->i.nbytes > 0)
    ocb->attr->flags |= IOFUNC_ATTR_ATIME;
```

According to POSIX, if the read succeeds and the reader had asked for more than zero bytes, then the access time must be marked for update. But POSIX doesn't say that it must be updated right away. If you're doing many reads, you may not want to read the time from the kernel for every read. In the code above, we mark the time only as needing to be updated. When the next _IO_STAT or _IO_CLOSE_OCB message is processed, the resource manager library will see that the time needs to be updated and will get it from the kernel then. This of course has the disadvantage that the time is not the time of the read.

Similarly for the write sample above, we did:

```
if (msg->i.nbytes > 0)
    ocb->attr->flags |= IOFUNC_ATTR_MTIME | IOFUNC_ATTR_CTIME;
```

so the same thing will happen.

If you *do* want to have the times represent the read or write times, then after setting the flags you need only call the *iofunc_time_update()* helper function. So the read lines become:

```
if (msg->i.nbytes > 0) {
    ocb->attr->flags |= IOFUNC_ATTR_ATIME;
    iofunc_time_update(ocb->attr);
}
```

and the write lines become:

```
if (msg->i.nbytes > 0) {
    ocb->attr->flags |= IOFUNC_ATTR_MTIME | IOFUNC_ATTR_CTIME;
    iofunc_time_update(ocb->attr);
}
```

You should call *iofunc_time_update()* before you flush out any cached attributes. As a result of changing the time fields, the attribute structure will have the IOFUNC_ATTR_DIRTY_TIME bit set in the flags field, indicating that this field of the attribute must be updated when the attribute is flushed from the cache.

Chapter 6
Combine Messages

In order to conserve network bandwidth and to provide support for atomic operations, *combine messages* are supported. A combine message is constructed by the client's C library and consists of a number of I/O and/or connect messages packaged together into one.

Where combine messages are used

Let's see how combine messages are used.

Atomic operations

Consider a case where two threads are executing the following code, trying to read from the same file descriptor:

```
Atomic
a_thread ()
{
    char buf [BUFSIZ];

    lseek (fd, position, SEEK_SET);
    read (fd, buf, BUFSIZ);
    …
}
```

The first thread performs the *lseek()* and then gets preempted by the second thread. When the first thread resumes executing, its offset into the file will be at the end of where the second thread read from, *not* the position that it had *lseek()*'d to.

This can be solved in one of three ways:

- The two threads can use a *mutex* to ensure that only one thread at a time is using the file descriptor.
- Each thread can open the file itself, thus generating a unique file descriptor that won't be affected by any other threads.
- The threads can use the *readblock()* function, which performs an atomic *lseek()* and *read()*.

Let's look at these three methods.

Using a mutex

In the first approach, if the two threads use a mutex between themselves, the following issue arises: every *read()*, *lseek()*, and *write()* operation *must* use the mutex.

If this practice isn't enforced, then you still have the exact same problem. For example, suppose one thread that's obeying the convention locks the mutex and does the *lseek()*, thinking that it's protected. However, another thread (that's not obeying the convention) can preempt it and move the offset to somewhere else. When the first thread resumes, we again encounter the problem where the offset is at a different (unexpected) location. Generally, using a mutex will be successful only in very tightly managed projects, where a code review will *ensure* that each and every thread's file functions obey the convention.

Per-thread files

The second approach—of using different file descriptors—is a good general-purpose solution, *unless* you explicitly wanted the file descriptor to be shared.

The *readblock()* function

In order for the *readblock()* function to be able to effect an atomic seek/read operation, it must ensure that the requests it sends to the resource manager will all be processed at the

same time. This is done by combining the _IO_LSEEK and _IO_READ messages into one message. Thus, when the base layer performs the *MsgReceive()*, it will receive the entire *readblock()* request in one atomic message.

Bandwidth considerations

Another place where combine messages are useful is in the *stat()* function, which can be implemented by calling *open()*, *fstat()*, and *close()* in sequence.

Rather than generate three separate messages (one for each of the functions), the C library combines them into one contiguous message. This boosts performance, especially over a networked connection, and also simplifies the resource manager, because it's not forced to have a connect function to handle *stat()*.

The library's combine-message handling

The resource manager library handles combine messages by presenting each component of the message to the appropriate handler routines. For example, if we get a combine message that has an _IO_LSEEK and _IO_READ in it (e.g., *readblock()*), the library will call our *io_lseek* and *io_read* handlers for us in turn.

But let's see what happens in the resource manager when it's handling these messages. With multiple threads, both of the client's threads may very well have sent in their "atomic" combine messages. Two threads in the resource manager will now attempt to service those two messages. We again run into the same synchronization problem as we originally had on the client end—one thread can be partway through processing the message and can then be preempted by the other thread.

The solution? The resource manager library provides callouts to lock the OCB while processing any message (except _IO_CLOSE and _IO_UNBLOCK—we'll return to these). As an example, when processing the *readblock()* combine message, the resource manager library performs callouts in this order:

1. *lock_ocb* handler
2. _IO_LSEEK message handler
3. _IO_READ message handler
4. *unlock_ocb* handler

Therefore, in our scenario, the two threads within the resource manager would be mutually exclusive to each other by virtue of the lock—the first thread to acquire the lock would completely process the combine message, unlock the lock, and then the second thread would perform its processing.

Let's examine several of the issues that are associated with handling combine messages.

Component responses

As we've seen, a combine message really consists of a number of "regular" resource manager messages combined into one large contiguous message. The resource manager library handles each component in the combine message separately by extracting the individual components and then out calling to the handlers you've specified in the connect and I/O function tables, as appropriate, for each component.

This generally doesn't present any new wrinkles for the message handlers themselves, except in one case. Consider the *readblock()* combine message:

Client call:

readblock()

Message(s):

_IO_LSEEK , _IO_READ

Callouts:

io_lock_ocb, *io_lseek*, *io_read*, *io_unlock_ocb*

Ordinarily, after processing the _IO_LSEEK message, your handler would return the current position within the file. However, the next message (the _IO_READ) also returns data. By convention, only the

last data-returning message within a combine message will actually return data. The intermediate messages are allowed to return only a pass/fail indication.

The impact of this is that the _IO_LSEEK message handler has to be aware of whether or not it's being invoked as part of combine message handling. If it is, it should only return either an EOK (indicating that the *lseek()* operation succeeded) or an error indication to indicate some form of failure.

But if the _IO_LSEEK handler isn't being invoked as part of combine message handling, it should return the EOK *and* the new offset (or, in case of error, an error indication only).

Here's a sample of the code for the default iofunc-layer *lseek()* handler:

```
int
iofunc_lseek_default (resmgr_context_t *ctp,
                      io_lseek_t *msg,
                      iofunc_ocb_t *ocb)
{
    /*
     *  performs the lseek processing here
     *  may "early-out" on error conditions
     */
     . . .

    /* decision re: combine messages done here */
    if (msg -> i.combine_len & _IO_COMBINE_FLAG) {
        return (EOK);
    }

    msg -> o = offset;
    return (_RESMGR_PTR (ctp, &msg -> o, sizeof (msg -> o)));
}
```

The relevant decision is made in this statement:

```
if (msg -> i.combine_len & _IO_COMBINE_FLAG)
```

If the _IO_COMBINE_FLAG bit is set in the *combine_len* member, this indicates that the message is being processed as part of a combine message.

When the resource manager library is processing the individual components of the combine message, it looks at the error return from the individual message handlers. If a handler returns anything other than EOK, then processing of further combine message components is aborted. The error that was returned from the failing component's handler is returned to the client.

Component data access

The second issue associated with handling combine messages is how to access the data area for subsequent message components.

For example, the *writeblock()* combine message format has an *lseek()* message first, followed by the *write()* message. This means that the data associated with the *write()* request is further in the received message buffer than would be the case for just a simple _IO_WRITE message:

Client call:

writeblock()

Message(s):

_IO_LSEEK , _IO_WRITE , data

Callouts:

io_lock_ocb, *io_lseek*, *io_write*, *io_unlock_ocb*

This issue is easy to work around. There's a resource manager library function called *resmgr_msgread()* that knows how to get the data corresponding to the correct message component. Therefore, in the *io_write* handler, if you used *resmgr_msgread()* instead of *MsgRead()*, this would be transparent to you.

Resource managers should always use *resmgr_msg*()* cover functions.

For reference, here's the source for *resmgr_msgread()*:

```
ssize_t resmgr_msgread( resmgr_context_t * const ctp,
                        void * const msg,
                        const size_t size,
                        const size_t offset)
{
    return MsgRead(ctp->rcvid, msg, size, ctp->offset + offset);
}
```

As you can see, *resmgr_msgread()* simply calls *MsgRead()* with the offset of the component message from the beginning of the combine message buffer. For completeness, there's also a *resmgr_msgwrite()* that works in an identical manner to *MsgWrite()*, except that it dereferences the passed *ctp* to obtain the *rcvid*.

Locking and unlocking the attribute structure

As mentioned above, another facet of the operation of the *readblock()* function from the client's perspective is that it's atomic. In order to process the requests for a particular OCB in an atomic manner, we must lock and unlock the attribute structure pointed to by the OCB, thus ensuring that only one resource manager thread has access to the OCB at a time.

The resource manager library provides two callouts for doing this:

- *lock_ocb*
- *unlock_ocb*

These are members of the I/O functions structure. The handlers that you provide for those callouts should lock and unlock the attribute structure pointed to by the OCB by calling *iofunc_attr_lock()* and *iofunc_attr_unlock()*. Therefore, if you're locking the attribute structure, there's a possibility that the *lock_ocb* callout will block for a period of time. This is normal and expected behavior. Note also that the attributes structure is automatically locked for you when your I/O function is called.

Connect message types

Let's take a look at the general case for the *io_open* handler—it doesn't always correspond to the client's *open()* call!

For example, consider the *stat()* and *access()* client function calls:

_IO_CONNECT_COMBINE_CLOSE

For a *stat()* client call, we essentially perform the sequence *open()*, *fstat()*, and *close()*. Note that if we actually did that, three messages would be required. For performance reasons, we implement the *stat()* function as one single combine message:

Client call:

stat()

Message(s):

_IO_CONNECT_COMBINE_CLOSE , _IO_STAT

Callouts:

io_open, *io_lock_ocb*, *io_stat*, *io_unlock_ocb*, *io_close*

The _IO_CONNECT_COMBINE_CLOSE message causes the *io_open* handler to be called. It then implicitly (at the end of processing for the combine message) causes the *io_close_ocb* handler to be called.

_IO_CONNECT_COMBINE

For the *access()* function, the client's C library will open a connection to the resource manager and perform a *stat()* call. Then, based on the results of the *stat()* call, the client's C library *access()* may perform an optional *devctl()* to get more information. In any event, because *access()* opened the device, it must also call *close()* to close it:

Client call:

access()

Message(s):

_IO_CONNECT_COMBINE , _IO_STAT, _IO_DEVCTL (optional), _IO_CLOSE

Callouts:

io_open, *io_lock_ocb*, *io_stat*, *io_unlock_ocb*, *io_lock_ocb* (optional), *io_devctl* (optional), *io_unlock_ocb* (optional), *io_close*

Notice how the *access()* function opened the pathname/device—it sent it an _IO_CONNECT_COMBINE message along with the _IO_STAT message. This creates an OCB (when the *io_open* handler is called), locks the associated attribute structure (via *io_lock_ocb*), performs the stat (*io_stat*), and then unlocks the attributes structure (*io_unlock_ocb*). Note that we don't implicitly close the OCB—this is left for a later, explicit, message. Contrast this handling with that of the plain *stat()* above.

Chapter 7
Extending the POSIX-Layer Data Structures

The *iofunc_*()* default functions operate on the assumption that you've used the default definitions for the context block and the attributes structures. This is a safe assumption for two reasons:

1. The default context and attribute structures contain sufficient information for most applications.
2. If the default structures don't hold enough information, you can encapsulate them within the structures that you've defined.

The default structures must be the first members of their respective superstructures, so that the *iofunc_*()* default functions can access them:

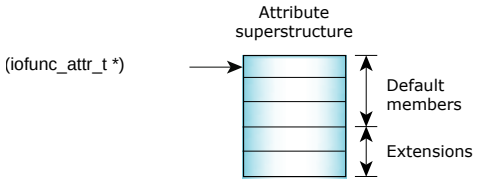

Figure 6: Encapsulating the POSIX-layer data structures.

Extending the OCB and attribute structures

In our **/dev/sample** example, we had a static buffer associated with the entire resource. Sometimes you may want to keep a pointer to a buffer associated with the resource, rather than in a global area. To maintain the pointer with the resource, we would have to store it in the `iofunc_attr_t` attribute structure. Since the attribute structure doesn't have any spare fields, we would have to extend it to contain that pointer.

Sometimes you may want to add extra entries to the standard *iofunc_*()* OCB (`iofunc_ocb_t`).

Let's see how we can extend both of these structures. The basic strategy used is to encapsulate the existing attributes and OCB structures within a newly defined superstructure that also contains our extensions. Here's the code (see the text following the listing for comments):

```
/* Define our overrides before including <sys/iofunc.h>  */
struct device;
#define IOFUNC_ATTR_T       struct device  /* see note 1 */
struct ocb;
#define IOFUNC_OCB_T        struct ocb     /* see note 1 */

#include <sys/iofunc.h>
#include <sys/dispatch.h>

struct ocb {                               /* see note 2 */
    iofunc_ocb_t            hdr;           /* see note 4; must always be first */
    struct ocb              *next;
    struct ocb              **prev;        /* see note 3 */
};

struct device {                            /* see note 2 */
    iofunc_attr_t           attr;          /* must always be first */
    struct ocb              *list;         /* waiting for write */
};

/* Prototypes, needed since we refer to them a few lines down */

struct ocb *ocb_calloc (resmgr_context_t *ctp, struct device *device);
void ocb_free (struct ocb *ocb);

iofunc_funcs_t ocb_funcs = { /* our ocb allocating & freeing functions */
    _IOFUNC_NFUNCS,
    ocb_calloc,
    ocb_free
};

/* The mount structure.  We have only one, so we statically declare it */

iofunc_mount_t          mountpoint = { 0, 0, 0, 0, &ocb_funcs };

/* One struct device per attached name (there's only one name in this
   example) */

struct device           deviceattr;
```

```
main()
{
    ...

    /*
     *  deviceattr will indirectly contain the addresses
     *  of the OCB allocating and freeing functions
     */

    deviceattr.attr.mount = &mountpoint;
    resmgr_attach (..., &deviceattr);

    ...
}

/*
 * ocb_calloc
 *
 *  The purpose of this is to give us a place to allocate our own OCB.
 *  It is called as a result of the open being done
 *  (e.g., iofunc_open_default causes it to be called). We
 *  registered it through the mount structure.
 */
IOFUNC_OCB_T
*ocb_calloc (resmgr_context_t *ctp, IOFUNC_ATTR_T *device)
{
    struct ocb *ocb;

    if (!(ocb = calloc (1, sizeof (*ocb)))) {
        return 0;
    }

    /* see note 3 */
    ocb -> prev = &device -> list;
    if (ocb -> next = device -> list) {
        device -> list -> prev = &ocb -> next;
    }
    device -> list = ocb;

    return (ocb);
}

/*
 * ocb_free
 *
 * The purpose of this is to give us a place to free our OCB.
 * It is called as a result of the close being done
 * (e.g., iofunc_close_ocb_default causes it to be called). We
 * registered it through the mount structure.
 */
void
ocb_free (IOFUNC_OCB_T *ocb)
{
```

```
    /* see note 3 */
    if (*ocb -> prev = ocb -> next) {
        ocb -> next -> prev = ocb -> prev;
        }
        free (ocb);
}
```

Here are the notes for the above code:

1. We place the definitions for our enhanced structures *before* including the standard I/O functions header file. Because the standard I/O functions header file checks to see if the two manifest constants are already defined, this allows a convenient way for us to semantically override the structures.
2. Define our new enhanced data structures, being sure to place the encapsulated members first.
3. The *ocb_calloc()* and *ocb_free()* sample functions shown here cause the newly allocated OCBs to be maintained in a linked list. Note the use of dual indirection on the `struct ocb **prev;` member.
4. You **must always** place the `iofunc` structure that you're overriding as the first member of the new extended structure. This lets the common library work properly in the default cases.

Extending the mount structure

You can also extend the `iofunc_mount_t` structure in the same manner as the attribute and OCB structures. In this case, you'd define:

```
#define IOFUNC_MOUNT_T       struct newmount
```

and then declare the new structure:

```
struct newmount {
    iofunc_mount_t          mount;
    int                    ourflag;
};
```

Chapter 8
Handling Other Messages

Custom messages

Although most of the time your resource manager will handle messages from clients, there still could be times when you need to control the behavior of the resource manager itself. For example, if the resource manager is for a serial port, you'll likely need a way to change the baud rate, and so on.

There are various ways you could send control information to a resource manager:

- Stop the resource manager, and then restart it with new command-line options. This is an awkward method that leaves the device unavailable while the resource manager is restarting. It isn't a viable solution if the resource manager controls various devices and you need to set different options (at different times) for different devices.
- Accept *write()* messages that include control instructions. This lets you control the resource manager from the command line or a script; you can simply `echo` a command to a path that the resource manager has registered.

  However, the *write()* messages might be broken into smaller messages, so your resource manager would have to be prepared to save the pieces, reassemble them, and then act on them. Your *io_write* handler also has to parse the commands, in addition to handling *write()* messages from the clients.
- Send an IPC message, via *MsgSend()*, directly to the resource manager. This message would contain a data structure that includes the control instructions. This method is fast, simple, and flexible, but it isn't portable because *MsgSend()* isn't a POSIX function.
- Send control instructions via a *devctl()* command. Such messages won't conflict with any other messages sent to the resource manager, but this approach might not be portable because *devctl()* isn't a POSIX function (it was in a draft standard, but wasn't adopted). See "*Handling devctl() messages*," below.
- Set up a handler for "other" I/O messages when you call *resmgr_attach()*, as described in "*Setting resource manager attributes*" in the "Fleshing Out the Skeleton" chapter. We don't recommend this approach.
- Use *message_attach()* or *pulse_attach()* to attach a specified message range or pulse code for the dispatch handle. When a message with a type in that range is received, the *dispatch_block()* function calls the handler function that you specify. The message range must be outside the I/O range, and the messages or pulses aren't associated with an OCB. For more information, see "*Handling private messages and pulses*," later in this chapter.
- Use an out-of-band I/O message, _IO_MSG. This type of message *is* associated with an OCB, and is handled through the resource manager's framework. For more information, see "*Handling out-of-band (_IO_MSG) messages*," later in this chapter.

Handling *devctl()* messages

The *devctl()* function is a general-purpose mechanism for communicating with a resource manager. Clients can send data to, receive data from, or both send and receive data from a resource manager. The prototype of the client *devctl()* call is:

```
int devctl( int fd,
            int dcmd,
            void * data,
            size_t nbytes,
            int * return_info);
```

The following values (described in detail in the *devctl()* documentation in the QNX Neutrino *C Library Reference*) map directly to the _IO_DEVCTL message itself:

```
struct _io_devctl {
        uint16_t                type;
        uint16_t                combine_len;
        int32_t                 dcmd;
        uint32_t                nbytes;
        int32_t                 zero;
/*      char                    data[nbytes]; */
};

struct _io_devctl_reply {
        uint32_t                zero;
        int32_t                 ret_val;
        uint32_t                nbytes;
        int32_t                 zero2;
/*      char                    data[nbytes]; */
    } ;

typedef union {
        struct _io_devctl       i;
        struct _io_devctl_reply o;
} io_devctl_t;
```

As with most resource manager messages, we've defined a `union` that contains the input structure (coming into the resource manager), and a reply or output structure (going back to the client). The *io_devctl* resource manager handler is prototyped with the argument:

```
io_devctl_t *msg
```

which is the pointer to the union containing the message.

The *type* member has the value _IO_DEVCTL.

The *combine_len* field has meaning for a combine message; see the *Combine Messages* chapter.

The *nbytes* value is the *nbytes* that's passed to the *devctl()* function. The value contains the size of the data to be sent to the device driver, or the maximum size of the data to be received from the device driver.

The most interesting item of the input structure is the *dcmd* argument that's passed to the *devctl()* function. This command is formed using the macros defined in **<devctl.h>**:

```
#define _POSIX_DEVDIR_NONE        0
#define _POSIX_DEVDIR_TO          0x80000000
#define _POSIX_DEVDIR_FROM        0x40000000
#define __DIOF(class, cmd, data)  ((sizeof(data)<<16) + ((class)<<8) + (cmd) + _POSIX_DEVDIR_FROM)
#define __DIOT(class, cmd, data)  ((sizeof(data)<<16) + ((class)<<8) + (cmd) + _POSIX_DEVDIR_TO)
#define __DIOTF(class, cmd, data) ((sizeof(data)<<16) + ((class)<<8) + (cmd) + _POSIX_DEVDIR_TOFROM)
#define __DION(class, cmd)        (((class)<<8) + (cmd) + _POSIX_DEVDIR_NONE)
```

It's important to understand how these macros pack data to create a command. An 8-bit class (defined in **<devctl.h>**) is combined with an 8-bit subtype that's manager-specific, and put together in the lower 16 bits of the integer.

The upper 16 bits contain the direction (TO, FROM) as well as a hint about the size of the data structure being passed. This size is only a hint put in to uniquely identify messages that may use the same class and code but pass different data structures.

In the following example, a command is generated to indicate that the client is sending data to the server (TO), but not receiving anything in return. The only bits that the library or the resource manager layer look at are the TO and FROM bits to determine which arguments are to be passed to *MsgSend()*.

```
struct _my_devctl_msg {
    ...
}

#define MYDCMD  __DIOT(_DCMD_MISC, 0x54, struct _my_devctl_msg)
```

The size of the structure that's passed as the last field to the *__DIO** macros **must** be less than 2^{14} == 16 KB. Anything larger than this interferes with the upper two directional bits.

The data directly follows this message structure, as indicated by the `/* char data[nbytes] */` comment in the `_io_devctl` structure.

Sample code for handling _IO_DEVCTL messages

You can add the following code samples to either of the "**/dev/null**" examples provided in the "*Simple device resource manager examples*" section of the Bones of a Resource Manager chapter. Both of those code samples provided the name **/dev/sample**. With the changes indicated below, the client can use *devctl()* to set and retrieve a global value (an integer in this case) that's maintained in the resource manager.

The first addition defines what the *devctl()* commands are going to be. This is generally put in a common or shared header file:

```
typedef union _my_devctl_msg {
        int tx;             /* Filled by client on send */
        int rx;             /* Filled by server on reply */
} data_t;

#define MY_CMD_CODE      1
#define MY_DEVCTL_GETVAL __DIOF(_DCMD_MISC,  MY_CMD_CODE + 0, int)
#define MY_DEVCTL_SETVAL __DIOT(_DCMD_MISC,  MY_CMD_CODE + 1, int)
#define MY_DEVCTL_SETGET __DIOTF(_DCMD_MISC, MY_CMD_CODE + 2, union _my_devctl_msg)
```

In the above code, we defined three commands that the client can use:

MY_DEVCTL_SETVAL

Sets the server's global variable to the integer the client provides.

MY_DEVCTL_GETVAL

Gets the value of the server's global variable and puts it into the client's buffer.

MY_DEVCTL_SETGET

Sets the server's global variable to the integer that the client provides, and then returns the previous value of the server's global variable in the client's buffer.

Add this code to the *main()* function:

```
/* For handling _IO_DEVCTL, sent by devctl() */
io_funcs.devctl = io_devctl;
```

And the following code gets added before the *main()* function:

```
int io_devctl(resmgr_context_t *ctp, io_devctl_t *msg,
              RESMGR_OCB_T *ocb);

int global_integer = 0;
```

Now, you need to include the new handler function to handle the _IO_DEVCTL message (see the text following the listing for additional notes):

```
int io_devctl(resmgr_context_t *ctp, io_devctl_t *msg,
              RESMGR_OCB_T *ocb) {
    int     nbytes, status, previous;

    union {  /* See note 1 */
        data_t  data;
        int     data32;
        /* ... other devctl types you can receive */
    } *rx_data;


    /*
     Let common code handle DCMD_ALL_* cases.
     You can do this before or after you intercept devctls, depending
     on your intentions.  Here we aren't using any predefined values,
     so let the system ones be handled first. See note 2.
    */
    if ((status = iofunc_devctl_default(ctp, msg, ocb)) !=
         _RESMGR_DEFAULT) {
        return(status);
    }
    status = nbytes = 0;

    /*
     Note this assumes that you can fit the entire data portion of
     the devctl into one message.  In reality you should probably
     perform a MsgReadv() once you know the type of message you
     have received to get all of the data, rather than assume
```

```
 it all fits in the message.  We have set in our main routine
 that we'll accept a total message size of up to 2 KB, so we
 don't worry about it in this example where we deal with ints.
*/

/* Get the data from the message. See Note 3. */
rx_data = _DEVCTL_DATA(msg->i);

/*
 Three examples of devctl operations:
 SET: Set a value (int) in the server
 GET: Get a value (int) from the server
 SETGET: Set a new value and return the previous value
*/
switch (msg->i.dcmd) {
case MY_DEVCTL_SETVAL:
    global_integer = rx_data->data32;
    nbytes = 0;
    break;

case MY_DEVCTL_GETVAL:
    rx_data->data32 = global_integer; /* See note 4 */
    nbytes = sizeof(rx_data->data32);
    break;

case MY_DEVCTL_SETGET:
    previous = global_integer;
    global_integer = rx_data->data.tx;

    /* See note 4. The rx data overwrites the tx data
       for this command. */

    rx_data->data.rx = previous;
    nbytes = sizeof(rx_data->data.rx);
    break;

default:
    return(ENOSYS);
}

/* Clear the return message. Note that we saved our data past
   this location in the message. */
memset(&msg->o, 0, sizeof(msg->o));

/*
 If you wanted to pass something different to the return
 field of the devctl() you could do it through this member.
 See note 5.
*/
msg->o.ret_val = status;

/* Indicate the number of bytes and return the message */
msg->o.nbytes = nbytes;
```

```
    return(_RESMGR_PTR(ctp, &msg->o, sizeof(msg->o) + nbytes));
}
```

Here are the notes for the above code:

1. We define a `union` for all the possible types of received data. The MY_DEVCTL_SETVAL and MY_DEVCTL_GETVAL commands use the *data32* member, and the MY_DEVCTL_SETGET uses the *data* member, of type `data_t`, which is a union of the received and transmitted data.
2. The default *devctl()* handler is called before we begin to service our messages. This allows normal system messages to be processed. If the message isn't handled by the default handler, then it returns _RESMGR_DEFAULT to indicate that the message might be a custom message. This means that we should check the incoming command against commands that our resource manager understands.
3. The data to be passed follows directly after the `io_devctl_t` structure. You can get a pointer to this location by using the `_DEVCTL_DATA(`*msg*`->i)` macro defined in **<devctl.h>**. The argument to this macro **must** be the input message structure: if it's the union message structure or a pointer to the input message structure, the pointer won't point to the right location.

   For your convenience, we've defined a union of all of the messages that this server can receive. However, this won't work with large data messages. In this case, you'd use *resmgr_msgread()* to read the message from the client. Our messages are never larger than `sizeof( int)` and this comfortably fits into the minimum receive buffer size.
4. The data being returned to the client is placed at the end of the reply message. This is the same mechanism used for the input data, so we can use the *_DEVCTL_DATA()* function to get a pointer to this location. With large replies that wouldn't necessarily fit into the server's receive buffer, you should use one of the reply mechanisms described in the "*Methods of returning and replying*" section in the Handling Read and Write Messages chapter. Again, in this example, we're only returning an integer that fits into the receive buffer without any problem.
5. The last argument to the *devctl()* function is a pointer to an integer. If this pointer is provided, then the integer is filled with the value stored in the `msg->o.ret_val` reply message. This is a convenient way for a resource manager to return simple status information without affecting the core *devctl()* operation. It's not used in this example.

If you add the following handler code, a client should be able to open **/dev/sample** and subsequently set and retrieve the global integer value:

```
int main(int argc, char **argv) {
    int     fd, ret, val;
    data_t  data;

    if ((fd = open("/dev/sample", O_RDONLY)) == -1) {
            return(1);
    }

    /* Find out what the value is set to initially */
    val = -1;
    ret = devctl(fd, MY_DEVCTL_GETVAL, &val, sizeof(val), NULL);
    printf("GET returned %d w/ server value %d \n", ret, val);

    /* Set the value to something else */
    val = 25;
```

```
    ret = devctl(fd, MY_DEVCTL_SETVAL, &val, sizeof(val), NULL);
    printf("SET returned %d \n", ret);

    /* Verify we actually did set the value */
    val = -1;
    ret = devctl(fd, MY_DEVCTL_GETVAL, &val, sizeof(val), NULL);
    printf("GET returned %d w/ server value %d == 25? \n", ret, val);

    /* Now do a set/get combination */
    memset(&data, 0, sizeof(data));
    data.tx = 50;
    ret = devctl(fd, MY_DEVCTL_SETGET, &data, sizeof(data), NULL);
    printf("SETGET returned with %d w/ server value %d == 25?\n",
           ret, data.rx);

    /* Check set/get worked */
    val = -1;
    ret = devctl(fd, MY_DEVCTL_GETVAL, &val, sizeof(val), NULL);
    printf("GET returned %d w/ server value %d == 50? \n", ret, val);

    return(0);
}
```

Handling *ionotify()*, *poll()*, and *select()*

A client uses *ionotify()*, *poll()*, and *select()* to ask a resource manager about the status of certain conditions (e.g., whether input data is available). The conditions may or may not have been met. The resource manager can be asked to:

- check the status of the conditions immediately, and return if any have been met
- deliver an event later on when a condition is met (this is referred to as arming the resource manager)

The *poll()* and *select()* functions differ from *ionotify()* in that most of the work is done in the library. For example, the client code would be unaware that any event is involved, nor would it be aware of the blocking function that waits for the event. This is all hidden in the library code for *poll()* and *select()*.

However, from a resource manager's point of view, there's no difference between *ionotify()*, *poll()*, and *select()*; they're handled with the same code.

For more information on the *ionotify()*, *poll()*, and *select()* functions, see the QNX Neutrino *C Library Reference*.

If multiple threads in the same client perform simultaneous operations with *poll()*, *select()*, and *ionotify()*, notification races may occur.

Since *ionotify()*, *poll()*, and *select()* require the resource manager to do the same work, they all send the _IO_NOTIFY or _IO_NOTIFY64 message to the resource manager. The *io_notify* handler is responsible for handling this message. Let's start by looking at the format of the message itself:

```
struct _io_notify {
    uint16_t                    type;
    uint16_t                    combine_len;
    int32_t                     action;
    int32_t                     flags;
    struct __sigevent32         event;

    /* Following fields only valid if (flags & _NOTIFY_COND_EXTEN) */
    int32_t                     mgr[2];    /* For use by manager */
    int32_t                     flags_extra_mask;
    int32_t                     flags_exten;
    int32_t                     nfds;
    int32_t                     fd_first;
    int32_t                     nfds_ready;
    int64_t                     timo;
    /* struct pollfd            fds[nfds]; */
};

struct _io_notify64 {
    uint16_t                    type;
    uint16_t                    combine_len;
    int32_t                     action;
    int32_t                     flags;
    struct __sigevent32         old_event;
```

```
    /* Following fields only valid if (flags & _NOTIFY_COND_EXTEN) */
    int32_t                     mgr[2];    /* For use by manager */
    int32_t                     flags_extra_mask;
    int32_t                     flags_exten;
    int32_t                     nfds;
    int32_t                     fd_first;
    int32_t                     nfds_ready;
    int64_t                     timo;
    union {
        struct __sigevent32     event32;
        struct __sigevent64     event64;
    };
    /* struct pollfd            fds[nfds]; */
};

struct _io_notify_reply {
    uint32_t                    zero;
    uint32_t                    flags;   /* actions above */
    int32_t                     flags2;  /* flags above */
    struct __sigevent32         event;

    /* Following fields only updated by new managers (if valid) */
    int32_t                     mgr[2];    /* For use by manager */
    int32_t                     flags_extra_mask;
    int32_t                     flags_exten;
    int32_t                     nfds;
    int32_t                     fd_first;
    int32_t                     nfds_ready;
    int64_t                     timo;
    /* struct pollfd            fds[nfds]; */
};

struct _io_notify_reply64 {
    uint32_t                    zero;
    uint32_t                    flags;  /* actions */
    int32_t                     flags2; /* flags above */
    struct __sigevent32         old_event;

    /* Following fields only updated by new managers (if valid) */
    int32_t                     mgr[2];    /* For use by manager */
    int32_t                     flags_extra_mask;
    int32_t                     flags_exten;
    int32_t                     nfds;
    int32_t                     fd_first;
    int32_t                     nfds_ready;
    int64_t                     timo;
    union {
        struct __sigevent32     event32;
        struct __sigevent64     event64;
    };
    /* struct pollfd            fds[nfds]; */
};

typedef union {
```

```
    struct _io_notify           i;
    struct _io_notify64         i64;
    struct _io_notify_reply     o;
    struct _io_notify_reply64   o64;
} io_notify_t;
```

The code samples used in this chapter are not always POSIX-compliant.

As with all resource manager messages, we've defined a `union` that contains the input structure (coming into the resource manager), and a reply or output structure (going back to the client). The *io_notify* handler is prototyped with the argument:

```
io_notify_t *msg
```

which is the pointer to the union containing the message. The main items in the input structure are:

- *type*
- *combine_len*
- *action*
- *flags*
- *event32* or *event64*

The *type* member has the value _IO_NOTIFY or _IO_NOTIFY64.

The *combine_len* field has meaning for a combine message; see the *Combine Messages* chapter.

The *action* member is used by the *iofunc_notify()* helper function to tell it whether it should:

- just check for conditions now
- check for conditions now, and if none are met, arm them
- just arm for transitions

Since *iofunc_notify()* looks at this, you don't have to worry about it.

The *flags* member contains the conditions that the client is interested in and can be any mixture of the following:

_NOTIFY_COND_INPUT

This condition is met when there are one or more units of input data available (i.e., clients can now issue reads). The number of units defaults to 1, but you can change it. The definition of a unit is up to you: for a character device such as a serial port, it would be a character; for a POSIX message queue, it would be a message. Each resource manager selects an appropriate object.

_NOTIFY_COND_OUTPUT

This condition is met when there's room in the output buffer for one or more units of data (i.e., clients can now issue writes). The number of units defaults to 1, but you can change it. The definition of a unit is up to you—some resource managers may default to an empty output buffer, while others may choose some percentage of the buffer empty.

_NOTIFY_COND_OBAND

The condition is met when one or more units of out-of-band data are available. The number of units defaults to 1, but you can change it. The definition of out-of-band data is specific to the resource manager.

_NOTIFY_COND_EXTEN

The conditions are defined with some extended flags; used internally.

The *event** member is what the resource manager delivers once a condition is met.

A resource manager needs to keep a list of clients that want to be notified as conditions are met, along with the events to use to do the notifying. When a condition is met, the resource manager must traverse the list to look for clients that are interested in that condition, and then deliver the appropriate event. As well, if a client closes its file descriptor, then any notification entries for that client must be removed from the list.

To make all this easier, the following structure and helper functions are provided for you to use in a resource manager:

`iofunc_notify_t` structure

Contains the three notification lists, one for each possible condition. Each is a list of the clients to be notified for that condition.

iofunc_notify()

Adds or removes notification entries; also polls for conditions. Call this function inside your *io_notify* handler function.

iofunc_notify_trigger()*, *iofunc_notify_trigger_strict()

Sends notifications to queued clients. Call either of these functions when one or more conditions have been met. When the client closes its file descriptor, call *iofunc_notify_trigger_strict()* for each condition.

iofunc_notify_remove()*, *iofunc_notify_remove_strict()

Removes notification entries from the list. Call *iofunc_notify_remove_strict()* when the client closes its file descriptor.

CAUTION:

In a multi-threaded server, you must serialize access to each `iofunc_notify_t` structure. In many cases, this is done implicitly by the default locking on the extended attribute structure that holds device-specific data and notification data copied from that other structure. (For an example of this implicit locking, see the *io_write handler* in the coding example shown in the next section.) But if an `iofunc_notify_t` structure is accessed somewhere that this default locking has not occurred (e.g., in an interrupt-handling thread), then explicit locking must be done, either of the extended attribute structure or with an explicitly added lock for the `iofunc_notify_t` structure.

Failure to serialize access to these structures can lead to server-side race conditions, which can introduce bugs that are difficult to detect and fix.

Don't return _RESMGR_NOREPLY from an *io_notify* handler, as it may be called multiple times from a single message if handling multiple file descriptors from a client call to *select()* or *poll()*. This is handled for you if you're using *iofunc_notify()*.

Sample code for handling _IO_NOTIFY messages

You can add the following code samples to either of the examples provided in the "*Simple device resource manager examples*" section of the Bones of a Resource Manager chapter. Both of those code samples provided the name **/dev/sample**. With the changes indicated below, clients can use writes to send it data, which it'll store as discrete messages. Other clients can use either *ionotify()* or *select()* to request notification when that data arrives. When clients receive notification, they can issue reads to get the data.

You'll need to replace this code that's located above the *main()* function:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

static resmgr_connect_funcs_t    connect_funcs;
static resmgr_io_funcs_t         io_funcs;
static iofunc_attr_t             attr;
```

with the following:

```
struct device_attr_s;
#define IOFUNC_ATTR_T   struct device_attr_s

#include <sys/iofunc.h>
#include <sys/dispatch.h>

/*
 * Define a structure and variables for storing the data that
 * is received. When clients write data to us, we store it here.
 * When clients do reads, we get the data from here.  Result: a
 * simple message queue.
*/
typedef struct item_s {
    struct item_s   *next;
    char            *data;
} item_t;

/* the extended attributes structure */
typedef struct device_attr_s {
    iofunc_attr_t   attr;
    iofunc_notify_t notify[3];  /* notification list used by
                                   iofunc_notify*() */
    item_t          *firstitem; /* the queue of items */
    int             nitems;     /* number of items in the queue */
} device_attr_t;

/* We only have one device; device_attr is its attribute structure */
```

```
static device_attr_t    device_attr;

int io_read( resmgr_context_t *ctp, io_read_t  *msg,
             RESMGR_OCB_T *ocb);
int io_write( resmgr_context_t *ctp, io_write_t *msg,
              RESMGR_OCB_T *ocb);
int io_notify( resmgr_context_t *ctp, io_notify_t *msg,
               RESMGR_OCB_T *ocb);
int io_close_dup( resmgr_context_t *ctp, io_close_t* msg,
                  RESMGR_OCB_T *ocb);

static resmgr_connect_funcs_t  connect_funcs;
static resmgr_io_funcs_t       io_funcs;
```

We need a place to keep data that's specific to our device. A good place for this is in an attribute structure that we can associate with the name we registered: **/dev/sample**. So, in the code above, we defined `device_attr_t` and `IOFUNC_ATTR_T` for this purpose. We talk more about this type of device-specific attribute structure in the *Extending the POSIX-Layer Data Structures* chapter.

We need two types of device-specific data:

- an array of three notification lists, one for each possible condition that a client can ask to be notified about. In `device_attr_t`, we called this *notify*.
- a queue to keep the data that gets written to us, and that we use to reply to a client. For this, we defined `item_t`; it's a type that contains data for a single item, as well as a pointer to the next `item_t`. In `device_attr_t` we use *firstitem* (points to the first item in the queue), and *nitems* (number of items).

Note that we removed the definition of *attr*, since we use *device_attr* instead.

Of course, we have to give the resource manager library the address of our handlers so that it'll know to call them. In the code for *main()* where we called *iofunc_func_init()*, we'll add the following code to register our handlers:

```
/* initialize functions for handling messages */
iofunc_func_init(_RESMGR_CONNECT_NFUNCS, &connect_funcs,
                 _RESMGR_IO_NFUNCS, &io_funcs);

/* For handling _IO_NOTIFY, sent as a result of client
   calls to ionotify() and select() */
io_funcs.notify = io_notify;

io_funcs.write = io_write;
io_funcs.read = io_read;
io_funcs.close_dup = io_close_dup;
```

And, since we're using *device_attr* in place of *attr*, we need to change the code wherever we use it in *main()*. So, you'll need to replace this code:

```
/* initialize attribute structure used by the device */
iofunc_attr_init(&attr, S_IFNAM | 0666, 0, 0);

/* attach our device name */
id = resmgr_attach(dpp,            /* dispatch handle        */
                   &resmgr_attr,   /* resource manager attrs */
```

```
                    "/dev/sample",  /* device name           */
                    _FTYPE_ANY,     /* open type             */
                    0,              /* flags                 */
                    &connect_funcs, /* connect routines      */
                    &io_funcs,      /* I/O routines          */
                    &attr);         /* handle                */
```

with the following:

```
/* initialize attribute structure used by the device */
iofunc_attr_init(&device_attr.attr, S_IFNAM | 0666, 0, 0);
IOFUNC_NOTIFY_INIT(device_attr.notify);
device_attr.firstitem = NULL;
device_attr.nitems = 0;

/* attach our device name */
id = resmgr_attach(dpp,            /* dispatch handle        */
                   &resmgr_attr,   /* resource manager attrs */
                   "/dev/sample",  /* device name            */
                   _FTYPE_ANY,     /* open type              */
                   0,              /* flags                  */
                   &connect_funcs, /* connect routines       */
                   &io_funcs,      /* I/O routines           */
                   &device_attr);  /* handle                 */
```

Note that we set up our device-specific data in *device_attr*. And, in the call to *resmgr_attach()*, we passed `&device_attr` (instead of `&attr`) for the handle parameter.

Now, you need to include the new handler function to handle the _IO_NOTIFY message:

```
int
io_notify( resmgr_context_t *ctp, io_notify_t *msg,
           RESMGR_OCB_T *ocb)
{
    device_attr_t   *dattr = (device_attr_t *) ocb->attr;
    int             trig;

    /*
     * 'trig' will tell iofunc_notify() which conditions are
     * currently satisfied.  'dattr->nitems' is the number of
     * messages in our list of stored messages.
    */

    trig = _NOTIFY_COND_OUTPUT; /* clients can always give us data */
    if (dattr->nitems > 0)
        trig |= _NOTIFY_COND_INPUT; /* we have some data available */

    /*
     * iofunc_notify() will do any necessary handling, including
     * adding the client to the notification list if need be.
    */

    return (iofunc_notify( ctp, msg, dattr->notify, trig,
                           NULL, NULL));
}
```

As stated above, our *io_notify* handler will be called when a client calls *ionotify()* or *select()*. In our handler, we're expected to remember who those clients are, and what conditions they want to be notified about. We should also be able to respond immediately with conditions that are already true. The *iofunc_notify()* helper function makes this easy.

The first thing we do is to figure out which of the conditions we handle have currently been met. In this example, we're always able to accept writes, so in the code above we set the *_NOTIFY_COND_OUTPUT* bit in *trig*. We also check *nitems* to see if we have data and set the _NOTIFY_COND_INPUT if we do.

We then call *iofunc_notify()*, passing it the message that was received (*msg*), the notification lists (*notify*), and which conditions have been met (*trig*). If one of the conditions that the client is asking about has been met, and the client wants us to poll for the condition before arming, then *iofunc_notify()* will return with a value that indicates what condition has been met and the condition will not be armed. Otherwise, the condition will be armed. In either case, we'll return from the handler with the return value from *iofunc_notify()*.

Earlier, when we talked about the three possible conditions, we mentioned that if you specify _NOTIFY_COND_INPUT, the client is notified when there's one or more units of input data available and that the number of units is up to you. We said a similar thing about _NOTIFY_COND_OUTPUT and _NOTIFY_COND_OBAND. In the code above, we let the number of units for all these default to 1. If you want to use something different, then you must declare an array such as:

```
int notifycounts[3] =  { 10, 2, 1 };
```

This sets the units for: _NOTIFY_COND_INPUT to 10; _NOTIFY_COND_OUTPUT to 2; and _NOTIFY_COND_OBAND to 1. We would pass *notifycounts* to *iofunc_notify()* as the second to last parameter.

Then, as data arrives, we notify whichever clients have asked for notification. In this sample, data arrives through clients sending us _IO_WRITE messages and we handle it using an *io_write* handler.

```
int
io_write(resmgr_context_t *ctp, io_write_t *msg,
         RESMGR_OCB_T *ocb)
{
    device_attr_t   *dattr = (device_attr_t *) ocb->attr;
    int             status;
    item_t          *newitem;

    if ((status = iofunc_write_verify(ctp, msg, ocb, NULL))
         != EOK)
        return (status);

    if ((msg->i.xtype & _IO_XTYPE_MASK) != _IO_XTYPE_NONE)
        return (ENOSYS);

    if (msg->i.nbytes > 0) {

        /* Get and store the data */

        if ((newitem = malloc(sizeof(item_t))) == NULL)
            return (errno);
        if ((newitem->data = malloc(msg->i.nbytes+1)) ==
```

```
            NULL) {
            free(newitem);
            return (errno);
        }
        /* reread the data from the sender's message buffer */
        resmgr_msgread(ctp, newitem->data, msg->i.nbytes,
                       sizeof(msg->i));
        newitem->data[msg->i.nbytes] = '\0';

        if (dattr->firstitem)
            newitem->next = dattr->firstitem;
        else
            newitem->next = NULL;
        dattr->firstitem = newitem;
        dattr->nitems++;

        /*
         * notify clients who may have asked to be notified
         * when there is data
        */

        if (IOFUNC_NOTIFY_INPUT_CHECK(dattr->notify,
            dattr->nitems, 0))
            iofunc_notify_trigger(dattr->notify, dattr->nitems,
                                  IOFUNC_NOTIFY_INPUT);
    }

    /* set up the number of bytes (returned by client's
       write()) */

    _IO_SET_WRITE_NBYTES(ctp, msg->i.nbytes);

    if (msg->i.nbytes > 0)
        ocb->attr->attr.flags |= IOFUNC_ATTR_MTIME |
                                 IOFUNC_ATTR_CTIME;

    return (_RESMGR_NPARTS(0));
}
```

The important part of the above *io_write* handler is the code within the following section:

```
if (msg->i.nbytes > 0) {
    ....
}
```

Here we first allocate space for the incoming data, and then use *resmgr_msgread()* to copy the data from the client's send buffer into the allocated space. Then, we add the data to our queue.

Next, we pass the number of input units that are available to *IOFUNC_NOTIFY_INPUT_CHECK()* to see if there are enough units to notify clients about. This is checked against the *notifycounts* that we mentioned above when talking about the *io_notify* handler. If there are enough units available then we call *iofunc_notify_trigger()* telling it that *nitems* of data are available (IOFUNC_NOTIFY_INPUT means input is available). The *iofunc_notify_trigger()* function checks the lists of clients asking for notification (*notify*) and notifies any that asked about data being available.

Any client that gets notified will then perform a read to get the data. In our sample, we handle this with the following *io_read* handler:

```
int
io_read(resmgr_context_t *ctp, io_read_t *msg, RESMGR_OCB_T *ocb)
{
    device_attr_t   *dattr = (device_attr_t *) ocb->attr;
    int             status;

    if ((status = iofunc_read_verify(ctp, msg, ocb, NULL)) != EOK)
        return (status);

    if ((msg->i.xtype & _IO_XTYPE_MASK) != _IO_XTYPE_NONE)
        return (ENOSYS);

    if (dattr->firstitem) {
        size_t  nbytes;
        item_t  *item, *prev;

        /* get last item */
        item = dattr->firstitem;
        prev = NULL;
        while (item->next != NULL) {
            prev = item;
            item = item->next;
        }

        /*
         * figure out number of bytes to give, write the data to the
         * client's reply buffer, even if we have more bytes than they
         * are asking for, we remove the item from our list
        */
        nbytes = min (strlen (item->data), msg->i.nbytes);

        /* set up the number of bytes (returned by client's read()) */
        _IO_SET_READ_NBYTES (ctp, nbytes);

        /*
         * write the bytes to the client's reply buffer now since we
         * are about to free the data
        */
        resmgr_msgwrite (ctp, item->data, nbytes, 0);

        /* remove the data from the queue */
        if (prev)
            prev->next = item->next;
        else
            dattr->firstitem = NULL;
        free(item->data);
        free(item);
        dattr->nitems--;
    } else {
        /* the read() will return with 0 bytes */
        _IO_SET_READ_NBYTES (ctp, 0);
```

```
    }

    /* mark the access time as invalid (we just accessed it) */

    if (msg->i.nbytes > 0)
        ocb->attr->attr.flags |= IOFUNC_ATTR_ATIME;

    return (EOK);
}
```

The important part of the above *io_read* handler is the code within this section:

```
if (firstitem) {
     ....
}
```

We first walk through the queue looking for the oldest item. Then we use *resmgr_msgwrite()* to write the data to the client's reply buffer. We do this now because the next step is to free the memory that we're using to store that data. We also remove the item from our queue.

Lastly, if a client closes its file descriptor, we must remove the client from our list. This is done using a *io_close_dup* handler:

```
int io_close_dup( resmgr_context_t *ctp, io_close_t* msg,
                  RESMGR_OCB_T *ocb)
{
    device_attr_t   *dattr = (device_attr_t *) ocb->attr;

    /*
     * A client has closed its file descriptor or has terminated.
     * Unblock any threads waiting for notification, then
     * remove the client from the notification list.
     */

    iofunc_notify_trigger_strict( ctp, dattr->notify, INT_MAX, IOFUNC_NOTIFY_INPUT);
    iofunc_notify_trigger_strict( ctp, dattr->notify, INT_MAX, IOFUNC_NOTIFY_OUTPUT );
    iofunc_notify_trigger_strict( ctp, dattr->notify, INT_MAX, IOFUNC_NOTIFY_OBAND );

    iofunc_notify_remove(ctp, dattr->notify);

    return (iofunc_close_dup_default(ctp, msg, ocb));
}
```

In the *io_close_dup* handler, we called *iofunc_notify_remove()* and passed it *ctp* (contains the information that identifies the client) and *notify* (contains the list of clients) to remove the client from the lists.

Handling out-of-band (_IO_MSG) messages

An _IO_MSG message lets a client send an "out-of-band" or control message to a resource manager, by way of a file descriptor. This interface is more general than an *ioctl()* or *devctl()*, but less portable.

The format of the message is specific to the resource manager, aside from the header, which we'll look at shortly. The client program sets up the message and uses *MsgSend()* to send it to the resource manager. The resource manager must set up an *io_msg* handler in order to receive the message; there isn't a default handler.

The message header is defined in **<sys/iomsg.h>** and looks like this:

```
struct _io_msg {
    uint16_t    type;
    uint16_t    combine_len;
    uint16_t    mgrid;
    uint16_t    subtype;
};
```

The fields include:

type

_IO_MSG

combine_len

Set this to `sizeof (struct _io_msg)`.

mgrid

A unique ID for your resource manager. The **<sys/iomgr.h>** header file defines some IDs that are reserved for various QNX Neutrino resource managers. You can use an ID in the range from _IOMGR_PRIVATE_BASE through _IOMGR_PRIVATE_MAX for your resource manager.

subtype

Use this field to distinguish different types of _IO_MSG messages that you want your resource manager to handle.

Any data should follow this header. For example:

```
typedef struct {
    struct _io_msg hdr;

    /* Add any required data fields here. */

} my_msg_t;
```

The client program would then do something like this:

```
#define MY_MGR_ID (_IOMGR_PRIVATE_BASE + 22)

my_msg_t msg, my_reply;
int fd;
```

```
long status;

fd = open ("/dev/sample", O_RDWR);

msg.hdr.type = _IO_MSG;
msg.hdr.combine_len = sizeof( msg.hdr );
msg.hdr.mgrid = MY_MGR_ID;
msg.hdr.subtype = 0;

/* Fill in the additional fields as required. */

status = MsgSend( fd, &msg, sizeof( msg ), &my_reply,
                  sizeof (my_reply));
```

The resource manager registers a function to handle the _IO_MSG messages:

```
/* Initialize the functions for handling messages */
iofunc_func_init(_RESMGR_CONNECT_NFUNCS, &connect_funcs,
                 _RESMGR_IO_NFUNCS, &io_funcs);

io_funcs.msg = my_io_msg;
```

This handler processes the message as appropriate. For example:

```
int my_io_msg (resmgr_context_t *ctp, io_msg_t *msg,
               RESMGR_OCB_T *ocb)
{
    my_msg_t my_msg;

    MsgRead (ctp->rcvid, &my_msg, sizeof (my_msg), 0);

    if (my_msg.hdr.mgrid != MY_MGR_ID)
    {
        return (ENOSYS);
    }

    /* Process the data as required. */

    /* Reply if necessary and tell the library that we've
       already replied. */

    MsgReply( ctp->rcvid, 0, &my_reply, sizeof(my_reply));
    return (_RESMGR_NOREPLY);
}
```

Note that the handler returns ENOSYS if the *mgrid* member of the header isn't the correct manager ID. This handler replies to the client, and then returns _RESMGR_NOREPLY to tell the library that there's no need for it to do the reply.

Handling private messages and pulses

A resource manager may need to receive and handle pulses, perhaps because an interrupt handler has returned a pulse or some other thread or process has sent a pulse.

The main issue with pulses is that they have to be received as a *message*. This means that a thread has to explicitly perform a *MsgReceive()* in order to get the pulse. But unless this pulse is sent to a different channel than the one that the resource manager is using for its main messaging interface, it will be received *by the library*. Therefore, we need to see how a resource manager can associate a pulse code with a handler routine and communicate that information to the library.

You can use the *pulse_attach()* function to associate a pulse code with a handler function. When the dispatch layer receives a pulse, it will look up the pulse code and see which associated handler to call to handle the pulse message.

In this example, we create the same resource manager, but this time we also attach a pulse, which is then used as a timer event:

```
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <string.h>

#define THREAD_POOL_PARAM_T     dispatch_context_t
#include <sys/iofunc.h>
#include <sys/dispatch.h>

static resmgr_connect_funcs_t   connect_func;
static resmgr_io_funcs_t        io_func;
static iofunc_attr_t            attr;

int
timer_tick( message_context_t *ctp, int code, unsigned flags,
            void *handle)
{
    union sigval value = ctp->msg->pulse.value;

    /* Do some useful work whenever the timer expires. */
    printf("received timer event, value %d\n", value.sival_int);

    return 0;
}

int
main(int argc, char **argv) {
    thread_pool_attr_t    pool_attr;
    struct sigevent       event;
    struct _itimer        itime;
    dispatch_t            *dpp;
    thread_pool_t         *tpp;
    int                   timer_id;
    int                   id;
```

```
dpp = dispatch_create();
if(dpp == NULL) {
    fprintf(stderr, "%s: Unable to allocate dispatch handle.\n", argv[0]);
    return EXIT_FAILURE;
}

memset(&pool_attr, 0, sizeof pool_attr);
pool_attr.handle = dpp;
/*  We are doing resmgr and pulse-type attaches.
 *
 *  If you're going to use custom messages or pulses with
 *  the message_attach() or pulse_attach() functions,
 *  then you MUST use the dispatch functions
 *  dispatch_block(), dispatch_handler(), and so on.
 */
pool_attr.context_alloc = dispatch_context_alloc;
pool_attr.block_func = dispatch_block;
pool_attr.unblock_func = dispatch_unblock;
pool_attr.handler_func = dispatch_handler;
pool_attr.context_free = dispatch_context_free;
pool_attr.lo_water = 2;
pool_attr.hi_water = 4;
pool_attr.increment = 1;
pool_attr.maximum = 50;

tpp = thread_pool_create(&pool_attr, POOL_FLAG_EXIT_SELF);
if(tpp == NULL) {
    fprintf(stderr, "%s: Unable to initialize thread pool.\n",argv[0]);
    return EXIT_FAILURE;
}

iofunc_func_init(_RESMGR_CONNECT_NFUNCS, &connect_func, _RESMGR_IO_NFUNCS,
                 &io_func);
iofunc_attr_init(&attr, S_IFNAM | 0666, 0, 0);

id = resmgr_attach(dpp, NULL, "/dev/sample", _FTYPE_ANY, 0, &connect_func, &io_func,
                   &attr);
if(id == -1) {
    fprintf(stderr, "%s: Unable to attach name.\n", argv[0]);
    return EXIT_FAILURE;
}

/* Initialize an event structure, and attach a pulse to it */
event.sigev_code = pulse_attach(dpp, MSG_FLAG_ALLOC_PULSE, 0, &timer_tick, NULL);
if(event.sigev_code == -1) {
    fprintf(stderr, "Unable to attach timer pulse.\n");
     return EXIT_FAILURE;
}

/* Connect to our channel */
event.sigev_coid = message_connect(dpp, MSG_FLAG_SIDE_CHANNEL);
if(event.sigev_coid == -1) {
    fprintf(stderr, "Unable to attach to channel.\n");
    return EXIT_FAILURE;
```

```
    }

    event.sigev_notify = SIGEV_PULSE;
    event.sigev_priority = -1;
    /* We could create several timers and use different sigev values for each */
    event.sigev_value.sival_int = 0;

    timer_id = TimerCreate(CLOCK_MONOTONIC, &event);
    if(timer_id == -1) {;
        fprintf(stderr, "Unable to attach channel and connection.\n");
        return EXIT_FAILURE;
    }

    /* And now set up our timer to fire every second */
    itime.nsec = 1000000000;
    itime.interval_nsec = 1000000000;
    TimerSettime(timer_id, 0, &itime, NULL);

    /* Never returns */
    thread_pool_start(tpp);
    return EXIT_SUCCESS;
}
```

We can either define our own pulse code (e.g., `#define OurPulseCode 57`) and pass it to *pulse_attach()*, or we can specify MSG_FLAG_ALLOC_PULSE in the *flags* argument to ask the function to dynamically generate a pulse code.

You can also use the path that a resource manager registers to get a connection ID (coid) that you can use with *MsgSend()* to send messages to your resource manager. This means that you don't have to use *read()* and *write()* to interact with a resource manager.

This example consists of client and server programs. Let's begin with the server:

```
/*
 * ResMgr and Message Server Process
 */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/neutrino.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>

resmgr_connect_funcs_t  ConnectFuncs;
resmgr_io_funcs_t       IoFuncs;
iofunc_attr_t           IoFuncAttr;


typedef struct
{
    uint16_t msg_no;
    char     msg_data[255];
```

```
} server_msg_t;


int message_callback( message_context_t *ctp, int type, unsigned flags,
                      void *handle )
{
    server_msg_t *msg;
    int num;
    char msg_reply[255];

    /* Cast a pointer to the message data */
    msg = (server_msg_t *)ctp->msg;

    /* Print some useful information about the message */
    printf( "\n\nServer Got Message:\n" );
    printf( "  type: %d\n" , type );
    printf( "  data: %s\n\n", msg->msg_data );

    /* Build the reply message */
    num = type - _IO_MAX;
    snprintf( msg_reply, 254, "Server got message code: _IO_MAX + %d", num );

    /* Send a reply to the waiting (blocked) client */
    MsgReply( ctp->rcvid, EOK, msg_reply, strlen( msg_reply ) + 1 );

    return 0;
}

int main( int argc, char **argv )
{
    dispatch_t           *dpp;
    dispatch_context_t   *ctp, *ctp_ret;
    int                  resmgr_id, message_id;

    /* Create the dispatch interface */
    dpp = dispatch_create();
    if( dpp == NULL )
    {
        fprintf( stderr, "dispatch_create() failed: %s\n",
                 strerror( errno ) );
        return EXIT_FAILURE;
    }

    /* Set up the default I/O functions to handle open/read/write/... */
    iofunc_func_init( _RESMGR_CONNECT_NFUNCS, &ConnectFuncs,
                      _RESMGR_IO_NFUNCS, &IoFuncs );

    /* Set up the attribute for the entry in the filesystem */
    iofunc_attr_init( &IoFuncAttr, S_IFNAM | 0666, 0, 0 );

    resmgr_id = resmgr_attach( dpp, NULL, "serv", _FTYPE_ANY,
                               0, &ConnectFuncs, &IoFuncs, &IoFuncAttr );
    if( resmgr_id == -1 )
    {
```

```
        fprintf( stderr, "resmgr_attach() failed: %s\n", strerror( errno ) );
        return EXIT_FAILURE;
    }

    /* Attach a callback (handler) for two message types */
    message_id = message_attach( dpp, NULL, _IO_MAX + 1,
                                 _IO_MAX + 2, message_callback, NULL );
    if( message_id == -1 )
    {
        fprintf( stderr, "message_attach() failed: %s\n", strerror( errno ) );
        return EXIT_FAILURE;
    }

    /* Set up a context for the dispatch layer to use */
    ctp = dispatch_context_alloc( dpp );
    if( ctp == NULL )
    {
        fprintf( stderr, "dispatch_context_alloc() failed: %s\n",
                 strerror( errno ) );
        return EXIT_FAILURE;
    }


    /* Get and process messages */
    while( 1 )
    {
        ctp_ret = dispatch_block( ctp );
        if( ctp_ret )
        {
            dispatch_handler( ctp );
        }
        else
        {
            fprintf( stderr, "dispatch_block() failed: %s\n",
                     strerror( errno ) );
            return EXIT_FAILURE;
        }
    }

    return EXIT_SUCCESS;
}
```

When the server calls *resmgr_attach()*, it registers the filesystem entry **serv**. Since the server doesn't specify an absolute path, the entry appears in the directory where the server was run. This gives us a filesystem entry that can be opened and closed, but generally behaves the same as **/dev/null**.

We call *message_attach()* 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. All incoming messages must have an unsigned 16-bit integer at the start indicating the message type. Note that the range `0x0` to _IO_MAX is reserved for the OS. We set up our *message_callback()* routine to handle messages of type _IO_MAX + 1 and _IO_MAX + 2. You can specify a pointer to arbitrary data to pass to the callback, but we don't need that, so we set it to NULL.

When a message of the appropriate type is received, our *message_callback()* routine is 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 prints the message type and the string that was sent from the client. It then prints the offset from _IO_MAX of the message type, and then formats a reply string and sends the reply back to the client via *ctp->rcvid* using *MsgReply()*.

The client is much simpler:

```
/*
 * Message Client Process
 */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>
#include <sys/neutrino.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>

typedef struct
{
    uint16_t msg_no;
    char msg_data[255];
} client_msg_t;

int main( int argc, char **argv )
{
    int fd;
    int c;
    client_msg_t msg;
    long ret;
    int num;
    char msg_reply[255];

    num = 1;

    /* Process any command-line arguments */
    while( ( c = getopt( argc, argv, "n:" ) ) != -1 )
    {
        if( c == 'n' )
        {
            num = strtol( optarg, 0, 0 );
        }
    }
    /* Open a connection to the server (fd == coid) */
    fd = open( "serv", O_RDWR );
    if( fd == -1 )
    {
        fprintf( stderr, "Unable to open server connection: %s\n",
            strerror( errno ) );
        return EXIT_FAILURE;
```

```
    }

    /* Clear the memory for the msg and the reply */
    memset( &msg, 0, sizeof( msg ) );
    memset( &msg_reply, 0, sizeof( msg_reply ) );

    /* Set up the message data to send to the server */
    msg.msg_no = _IO_MAX + num;
    snprintf( msg.msg_data, 254, "client %d requesting reply.", getpid() );

    printf( "client: msg_no: _IO_MAX + %d\n", num );
    fflush( stdout );

    /* Send the data to the server and get a reply */
    ret = MsgSend( fd, &msg, sizeof( msg ), msg_reply, 255 );
    if( ret == -1 )
    {
        fprintf( stderr, "Unable to MsgSend() to server: %s\n", strerror( errno ) );
        return EXIT_FAILURE;
    }

    /* Print out the reply data */
    printf( "client: server replied: %s\n", msg_reply );

    close( fd );

    return EXIT_SUCCESS;
}
```

Remember that since the server registers a *relative* pathname, the client must be run from the same directory as the server.

The client 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 then waits for the reply. When the reply comes back, the client prints the reply data.

You can give the client the command-line option `-n` *offset* to specify the offset from _IO_MAX to use for the message type. If you give anything other than 1 or 2 as the offset, the *MsgSend()* fails, since the server hasn't set up handlers for those messages.

Handling *open()*, *dup()*, and *close()* messages

The resource manager library provides another convenient service for us: it knows how to handle *dup()* messages.

Suppose that the client executed code that eventually ended up performing:

```
fd = open ("/dev/sample", O_RDONLY);
…
fd2 = dup (fd);
…
fd3 = dup (fd);
…
close (fd3);
…
close (fd2);
…
close (fd);
```

Our resource manager would get an _IO_CONNECT message for the first *open()*, followed by two _IO_DUP messages for the two *dup()* calls. Then, when the client executed the *close()* calls, we would get three _IO_CLOSE messages.

Since the *dup()* functions generate duplicates of the file descriptors, we don't want to allocate new OCBs for each one. And since we're not allocating new OCBs for each *dup()*, we don't want to *release* the memory in each _IO_CLOSE message when the _IO_CLOSE messages arrive! If we did that, the first close would wipe out the OCB.

The resource manager library knows how to manage this for us; it keeps count of the number of _IO_DUP and _IO_CLOSE messages sent by the client. Only on the *last* _IO_CLOSE message will the library synthesize a call to our _IO_CLOSE_OCB handler.

Some users of the library will want to have the default functions manage the _IO_DUP and _IO_CLOSE messages, but some will want to override the default actions. For instance, if you need to leave clients blocked, you could unblock the requests in a custom handler for _IO_CLOSE messages. An example of such a function is given in "*Unblocking if someone closes a file descriptor*".

Handling *mount()*

Mount requests can provide a very convenient and flexible interface for programs that need to enable and disable components of their resource managers' systems.

The main areas to consider when using and building mount functionality into your resource manager are:

- `mount` utility
- *mount()* function call
- *mount()* callout in the resource manager

These components represent the stream of communication for the mount request. Let's start in the middle with the *mount()* function call and work our way out.

The *mount()* function call is at the bottom of the utility and represents a client's access point to the resource manager. The function is implemented in the C library, defined in **<sys/mount.h>**, and described in the QNX Neutrino *C Library Reference*.

mount() function call

The prototype for *mount()* is as follows:

```
int mount( const char *special_device,
           const char *mount_directory,
           int flags,
           const char *mount_type,
           const void *mount_data,
           int mount_datalen);
```

The argument that we need to consider here is the *flags* field.

To support the mounting of non-existent special devices (such as NFS devices) or arbitrary strings (such as the name of shared object or DLL), we need to massage the arguments to this function slightly because the `mount` utility has two methods (`-T` and `-t`) for specifying the mount type.

In the general case where *special_device* is an actual device, a typical `mount` command may look like:

```
% mount -t qnx6 /dev/hd0t177 /mnt/fs
```

In this case the *special_device* is **/dev/hd0t177**, the *mount_directory* is **/mnt/fs**, and the mount type is `qnx6`. In this case, the mount request should be directed only to the process responsible for managing the *special_device*. That is to say the resource manager that has provided the **/dev/hd0t177** path into the pathname space. In this type of scenario, the resource manager is given an OCB for the *special_device* (**/dev/hd0t177**), rather than the string `/dev/hd0t177`. This simplifies the processing in the resource manager since having the OCB, which is an internal data pointer, for the special device implies that the server doesn't have to recursively communicate with itself to get a handle for the device.

A less frequently used, but very useful case, is where the *special_device* isn't an actual device. For example:

```
% mount -T io-pkt /lib/dll/devnp-e1000.so
```

Note that the mountpoint is missing from the command line. In this case, NULL (or **/**) acts as an implied *mount_directory*, which causes the process handling the request (i.e., the currently running variant of `io-pkt`) to take the appropriate action when it receives the mount request. The special_device is **/lib/dll/devnp-e1000.so** and the type is `io-pkt`.

In this case, you want to avoid having the special device interpreted as being provided by the same process that will handle the mount request. So while the file **/lib/dll/devnp-e1000.so** is probably handled by some filesystem process, we're actually interested in mounting a network interface that's managed by the `io-pkt` process. Ideally, the mount callout will receive only the special device string **/lib/dll/devnp-e1000.so**, and not the OCB for the device.

The behavioral difference between the `-t` and `-T` options for the `mount` utility can be obtained by ORing in _MFLAG_OCB to the standard *mount() flags* parameter. If you don't want to use the OCB method of performing the mount request, use `-T`, which we translate to _MFLAG_OCB.

Mount requests are connection requests, which means they operate on a path in the same way that the *open()* or *unlink()* calls do. The requests are sent along the path specified by *dir*. When a resource manager receives a request to mount something, the information is already provided in the same way that it would be for an *open()* for creation request, namely in the *msg->connect.path* variable.

For more information on the name-resolution process, see "*Handling private messages and pulses*" in this chapter.

Mount in the resource manager

Your resource manager will be called upon to perform a mount request via the *mount* function callout in the `resmgr_connect_funcs_t` structure, defined as:

```
int mount( resmgr_context_t *ctp,
           io_mount_t *msg,
           RESMGR_HANDLE_T *handle,
           io_mount_extra_t *extra);
```

The only field here that differs from the other connect functions is the `io_mount_extra_t` structure. It's defined in **<sys/iomsg.h>** as:

```
typedef struct _io_mount_extra {
    uint32_t flags; /* _MOUNT_? or ST_? flags above */
    uint32_t nbytes; /* Size of entire structure */
    uint32_t datalen; /* Length of the data structure following */
    uint32_t zero[1];

    union { /* If EXTRA_MOUNT_PATHNAME these set*/
        struct { /* Sent from client to resmgr framework */
            struct _msg_info32 info; /* Special info on first mount,
                                        path info on remount */

        } cl;
```

```
        struct { /* Server receives this structure filled in */
            void * ocb; /* OCB to the special device */
            void * data; /* Server specific data of len datalen */
            char * type; /* Character string with type information */
            char * special; /* Optional special device info */
            void * zero[4]; /* Padding */
        } srv;
    } extra;
} io_mount_extra_t;
```

This structure is provided with all of the pointers already resolved, so you can use it without doing any extra fiddling.

The members are:

flags

Flag fields provided to the `mount` command containing the common mount flags defined in **<sys/mount.h>**.

nbytes

Size of the entire mount-extra message:

```
sizeof(_io_mount_extra) + datalen + strlen(type)
+ 1 + strlen(special) + 1
```

datalen

Size of the *data* pointer.

info

Used by the resource manager layer.

ocb

OCB of the special device if it was requested via the _MOUNT_OCB flag. NULL otherwise.

data

Pointer to the user data of length *datalen*.

type

Null-terminated string containing the mount type, such as `nfs`, `cifs`, or `qnx6`.

special

Null-terminated string containing the special device if it was requested via the _MOUNT_SPEC flag. NULL otherwise.

In order to receive mount requests, the resource manager should register a NULL path with an FTYPE of _FTYPE_MOUNT and with the flags _RESMGR_FLAG_FTYPEONLY. This would be done with code that looks something like:

```
mntid = resmgr_attach(
            dpp, /* Dispatch pointer */
```

```
            &resmgr_attr, /* Dispatch attributes */
            NULL, /* Attach at "/" */

            /* We are a directory and want only matching ftypes */

            _FTYPE_MOUNT,
            _RESMGR_FLAG_DIR | _RESMGR_FLAG_FTYPEONLY,
            mount_connect, /* Only mount filled in */
            NULL, /* No io handlers */
            & handle); /* Handle to pass to mount callout */
```

Again, we're attaching at the root of the filesystem so that we'll be able to receive the full path of the new mount requests in the *msg->connect* structure.

Adding the _RESMGR_FLAG_FTYPEONLY flag ensures that this request is used only when there's an _FTYPE_MOUNT-style of connection. Once this is done, the resource manager is ready to start receiving mount requests from users.

An outline of a sample mount handler would look something like this:

```
int io_mount( ... ) {

   Do any sanity checks that you need to do.

   Check type against our type with strcmp(), since
   there may be no name for REMOUNT/UNMOUNT flags.

   Error with ENOENT out if no match.

   If no name, check the validity of the REMOUNT/UNMOUNT request.

   Parse arguments or set up your data structure.

   Check to see if we are remounting (_MOUNT_REMOUNT)

      Change flags, etc., if you can remount.
      Return EOK.

   Check to see if we are unmounting _MOUNT_UNMOUNT

      Change flags, etc., if you can unmount.
      Return EOK.

   Create a new node and attach it at the msg->connect.path
   point (unless some other path is implied based on the
   input variables and the resource manager) with resmgr_attach().

   Return EOK.
}
```

What's important to notice here is that each resource manager that registers a mount handler will potentially get a chance to examine the request to see if it can handle it. This means that you have to be rigorous in your type- and error-checking to make sure that the request is indeed destined for your manager. If your manager returns anything other than ENOSYS or ENOENT, it's assumed that the

request was valid for this manager, but there was some other sort of error. Only errors of ENOSYS or ENOENT cause the request to "fall through" to other resource managers.

When you unmount, you would perform any cleanup and integrity checks that you need, and then call *resmgr_detach()* with the *ctp->id* field. In general, you should support umounted calls only on the root of a mounted filesystem.

`mount` utility

By covering the *mount()* library function and the operation in the resource manager, we've pretty well covered the `mount` utility. The usage for the utility is shown here for reference:

```
mount [-wreuv] -t type [-o options] [special] mntpoint

mount [-wreuv] -T type [-o options] special [mntpoint]

mount
```

The options are:

`-t`

Indicates the special device, if it's present, is generally a real device and the same server will handle the mountpoint.

`-T`

Indicates the special device isn't a real device but rather a key for the server. The server will automatically create an appropriate mountpoint if *mntpoint* isn't specified.

`-v`

Increases the verbosity.

`-w`

Mount read/write.

`-r`

Mount read-only.

`-u`

Mount for update (remount).

However, if you're writing a mount handler, there may be occasions when you want to do custom parsing of arguments and provide your own data structure to your server. This is why the mount command will always first try and call out to a separate program named **mount**_*XXX*, where *XXX* is the type that you specified with the `-t` option. To see just what would be called (in terms of options, etc.), you can use the `-v` option, which should provide you with the command line that would be *exec()*'ed.

In order to help with the argument parsing, there's a utility function, *mount_parse_generic_args()*, that you can call to process the common options. The function is defined in **<sys/mount.h>** as:

```
char *mount_parse_generic_args(char *options, int *flags);
```

This function parses the given *options*, removes any options that it recognizes, and sets or clears the appropriate bits in the *flags*. It returns a pointer to the modified version of *options* containing any options that it didn't recognize, or NULL if it recognized all the options. You use *mount_parse_generic_args()* like this:

```
while ((c = getopt(argv, argc, "o:"))) {
   switch (c) {

      case 'o':

        if ((mysteryop = mount_parse_generic_args(optarg, &flags))) {

           /* You can do your own getsubopt-type processing here.
              The common options are removed from mysteryop. */
        }

        break;
    }
}
```

For more information about the stripped options and the corresponding flags, see the entry for *mount_parse_generic_args()*, in the QNX Neutrino *C Library Reference*.

Handling *stat()*

Your resource manager will receive an _IO_STAT message when a client calls *stat()*, *lstat()*, or *fstat()*. You usually don't need to provide your own handler for this message. The prototype for the *io_stat* handler is as follows:

```
int io_stat ( resmgr_context_t *ctp,
              io_stat_t *msg,
              RESMGR_OCB_T *ocb)
```

The default handler for the _IO_STAT message, *iofunc_stat_default()*, calls *iofunc_time_update()* to ensure that the time entries in the *ocb->attr* structure are current and valid, and then calls the *iofunc_stat_format()* helper function to fill in the `stat` structure based on the information in the *ocb->attr* structure.

The `io_stat_t` structure holds the _IO_STAT message received by the resource manager:

```
struct _io_stat {
    uint16_t                    type;
    uint16_t                    combine_len;
    union {
       uint32_t                 zero;
       uint32_t                 format;
    };
};

typedef union {
    struct _io_stat             i;
    struct __stat_t32_2001      o_t32_2001;
    struct __stat_t32_2008      o_t32_2008;
    struct __stat_t64_2008      o_t64_2008;
    struct stat                 o;
} io_stat_t;
```

As with all the I/O messages, this structure is a union of an input message (coming to the resource manager) and an output or reply message (going back to the client). The *i* member is a structure of type `_io_stat` that contains the following members:

type

_IO_STAT.

combine_len

If the message is a combine message, _IO_COMBINE_FLAG is set in this member. For more information, see the *Combine Messages* chapter of this guide.

format

(QNX Neutrino 7.0 or later) The form of the information; one of the following:

- _STAT_FORM_UNSET — unknown; this is assumed to be the same as _STAT_FORM_T32_2001
- _STAT_FORM_T32_2001 — 32-bit fields, POSIX 2001

- _STAT_FORM_T32_2008 — 32-bit fields, POSIX 2008
- _STAT_FORM_T64_2008 — 64-bit fields, POSIX 2008
- _STAT_FORM_SYS_2008 — _STAT_FORM_T32_2008 in programs compiled for a 32-bit architecture, or _STAT_FORM_T64_2008 in programs compiled for a 64-bit architecture.
- _STAT_FORM_PREFERRED — the preferred form: _STAT_FORM_T32_2001 in programs compiled for a 32-bit architecture, or _STAT_FORM_T64_2008 in programs compiled for a 64-bit architecture.

The *o** members are variants of a `struct stat`, corresponding to the requested form; for more information, see the entry for `struct stat` in the QNX Neutrino *C Library Reference*.

If you write your own handler, it should return the status via the helper macro *_RESMGR_STATUS()* and the `struct stat` via message reply.

If your resource manager is for a filesystem, you might want to include the `stat` information in the reply for other messages. For more information, see "*Returning directory entries from* _IO_READ" in the Filesystem Resource Managers chapter of this guide.

Handling *lseek()*

Your resource manager will receive an _IO_LSEEK message when a client calls *lseek()*, *fseek()*, or *rewinddir()*.

A resource manager that handles directories will also need to interpret the _IO_LSEEK message for directory operations.

The prototype for the *io_lseek* handler is as follows:

```
int io_lseek ( resmgr_context_t *ctp,
               io_lseek_t *msg,
               RESMGR_OCB_T *ocb)
```

The default handler, *iofunc_lseek_default()*, simply calls the *iofunc_lseek()* helper function.

The `io_lseek_t` structure is (once again), a union of an input message and an output message:

```
struct _io_lseek {
  uint16_t          type;
  uint16_t          combine_len;
  short             whence;
  uint16_t          flags;
  uint64_t          offset;
};

typedef union {
  struct _io_lseek i;
  uint64_t         o;
} io_lseek_t;
```

The *whence* and *offset* members are passed from the client's *lseek()* function. The routine should adjust the OCB's *offset* parameter after interpreting the *whence* and *offset* parameters from the message and should return the new offset or an error.

The only currently defined flag is _IO_LSEEK_IGNORE_NON_SEEKABLE. The client sets this flag if it wants the resource manager to ignore the request if lseek operations aren't supported (for example, an *aio_read()* on a pipe). In this case, the resource manager shouldn't change the offset.

The handler should return the status via the helper macro *_RESMGR_STATUS()*, and optionally (if no error occurred, and if the message isn't part of a combine message) the current offset.

Chapter 9
Unblocking Clients and Handling Interrupts

Your resource manager needs to avoid leaving clients blocked indefinitely, and it might need to handle interrupts.

Handling client unblocking due to signals or timeouts

Another convenient service that the resource manager library does for us is *unblocking*.

When a client issues a request (e.g., *read()*), this translates (via the client's C library) into a *MsgSend()* to our resource manager. The *MsgSend()* is a blocking call. If the client receives a signal during the time that the *MsgSend()* is outstanding, our resource manager needs to have some indication of this so that it can abort the request.

Because the library set the _NTO_CHF_UNBLOCK flag when it called *ChannelCreate()*, we'll receive a pulse whenever the client tries to unblock from a *MsgSend()* that we have *MsgReceive()*'d.

As an aside, recall that in the QNX Neutrino messaging model, the client can be in one of two states as a result of calling *MsgSend()*. If the server hasn't yet received the message (via the server's *MsgReceive()*), the client is in a *SEND-blocked* state—the client is waiting for the server to receive the message. When the server has actually received the message, the client transits to a *REPLY-blocked* state—the client is now waiting for the server to reply to the message (via *MsgReply()*).

When this happens and the pulse is generated, the resource manager library handles the pulse message and synthesizes an _IO_UNBLOCK message.

Looking through the `resmgr_io_funcs_t` and the `resmgr_connect_funcs_t` structures (see the QNX Neutrino *C Library Reference*), you'll notice that there are actually *two* `unblock` message handlers: one in the I/O functions structure and one in the connect functions structure.

Why two? Because we may get an abort in one of two places. We can get the abort pulse right after the client has sent the _IO_OPEN message (but before we've replied to it), or we can get the abort during an I/O message.

Once we've performed the handling of the _IO_CONNECT message, the I/O functions' unblock member will be used to service an unblock pulse. Therefore, if you're supplying your own *io_open* handler, be sure to set up all relevant fields in the OCB *before* you call *resmgr_open_bind()*; otherwise, your I/O functions' version of the unblock handler may get called with invalid data in the OCB. (Note that this issue of abort pulses “during” message processing arises only if there are multiple threads running in your resource manager. If there's only one thread, then the messages will be serialized by the library's *MsgReceive()* function.)

The effect of this is that if the client is SEND-blocked, the server doesn't need to know that the client is aborting the request, because the server hasn't yet received it.

Only in the case where the server has received the request and is performing processing on that request does the server need to know that the client now wishes to abort.

For more information on these states and their interactions, see the *MsgSend()*, *MsgReceive()*, *MsgReply()*, and *ChannelCreate()* functions in the QNX Neutrino *C Library Reference*; see also the chapter on Interprocess Communication in the *System Architecture* book.

If you're overriding the default unblock handler, *iofunc_unblock_default()*, you should always call the default handler to process any generic unblocking cases first (which it does by calling *iofunc_unblock()*). For example:

```
if((status = iofunc_unblock_default(...)) != _RESMGR_DEFAULT) {
    return status;
}
```

This ensures that any client waiting on a resource manager list (such as an advisory lock list) will be unblocked if possible.

Then you'll need some way to walk your table of blocked client rcvids, looking for a match, and unblocking them. That is done by replying to them; you aren't replying to the unblock request as such, but to the original client call (so either partially read the data or give an EINTR error as appropriate).

The routine should confirm the unblock is still pending (to avoid the race condition where the client was replied to by the time you got the unblock pulse, and the rcvid now indicates another client), by calling *MsgInfo()* and then checking for the _NTO_MI_UNBLOCK_REQ flag. If you can't find a matching client, you can ignore the unblock request by returning _RESMGR_NOREPLY:

```
/* Check if rcvid is still valid and still has an unblock
   request pending. */
if (MsgInfo(ctp->rcvid, &info) == -1 ||
    !(info.flags & _NTO_MI_UNBLOCK_REQ)) {
 return _RESMGR_NOREPLY;
}
```

If you don't provide an unblock handler, having your client thread left REPLY-blocked on the server is expected behavior; the server has to be given a chance to clean up client data structures when a client terminates.

Unblocking if someone closes a file descriptor

Suppose the following sequence occurs:

- A client opens a file descriptor and calls *read()* on it.
- The resource manager doesn't reply in the *io_read()* handler, so the client remains blocked.
- A second thread in the client calls *close()* for the file descriptor while the first thread is blocked on *read()*.

If your resource manager handles the disconnection without considering the *read()*, then the client's *read()* is indefinitely blocked, even after the fd has been detached. This could happen for other blocking operations, such as calls to *write()* and *devctl()*.

To avoid this situation, your resource manager needs to maintain a list of blocked clients. Whenever it blocks a client that's waiting for an operation to complete, you must add this client to the list—and you need to remove a client from the list when you unblock it.

The resource manager framework provides an *io_close_dup()* callout (see the Resource Managers chapter of *Getting Started with QNX Neutrino*) that gets called whenever *close()* is called on a file descriptor or a client otherwise disconnects. In this callout, you must traverse the list and use the server connection ID (*scoid*) and connection ID (*coid*) to determine which client threads are blocked on that file descriptor, and reply to them (via *MsgError()* or otherwise) to unblock them. For example:

```
int io_close_dup (resmgr_context_t *ctp, io_close_t *msg, RESMGR_OCB_T *ocb)
{
    // unblock any clients blocked on the file descriptor being closed
    blocked_client_t *client, *prev;
    prev = NULL;

    iofunc_lock_ocb_default(ctp, NULL, ocb);
    client = blocked_clients;
    while ( client != NULL )
    {
        if ( (client->coid == ctp->info.coid) && (client->scoid == ctp->info.scoid) )
        {
            MsgError( client -> rcvid, EBADF );
            if (prev != NULL) // anywhere but the head of the list
            {
                prev->next = client->next;
                free(client);
                client = prev->next;
            }
            else // head of the list case
            {
                blocked_clients = client->next;
                free(client);
                client = blocked_clients;
            }
        }
        else // no match, move to the next entry and track previous entry
        {
            prev = client;
```

```
            client = client->next;
        }
    }
    iofunc_unlock_ocb_default(ctp, NULL, ocb);

    // do rest of the regular close handling
    return iofunc_close_dup_default(ctp, msg, ocb );
}
```

CAUTION:

Unlike with most other I/O handlers, when the *io_close_dup()* handler runs, the attribute structure in the OCB (`ocb->attr`) is *not* locked by default. In your implementation, you may need to modify some fields in this structure. In this case, you must explicitly lock it to prevent server-side race conditions, which can cause bugs that are difficult to detect and fix.

Handling interrupts

Resource managers that manage an actual hardware resource will likely need to handle interrupts generated by the hardware. For a detailed discussion on strategies for interrupt handlers, see the chapter on Writing an Interrupt Handler in the QNX Neutrino *Programmer's Guide*.

How do interrupt handlers relate to resource managers? When a significant event happens within the interrupt handler, the handler needs to inform a thread in the resource manager. This is usually done via a pulse (discussed in the "*Handling private messages and pulses*" section of the Handling Other Messages chapter), but it can also be done with the SIGEV_INTR event notification type. Let's look at this in more detail.

When the resource manager starts up, it transfers control to *thread_pool_start()*. This function may or may not return, depending on the flags passed to *thread_pool_create()* (if you don't pass any flags, the function returns after the thread pool is created). This means that if you're going to set up an interrupt handler, you should do so *before* starting the thread pool, or use one of the strategies we discussed above (such as starting a thread for your entire resource manager).

However, if you're going to use the SIGEV_INTR event notification type, there's a catch: the thread that *attaches* the interrupt (via *InterruptAttach()* or *InterruptAttachEvent()*) must be the same thread that calls *InterruptWait()*.

Sample code for handling interrupts

Here's an example that includes relevant portions of the interrupt service routine and the handling thread:

```
#define INTNUM 0
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>
#include <sys/neutrino.h>

static resmgr_connect_funcs_t   connect_funcs;
static resmgr_io_funcs_t        io_funcs;
static iofunc_attr_t            attr;

void *
interrupt_thread (void * data)
{
    struct sigevent event;
    int             id;

    /* fill in "event" structure */
    memset(&event, 0, sizeof(event));
    event.sigev_notify = SIGEV_INTR;

    /* Enable the INTERRUPTEVENT ability */
```

```
    procmgr_ability(0,
        PROCMGR_ADN_ROOT|PROCMGR_AOP_ALLOW|PROCMGR_AID_INTERRUPTEVENT,
        PROCMGR_AID_EOL);

    /* intNum is the desired interrupt level */
    id = InterruptAttachEvent (INTNUM, &event, 0);

    /*... insert your code here ... */

    while (1) {
        InterruptWait (NULL, NULL);
        /*  do something about the interrupt,
         *  perhaps updating some shared
         *  structures in the resource manager
         *
         *  unmask the interrupt when done
         */
        InterruptUnmask(INTNUM, id);
    }
}

int
main(int argc, char **argv) {
    thread_pool_attr_t    pool_attr;
    resmgr_attr_t         resmgr_attr;
    dispatch_t            *dpp;
    thread_pool_t         *tpp;
    int                   id;


    if((dpp = dispatch_create()) == NULL) {
        fprintf(stderr,
                "%s: Unable to allocate dispatch handle.\n",
                argv[0]);
        return EXIT_FAILURE;
    }

    memset(&pool_attr, 0, sizeof pool_attr);
    pool_attr.handle = dpp;
    pool_attr.context_alloc = (void *) dispatch_context_alloc;
    pool_attr.block_func = (void *) dispatch_block;
    pool_attr.unblock_func = (void *) dispatch_unblock;
    pool_attr.handler_func = (void *) dispatch_handler;
    pool_attr.context_free = (void *) dispatch_context_free;
    pool_attr.lo_water = 2;
    pool_attr.hi_water = 4;
    pool_attr.increment = 1;
    pool_attr.maximum = 50;

    if((tpp = thread_pool_create(&pool_attr,
                                 POOL_FLAG_EXIT_SELF)) == NULL) {
        fprintf(stderr, "%s: Unable to initialize thread pool.\n",
                argv[0]);
        return EXIT_FAILURE;
```

```
    }

    iofunc_func_init(_RESMGR_CONNECT_NFUNCS, &connect_funcs,
                     _RESMGR_IO_NFUNCS, &io_funcs);
    iofunc_attr_init(&attr, S_IFNAM | 0666, 0, 0);

    memset(&resmgr_attr, 0, sizeof resmgr_attr);
    resmgr_attr.nparts_max = 1;
    resmgr_attr.msg_max_size = 2048;

    if((id = resmgr_attach(dpp, &resmgr_attr, "/dev/sample",
                           _FTYPE_ANY, 0,
                &connect_funcs, &io_funcs, &attr)) == -1) {
        fprintf(stderr, "%s: Unable to attach name.\n", argv[0]);
        return EXIT_FAILURE;
    }

    /* Start the thread that will handle interrupt events. */
    pthread_create (NULL, NULL, interrupt_thread, NULL);

    /* Never returns */
    thread_pool_start(tpp);
    return EXIT_SUCCESS;
}
```

Here the *interrupt_thread()* function uses *InterruptAttachEvent()* to bind the interrupt source (*intNum*) to the event (passed in *event*), and then waits for the event to occur.

This approach has a major advantage over using a pulse. A pulse is delivered as a message to the resource manager, which means that if the resource manager's message-handling threads are busy processing requests, the pulse will be queued until a thread does a *MsgReceive()*.

With the *InterruptWait()* approach, if the thread that's executing the *InterruptWait()* is of sufficient priority, it unblocks and runs *immediately* after the SIGEV_INTR is generated.

Chapter 10
Multithreaded Resource Managers

When you create a multithreaded resource manager, you should design it so that there's always at least one RECEIVE-blocked thread. For more information, see “Server boost” in the Interprocess Communication chapter of the *System Architecture* guide.

Multithreaded resource manager example

Let's look at our multithreaded resource manager example in more detail:

```
#include <errno.h>
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

/*
 * Define THREAD_POOL_PARAM_T such that we can avoid a compiler
 * warning when we use the dispatch_*() functions below
 */
#define THREAD_POOL_PARAM_T dispatch_context_t

#include <sys/iofunc.h>
#include <sys/dispatch.h>

static resmgr_connect_funcs_t    connect_funcs;
static resmgr_io_funcs_t         io_funcs;
static iofunc_attr_t             attr;

int main(int argc, char **argv)
{
    /* declare variables we'll be using */
    thread_pool_attr_t   pool_attr;
    resmgr_attr_t        resmgr_attr;
    dispatch_t           *dpp;
    thread_pool_t        *tpp;
    int                  id;

    /* initialize dispatch interface */
    if((dpp = dispatch_create()) == NULL) {
        fprintf(stderr, "%s: Unable to allocate dispatch handle.\n",
                argv[0]);
        return EXIT_FAILURE;
    }

    /* initialize resource manager attributes */
    memset(&resmgr_attr, 0, sizeof resmgr_attr);
    resmgr_attr.nparts_max = 1;
    resmgr_attr.msg_max_size = 2048;

    /* initialize functions for handling messages */
    iofunc_func_init(_RESMGR_CONNECT_NFUNCS, &connect_funcs,
                     _RESMGR_IO_NFUNCS, &io_funcs);

    /* initialize attribute structure used by the device */
    iofunc_attr_init(&attr, S_IFNAM | 0666, 0, 0);

    /* attach our device name */
```

```
    id = resmgr_attach(dpp,            /* dispatch handle        */
                       &resmgr_attr,   /* resource manager attrs */
                       "/dev/sample",  /* device name            */
                       _FTYPE_ANY,     /* open type              */
                       0,              /* flags                  */
                       &connect_funcs, /* connect routines       */
                       &io_funcs,      /* I/O routines           */
                       &attr);         /* handle                 */
    if(id == -1) {
        fprintf(stderr, "%s: Unable to attach name.\n", argv[0]);
        return EXIT_FAILURE;
    }

    /* initialize thread pool attributes */
    memset(&pool_attr, 0, sizeof pool_attr);
    pool_attr.handle = dpp;
    pool_attr.context_alloc = dispatch_context_alloc;
    pool_attr.block_func = dispatch_block;
    pool_attr.unblock_func = dispatch_unblock;
    pool_attr.handler_func = dispatch_handler;
    pool_attr.context_free = dispatch_context_free;
    pool_attr.lo_water = 2;
    pool_attr.hi_water = 4;
    pool_attr.increment = 1;
    pool_attr.maximum = 50;

    /* allocate a thread pool handle */
    if((tpp = thread_pool_create(&pool_attr,
                                 POOL_FLAG_EXIT_SELF)) == NULL) {
        fprintf(stderr, "%s: Unable to initialize thread pool.\n",
                argv[0]);
        return EXIT_FAILURE;
    }

    /* Start the threads. This function doesn't return. */
    thread_pool_start(tpp);
    return EXIT_SUCCESS;
}
```

The thread pool attribute (*pool_attr*) controls various aspects of the thread pool, such as which functions get called when a new thread is started or dies, the total number of worker threads, the minimum number, and so on.

Thread pool attributes

Here's the `_thread_pool_attr` structure:

```
typedef struct _thread_pool_attr {
  THREAD_POOL_HANDLE_T  *handle;
  THREAD_POOL_PARAM_T   *(*block_func)(THREAD_POOL_PARAM_T *ctp);
  void                  (*unblock_func)(THREAD_POOL_PARAM_T *ctp);
  int                   (*handler_func)(THREAD_POOL_PARAM_T *ctp);
  THREAD_POOL_PARAM_T   *(*context_alloc)(
                            THREAD_POOL_HANDLE_T *handle);
  void                  (*context_free)(THREAD_POOL_PARAM_T *ctp);
  pthread_attr_t        *attr;
  unsigned short        lo_water;
  unsigned short        increment;
  unsigned short        hi_water;
  unsigned short        maximum;
  unsigned              reserved[8];
} thread_pool_attr_t;
```

The functions that you fill into the above structure can be taken from the dispatch layer (*dispatch_block()*, ...), the resmgr layer (*resmgr_block()*, ...) or they can be of your own making. If you're not using the resmgr layer functions, then you'll have to define THREAD_POOL_PARAM_T to some sort of context structure for the library to pass between the various functions. By default, it's defined as a `resmgr_context_t` but since this sample is using the dispatch layer, we needed it to be a `dispatch_context_t`. We defined it prior to doing the includes above since the header files refer to it.

In order to correctly define the default `THREAD_POOL_PARAM_T`, `#include` **<sys/resmgr.h>** before **<sys/dispatch.h>**.

Part of the above structure contains information telling the resource manager library how you want it to handle multiple threads (if at all). During development, you should design your resource manager with multiple threads in mind. But during testing, you'll most likely have only one thread running (to simplify debugging). Later, after you've ensured that the base functionality of your resource manager is stable, you may wish to "turn on" multiple threads and revisit the debug cycle.

The following members control the number of threads that are running:

lo_water

Minimum number of blocked threads.

increment

Number of thread to create at a time to achieve *lo_water*.

hi_water

Maximum number of blocked threads.

maximum

Total number of threads created at any time.

The important parameters specify the maximum thread count and the increment. The value for *maximum* should ensure that there's always a thread in a RECEIVE-blocked state. If you're at the number of maximum threads, then your clients will block until a free thread is ready to receive data. The value you specify for *increment* will cut down on the number of times your driver needs to create threads. It's probably wise to err on the side of creating more threads and leaving them around rather than have them being created/destroyed all the time.

You determine the number of threads you want to be RECEIVE-blocked on the *MsgReceive()* at any time by filling in the *lo_water* parameter.

If you ever have fewer than *lo_water* threads RECEIVE-blocked, the *increment* parameter specifies how many threads should be created at once, so that at least *lo_water* number of threads are once again RECEIVE-blocked.

Once the threads are done their processing, they will return to the block function. The *hi_water* variable specifies an upper limit to the number of threads that are RECEIVE-blocked. Once this limit is reached, the threads will destroy themselves to ensure that no more than *hi_water* number of threads are RECEIVE-blocked.

To prevent the number of threads from increasing without bounds, the *maximum* parameter limits the absolute maximum number of threads that will ever run simultaneously.

When threads are created by the resource manager library, they'll have a stack size as specified by the *thread_stack_size* parameter. If you want to specify stack size or priority, fill in *pool_attr.attr* with a proper *pthread_attr_t* pointer.

The `thread_pool_attr_t` structure contains pointers to several functions:

block_func()

Called by the worker thread when it needs to block waiting for some message.

handler_func()

Called by the thread when it has unblocked because it received a message. This function processes the message.

context_alloc()

Called when a new thread is created. Returns a context that this thread uses to do its work.

context_free()

Free the context when the worker thread exits.

unblock_func()

Called by the library to shutdown the thread pool or change the number of running threads.

Thread pool functions

The library provides the following thread pool functions:

thread_pool_create()

Initializes the pool context. Returns a thread pool handle (*tpp*) that's used to start the thread pool.

thread_pool_start()

Start the thread pool. This function may or may not return, depending on the flags passed to *thread_pool_create()*.

thread_pool_destroy()

Destroy a thread pool.

thread_pool_control()

Control the number of threads.

In the example provided in the *multithreaded resource managers* section, `thread_pool_start(tpp)` never returns because we set the POOL_FLAG_EXIT_SELF bit. Also, the POOL_FLAG_USE_SELF flag itself never returns, but the current thread becomes part of the thread pool.

If no flags are passed (i.e., `0` instead of any flags), the function returns after the thread pool is created.

Chapter 11
Filesystem Resource Managers

Considerations for filesystem resource managers

Since a filesystem resource manager may potentially receive long pathnames, it must be able to parse and handle each component of the path properly.

Let's say that a resource manager registers the mountpoint **/mount/**, and a user types:

```
ls -l /mount/home
```

where **/mount/home** is a directory on the device. The `ls` utility does the following:

```
d = opendir("/mount/home");
while (...) {
    dirent = readdir(d);
    ...
}
```

Taking over more than one device

If we wanted our resource manager to handle multiple devices, the change is really quite simple. We would call *resmgr_attach()* for each device name we wanted to register. We would also pass in an attributes structure that was unique to each registered device, so that functions such as *chmod()* would be able to modify the attributes associated with the correct resource.

Here are the modifications necessary to handle both **/dev/sample1** and **/dev/sample2**:

```
/*
 *  MOD [1]:  allocate multiple attribute structures,
 *            and fill in a names array (convenience)
 */

#define NumDevices  2
iofunc_attr_t     sample_attrs [NumDevices];
char              *names [NumDevices] =
{
    "/dev/sample1",
    "/dev/sample2"
};

int main ( void )
{
    ...
    /*
     *  MOD [2]:  fill in the attribute structure for each device
     *           and call resmgr_attach for each device
     */
    for (i = 0; i < NumDevices; i++) {
        iofunc_attr_init (&sample_attrs [i],
                          S_IFCHR | 0666, NULL, NULL);
        pathID = resmgr_attach (dpp, &resmgr_attr, name[i],
                                _FTYPE_ANY, 0,
                                &my_connect_funcs,
                                &my_io_funcs,
                                &sample_attrs [i]);
    }
    ...
}
```

The first modification simply declares an array of attributes, so that each device has its own attributes structure. As a convenience, we've also declared an array of names to simplify passing the name of the device in the *for* loop. Some resource managers (such as `devc-ser8250`) construct the device names on the fly or fetch them from the command line.

The second modification initializes the array of attribute structures and then calls *resmgr_attach()* multiple times, once for each device, passing in a unique name and a unique attribute structure.

Those are all the changes required. Nothing in our *io_read* or *io_write* handlers has to change—the iofunc-layer default functions will gracefully handle the multiple devices.

Handling directories

Up until this point, our discussion has focused on resource managers that associate each device name via discrete calls to *resmgr_attach()*. We've shown how to "take over" a single pathname. (Our examples have used pathnames under **/dev**, but there's no reason you couldn't take over any other pathnames, e.g., **/MyDevice**.)

A typical resource manager can take over any number of pathnames. A practical limit, however, is on the order of a hundred—the real limit is a function of memory size and lookup speed in the process manager.

What if you wanted to take over thousands or even millions of pathnames?

The most straightforward method of doing this is to take over a *pathname prefix* and manage a directory structure below that prefix (or *mountpoint*).

Running more than one pass-through filesystem or resource manager on overlapping pathname spaces might cause deadlocks.

Here are some examples of resource managers that may wish to do this:

- A CD-ROM filesystem might take over the pathname prefix **/cdrom**, and then handle any requests for files below that pathname by going out to the CD-ROM device.
- A filesystem for managing compressed files might take over a pathname prefix of **/uncompressed**, and then uncompress disk files on the fly as read requests arrive.
- A network filesystem could present the directory structure of a remote machine called "flipper" under the pathname prefix of **/mount/flipper** and allow the user to access flipper's files as if they were local to the current machine.

And those are just the most obvious ones. The reasons (and possibilities) are almost endless.

The common characteristic of these resource managers is that they all implement *filesystems*. A filesystem resource manager differs from the "device" resource managers (that we have shown so far) in the following key areas:

1. The _RESMGR_FLAG_DIR bit in the *flags* argument to *resmgr_attach()* informs the library that the resource manager will *accept matches at or below the defined mountpoint*.
2. The _IO_CONNECT logic has to check the individual pathname components against permissions and access authorizations. It must also ensure that the proper attribute is bound when a particular filename is accessed.
3. The _IO_READ logic has to return the data for either the "file" or "directory" specified by the pathname.

Let's look at these points in turn.

Matching at or below a mountpoint

When we specified the *flags* argument to *resmgr_attach()* for our sample resource manager, we specified a 0, implying that the library should "use the defaults."

If we specified the value _RESMGR_FLAG_DIR instead of 0, the library would allow the resolution of pathnames at or below the specified mountpoint.

The _IO_OPEN message for filesystems

Once we've specified a mountpoint, it would then be up to the resource manager to determine a suitable response to an open request. Let's assume that we've defined a mountpoint of **/sample_fsys** for our resource manager:

```
pathID = resmgr_attach
             (dpp,
             &resmgr_attr,
             "/sample_fsys",    /* mountpoint */
            _FTYPE_ANY,
             _RESMGR_FLAG_DIR,   /* it's a directory */
             &connect_funcs,
             &io_funcs,
             &attr);
```

Now when the client performs a call like this:

```
fopen ("/sample_fsys/spud", "r");
```

we receive an _IO_CONNECT message, and our *io_open* handler will be called. Since we haven't yet looked at the _IO_CONNECT message in depth, let's take a look now:

```
struct _io_connect {
    unsigned short  type;
    unsigned short  subtype;     /* _IO_CONNECT_*              */
    unsigned long   file_type;   /* _FTYPE_* in sys/ftype.h    */
    unsigned short  reply_max;
    unsigned short  entry_max;
    unsigned long   key;
    unsigned long   handle;
    unsigned long   ioflag;      /* O_* in fcntl.h, _IO_FLAG_* */
    unsigned long   mode;        /* S_IF* in sys/stat.h        */
    unsigned short  sflag;       /* SH_* in share.h            */
    unsigned short  access;      /* S_I in sys/stat.h          */
    unsigned short  zero;
    unsigned short  path_len;
    unsigned char   eflag;       /* _IO_CONNECT_EFLAG_*        */
    unsigned char   extra_type;  /* _IO_EXTRA_*                */
    unsigned short  extra_len;
    unsigned char   path[1];     /* path_len, null, extra_len  */
};
```

Looking at the relevant fields, we see *ioflag*, *mode*, *sflag*, and *access*, which tell us how the resource was opened.

The *path_len* parameter tells us how many bytes the pathname takes; the actual pathname appears in the *path* parameter. Note that the pathname that appears is *not* **/sample_fsys/spud**, as you might expect, but instead is just **spud**—the message contains only the pathname relative to the resource manager's mountpoint. This simplifies coding because you don't have to skip past the mountpoint

name each time, the code doesn't have to know what the mountpoint is, and the messages will be a little bit shorter.

Note also that the pathname will *never* have relative (**.** and **..**) path components, nor redundant slashes (e.g., **spud//stuff**) in it—these are all resolved and removed by the time the message is sent to the resource manager.

When writing filesystem resource managers, we encounter additional complexity when dealing with the pathnames. For verification of access, we need to break apart the passed pathname and check each component. You can use *strtok()* and friends to break apart the string, and then there's *iofunc_check_access()*, a convenient iofunc-layer call that performs the access verification of pathname components leading up to the target. (See the QNX Neutrino *C Library Reference* page for the *iofunc_open()* for information detailing the steps needed for this level of checking.)

The binding that takes place after the name is validated requires that every path that's handled has its own attribute structure passed to *iofunc_open_default()*. Unexpected behavior will result if the wrong attribute is bound to the pathname that's provided.

Returning directory entries from _IO_READ

When the _IO_READ handler is called, it may need to return data for either a file (if `S_ISDIR (ocb->attr->mode)` is false) or a directory (if `S_ISDIR (ocb->attr->mode)` is true). The *readdir()* function sets _IO_XTYPE_READDIR in the *xtype* member of the _IO_READ message. We've seen the algorithm for returning data, especially the method for matching the returned data's size to the smaller of the data available or the client's buffer size.

A similar constraint is in effect for returning directory data to a client, except we have the added issue of returning *block-integral* data. What this means is that instead of returning a stream of bytes, where we can arbitrarily package the data, we're actually returning a number of `struct dirent` structures. (In other words, we can't return 1.5 of those structures; we always have to return an integral number.) The `dirent` structures must be aligned on 4-byte boundaries in the reply.

A `struct dirent` looks like this:

```
struct dirent {
#if _FILE_OFFSET_BITS - 0 == 64
    ino_t           d_ino;          /* File serial number. */
    off_t           d_offset;
#elif !defined(_FILE_OFFSET_BITS) || _FILE_OFFSET_BITS == 32
#if defined(__LITTLEENDIAN__)
    ino_t           d_ino;          /* File serial number. */
    ino_t           d_ino_hi;
    off_t           d_offset;
    off_t           d_offset_hi;
#elif defined(__BIGENDIAN__)
    ino_t           d_ino_hi;
    ino_t           d_ino;          /* File serial number. */
    off_t           d_offset_hi;
    off_t           d_offset;
#else
 #error endian not configured for system
```

```
#endif
#else
 #error _FILE_OFFSET_BITS value is unsupported
#endif
    int16_t             d_reclen;
    int16_t             d_namelen;
    char                d_name[1];
};
```

The *d_ino* member contains a mountpoint-unique file serial number. This serial number is often used in various disk-checking utilities for such operations as determining infinite-loop directory links. (Note that the inode value cannot be zero, which would indicate that the inode represents an *unused* entry.)

In some filesystems, the *d_offset* member is used to identify the directory entry itself; in others, it's the offset of the *next* directory entry. For a disk-based filesystem, this value might be the actual offset into the on-disk directory structure.

The *d_reclen* member contains the size of this directory entry *and any other associated information* (such as an optional `struct stat` structure appended to the `struct dirent` entry; see below).

The *d_namelen* parameter indicates the size of the *d_name* parameter, which holds the actual name of that directory entry. (Since the size is calculated using *strlen()*, the `\0` string terminator, which must be present, is *not* counted.)

The `dirent` structure includes space only for the first four bytes of the name; your _IO_READ handler needs to return the name and the `struct dirent` as a bigger structure:

```
struct {
    struct dirent ent;
    char namebuf[NAME_MAX + 1 + offsetof(struct dirent, d_name) -
                 sizeof( struct dirent)];
} entry
```

or as a union:

```
union {
    struct dirent ent;
    char filler[ offsetof( struct dirent, dname ) + NAME_MAX + 1];
} entry;
```

So in our *io_read* handler, we need to generate a number of `struct dirent` entries and return them to the client. If we have a cache of directory entries that we maintain in our resource manager, it's a simple matter to construct a set of IOVs to point to those entries. If we don't have a cache, then we must manually assemble the directory entries into a buffer and then return an IOV that points to that.

Returning information associated with a directory structure

Instead of returning just the `struct dirent` in the _IO_READ message, you can also return a `struct stat`. Although this will improve efficiency, returning the `struct stat` is entirely optional. If you don't return one, the users of your device will then have to call *stat()* or *lstat()* to get that information. (This is basically a usage question. If your device is typically used in such a way that

readdir() is called, and then *stat()* is called, it will be more efficient to return both. See the documentation for *readdir()* in the QNX Neutrino *C Library Reference* for more information.)

The client can set the *xtype* member of the message to _IO_XFLAG_DIR_EXTRA_HINT to send a hint to the filesystem to return the extra information, however the filesystem isn't guaranteed to do so. If the resource manager provides the information, it must put it in a `struct dirent_extra_stat`, which is defined as follows:

```
struct dirent_extra_stat {
    uint16_t            d_datalen;
    uint16_t            d_type;
    uint32_t            d_reserved;
    struct stat         d_stat;
};
```

The resource manager must set *d_type* to the appropriate _DTYPE_LSTAT* or _DTYPE_STAT* value, depending on whether or not it resolves symbolic links, whether the information is for a 32- or 64-bit architecture, and which version of POSIX is being used (see *readdir()*). For example:

```
if(msg->i.xtype & _IO_XFLAG_DIR_EXTRA_HINT) {
    struct dirent_extra_stat    extra;
    extra.d_datalen = sizeof extra.d_stat;
    extra.d_type = _DTYPE_LSTAT;
    extra.d_reserved = 0;
    iofunc_stat(ctp, &attr, &extra.d_stat);
    ...
}
```

There's a `dirent_extra_stat` after each directory entry:

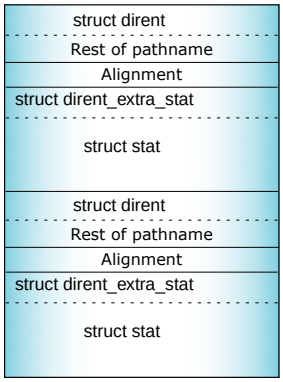

Figure 7: Returning the optional `struct dirent_extra_stat` along with the `struct dirent` entry can improve efficiency.

The `dirent` structures must be aligned on 4-byte boundaries, and the `dirent_extra_stat` structures on 8-byte boundaries. The *d_reclen* member of the `struct dirent` must contain the size of *both* structures, including any space necessary for the pathname and alignment. There must be no more than seven bytes of alignment filler.

The client has to check for extra data by using the *_DEXTRA_*()* macros (see the entry for *readdir()* in the QNX Neutrino *C Library Reference*.) If this check fails, the client will need to call *lstat()* or *stat()* explicitly. For example, `ls -l` checks for extra _DTYPE_LSTAT information; if it isn't present, `ls`

calls *lstat()*. The `ls -L` command checks for extra _DTYPE_STAT information; if it isn't present, `ls` calls *stat()*.

Appendix A
Glossary

administrator

See *superuser*.

alias

A shell feature that lets you create new commands or specify your favorite options. For example, `alias my_ls='ls -F'` creates an alias called `my_ls` that the shell replaces with `ls -F`.

atomic

Of or relating to atoms. :-)

In operating systems, this refers to the requirement that an operation, or sequence of operations, be considered *indivisible*. For example, a thread may need to move a file position to a given location and read data. These operations must be performed in an atomic manner; otherwise, another thread could preempt the original thread and move the file position to a different location, thus causing the original thread to read data from the second thread's position.

BIOS/ROM Monitor extension signature

A certain sequence of bytes indicating to the BIOS or ROM Monitor that the device is to be considered an "extension" to the BIOS or ROM Monitor; control is to be transferred to the device by the BIOS or ROM Monitor, with the expectation that the device will perform additional initializations.

On the x86 architecture, the two bytes `0x55` and `0xAA` must be present (in that order) as the first two bytes in the device, with control being transferred to offset `0x0003`.

budget

In *sporadic* scheduling, the amount of time a thread is permitted to execute at its normal priority before being dropped to its low priority.

buildfile

A text file containing instructions for `mkifs` specifying the contents and other details of an *image*, or for `mkefs` specifying the contents and other details of an embedded filesystem image.

canonical mode

Also called edited mode or "cooked" mode. In this mode the character device library performs line-editing operations on each received character. Only when a line is "completely entered"—typically when a carriage return (CR) is received—will the line of data be made available to application processes. Contrast *raw mode*.

channel

A kernel object used with message passing.

In QNX Neutrino, message passing is directed towards a *connection* (made to a channel); threads can receive messages from channels. A thread that wishes to receive messages creates a channel (using *ChannelCreate()*), and then receives messages from that channel (using *MsgReceive()*). Another thread that wishes to send a message to the first thread must make a connection to that channel by "attaching" to the channel (using *ConnectAttach()*) and then sending data (using *MsgSend()*).

CIFS

Common Internet File System (aka SMB) — a protocol that allows a client workstation to perform transparent file access over a network to a Windows server. Client file access calls are converted to CIFS protocol requests and are sent to the server over the network. The server receives the request, performs the actual filesystem operation, and sends a response back to the client.

CIS

Card Information Structure.

command completion

A shell feature that saves typing; type enough of the command's name to identify it uniquely, and then press **Esc** twice. If possible, the shell fills in the rest of the name.

command interpreter

A process that parses what you type on the command line; also known as a *shell*.

compound command

A command that includes a shell's reserved words, grouping constructs, and function definitions (e.g., `ls -al | less`). Contrast *simple command*.

configurable limit

A special variable that stores system information. Some (e.g., _PC_NAME_MAX) depend on the filesystem and are associated with a path; others (e.g., _SC_ARG_MAX) are independent of paths.

configuration string

A system variable that's like an environment variable, but is more dynamic. When you set an environment variable, the new value affects only the current instance of the shell and any of its children that you create after setting the variable; when you set a configuration string, its new value is immediately available to the entire system.

connection

A kernel object used with message passing.

Connections are created by client threads to "connect" to the channels made available by servers. Once connections are established, clients can *MsgSend()* messages over them.

console

The display adapter, the screen, and the system keyboard are collectively referred to as the *physical console*. A *virtual console* emulates a physical console and lets you run more than one terminal session at a time on a machine.

cooked mode

See *canonical mode*.

core dump

A file describing the state of a process that terminated abnormally.

critical section

A code passage that *must* be executed "serially" (i.e., by only one thread at a time). The simplest from of critical section enforcement is via a *mutex*.

device driver

A process that allows the OS and application programs to make use of the underlying hardware in a generic way (e.g., a disk drive, a network interface). Unlike OSs that require device drivers to be tightly bound into the OS itself, device drivers for QNX Neutrino are standard processes that can be started and stopped dynamically. As a result, adding device drivers doesn't affect any other part of the OS—drivers can be developed and debugged like any other application. Also, device drivers are in their own protected address space, so a bug in a device driver won't cause the entire OS to shut down.

DNS

Domain Name Service — an Internet protocol used to convert ASCII domain names into IP addresses. In QNX Neutrino native networking, `dns` is one of *Qnet's* builtin resolvers.

edge-sensitive

One of two ways in which a *PIC* (Programmable Interrupt Controller) can be programmed to respond to interrupts. In edge-sensitive mode, the interrupt is "noticed" upon a transition to/from the rising/falling edge of a pulse. Contrast *level-sensitive*.

edited mode

See *canonical mode*.

EPROM

Erasable Programmable Read-Only Memory — a memory technology that allows the device to be programmed (typically with higher-than-operating voltages, e.g., 12 V), with the characteristic that any bit (or bits) may be individually programmed from a 1 state to a 0 state. To change a bit from a 0 state into a 1 state can only be accomplished by erasing the *entire* device, setting *all* of the bits to a 1 state. Erasing is accomplished by shining an ultraviolet light through the erase window of the device for a fixed period of time (typically 10–20 minutes). The device is further characterized by having a limited number of erase cycles (typically 10e5 - 10e6). Contrast *EEPROM*, *flash*, and *RAM*.

EEPROM

Electrically Erasable Programmable Read-Only Memory — a memory technology that's similar to *EPROM*, but you can erase the entire device electrically instead of via ultraviolet light. Contrast *flash* and *RAM*.

event

A notification scheme used to inform a thread that a particular condition has occurred. Events can be signals or pulses in the general case; they can also be unblocking events or interrupt events in the case of kernel timeouts and interrupt service routines. An event is delivered by a thread, a timer, the kernel, or an interrupt service routine when appropriate to the requester of the event.

extent

A contiguous sequence of blocks on a disk.

fd

File Descriptor — a client must open a file descriptor to a resource manager via the *open()* function call. The file descriptor then serves as a handle for the client to use in subsequent messages.

FIFO

First In First Out — a scheduling policy whereby a thread is able to consume CPU at its priority level without bounds. See also *round robin* and *sporadic*.

filename completion

A shell feature that saves typing; type enough of the file's name to identify it uniquely, and then press **Esc** twice. If possible, the shell fills in the rest of the name.

filter

A program that reads from standard input and writes to standard output, such as `grep` and `sort`. You can use a pipe (`|`) to combine filters.

flash memory

A memory technology similar in characteristics to *EEPROM* memory, with the exception that erasing is performed electrically, and, depending upon the organization of the flash memory device, erasing may be accomplished in blocks (typically 64 KB bytes at a time) instead of the entire device. Contrast *EPROM* and *RAM*.

FQNN

Fully Qualified Node Name — a unique name that identifies a QNX Neutrino node on a network. The FQNN consists of the nodename plus the node domain tacked together.

garbage collection

The process whereby a filesystem manager recovers the space occupied by deleted files and directories. Also known as space reclamation.

group

A collection of users who share similar file permissions.

HA

High Availability — in telecommunications and other industries, HA describes a system's ability to remain up and running without interruption for extended periods of time.

hard link

See *link*.

hidden file

A file whose name starts with a dot (`.`), such as **.profile**. Commands such as `ls` don't operate on hidden files unless you explicitly say to.

image

In the context of embedded QNX Neutrino systems, an "image" can mean either a structure that contains files (i.e., an OS image) or a structure that can be used in a read-only, read/write, or read/write/reclaim filesystem (i.e., a flash filesystem image).

inode

Information node — a storage table that holds information about files, other than the files' names. In order to support links for each file, the filename is separated from the other information that describes a file.

interrupt

An event (usually caused by hardware) that interrupts whatever the processor was doing and asks it do something else. The hardware will generate an interrupt whenever it has reached some state where software intervention is required.

interrupt latency

The amount of elapsed time between the generation of a hardware interrupt and the first instruction executed by the relevant interrupt service routine. Also designated as "T_{il}". Contrast *scheduling latency*.

IPC

Interprocess Communication — the ability for two processes (or threads) to communicate. QNX Neutrino offers several forms of IPC, most notably native messaging (synchronous, client/server relationship), POSIX message queues and pipes (asynchronous), as well as signals.

IPL

Initial Program Loader — the software component that either takes control at the processor's reset vector (e.g., location `0xFFFFFFF0` on the x86), or is a BIOS extension. This component is responsible for setting up the machine into a usable state, such that the startup program can then perform further initializations. The IPL is written in assembler and C. See also *BIOS/ROM Monitor extension signature* and *startup code*.

IRQ

Interrupt Request — a hardware request line asserted by a peripheral to indicate that it requires servicing by software. The IRQ is handled by the *PIC*, which then interrupts the processor, usually causing the processor to execute an *Interrupt Service Routine (ISR)*.

ISR

Interrupt Service Routine — a routine responsible for servicing hardware (e.g., reading and/or writing some device ports), for updating some data structures shared between the ISR and

the thread(s) running in the application, and for signalling the thread that some kind of event has occurred.

kernel

See *microkernel*.

level-sensitive

One of two ways in which a *PIC* (Programmable Interrupt Controller) can be programmed to respond to interrupts. If the PIC is operating in level-sensitive mode, the IRQ is considered active whenever the corresponding hardware line is active. Contrast *edge-sensitive*.

link

A filename; a pointer to the file's contents. Contrast *symbolic link*.

message

A parcel of bytes passed from one process to another. The OS attaches no special meaning to the content of a message; the data in a message has meaning for the sender of the message and for its receiver, but for no one else.

Message passing not only allows processes to pass data to each other, but also provides a means of synchronizing the execution of several processes. As they send, receive, and reply to messages, processes undergo various "changes of state" that affect when, and for how long, they may run.

metadata

Data about data; for a filesystem, metadata includes all the overhead and attributes involved in storing the user data itself, such as the name of a file, the physical blocks it uses, modification and access timestamps, and so on.

microkernel

A part of the operating system that provides the minimal services used by a team of optional cooperating processes, which in turn provide the higher-level OS functionality. The microkernel itself lacks filesystems and many other services normally expected of an OS; those services are provided by optional processes.

mountpoint

The location in the pathname space where a resource manager has "registered" itself. For example, a CD-ROM filesystem may register a single mountpoint of **/cdrom**.

mutex

Mutual exclusion lock, a simple synchronization service used to ensure exclusive access to data shared between threads. It is typically acquired (*pthread_mutex_lock()*) and released (*pthread_mutex_unlock()*) around the code that accesses the shared data (usually a *critical section*).

name resolution

In a QNX Neutrino network, the process by which the *Qnet* network manager converts an *FQNN* to a list of destination addresses that the transport layer knows how to get to.

name resolver

Program code that attempts to convert an *FQNN* to a destination address.

NDP

Node Discovery Protocol — proprietary QNX Software Systems protocol for broadcasting name resolution requests on a QNX Neutrino LAN.

network directory

A directory in the pathname space that's implemented by the *Qnet* network manager.

Neutrino

Product name of an RTOS developed by QNX Software Systems.

NFS

Network FileSystem — a TCP/IP application that lets you graft remote filesystems (or portions of them) onto your local namespace. Directories on the remote systems appear as part of your local filesystem and all the utilities you use for listing and managing files (e.g., `ls`, `cp`, `mv`) operate on the remote files exactly as they do on your local files.

Node Discovery Protocol

See *NDP*.

node domain

A character string that the *Qnet* network manager tacks onto the nodename to form an *FQNN*.

nodename

A unique name consisting of a character string that identifies a node on a network.

pathname prefix

See *mountpoint*.

pathname-space mapping

The process whereby the Process Manager maintains an association between resource managers and entries in the pathname space.

persistent

When applied to storage media, the ability for the media to retain information across a power-cycle. For example, a hard disk is a persistent storage media, whereas a ramdisk is not, because the data is lost when power is lost.

PIC

Programmable Interrupt Controller — a hardware component that handles IRQs.

PID

Process ID. Also often *pid* (e.g., as an argument in a function call). See also *process ID*.

POSIX

An IEEE/ISO standard. The term is an acronym (of sorts) for Portable Operating System Interface—the "X" alludes to "UNIX", on which the interface is based.

preemption

The act of suspending the execution of one thread and starting (or resuming) another. The suspended thread is said to have been "preempted" by the new thread. Whenever a lower-priority thread is actively consuming the CPU, and a higher-priority thread becomes READY, the lower-priority thread is immediately preempted by the higher-priority thread.

prefix tree

The internal representation used by the Process Manager to store the pathname table.

priority inheritance

The characteristic of a thread that causes its priority to be raised or lowered to that of the thread that sent it a message. Also used with mutexes. Priority inheritance is a method used to prevent *priority inversion*.

priority inversion

A condition that can occur when a low-priority thread consumes CPU at a higher priority than it should. This can be caused by not supporting priority inheritance, such that when the lower-priority thread sends a message to a higher-priority thread, the higher-priority thread consumes CPU *on behalf of* the lower-priority thread. This is solved by having the higher-priority thread inherit the priority of the thread on whose behalf it's working.

process

A nonschedulable entity, which defines the address space and a few data areas. A process must have at least one *thread* running in it.

process group

A collection of processes that permits the signalling of related processes. Each process in the system is a member of a process group identified by a process group ID. A newly created process joins the process group of its creator.

process group ID

The unique identifier representing a process group during its lifetime. A process group ID is a positive integer. The system may reuse a process group ID after the process group dies.

process group leader

A process whose ID is the same as its process group ID.

process ID (PID)

The unique identifier representing a process. A PID is a positive integer. The system may reuse a process ID after the process dies, provided no existing process group has the same ID. Only the Process Manager can have a process ID of 1.

pty

Pseudo-TTY — a character-based device that has two "ends": a master end and a slave end. Data written to the master end shows up on the slave end, and vice versa. You typically use these devices when a program requires a terminal for standard input and output, and one doesn't exist, for example as with sockets.

Qnet

The native network manager in QNX Neutrino.

QNX

Name of an earlier-generation RTOS and the current OS created by QNX Software Systems. Also, short form of the company's name.

QoS

Quality of Service — a policy (e.g., `loadbalance`) used to connect nodes in a network in order to ensure highly dependable transmission. QoS is an issue that often arises in high-availability (*HA*) networks as well as realtime control systems.

QSS

QNX Software Systems.

quoting

A method of forcing a shell's special characters to be treated as simple characters instead of being interpreted in a special way by the shell. For example, `less "my file name"` escapes the special meaning of the spaces in a filename.

RAM

Random Access Memory — a memory technology characterized by the ability to read and write any location in the device without limitation. Contrast *flash*, *EPROM*, and *EEPROM*.

raw mode

In raw input mode, the character device library performs no editing on received characters. This reduces the processing done on each character to a minimum and provides the highest performance interface for reading data. Also, raw mode is used with devices that typically generate binary data—you don't want any translations of the raw binary stream between the device and the application. Contrast *canonical mode*.

remote execution

Running commands on a machine other than your own over a network.

replenishment

In *sporadic* scheduling, the period of time during which a thread is allowed to consume its execution *budget*.

reset vector

The address at which the processor begins executing instructions after the processor's reset line has been activated. On the x86, for example, this is the address `0xFFFFFFF0`.

resource manager

A user-level server program that accepts messages from other programs and, optionally, communicates with hardware. QNX Neutrino resource managers are responsible for presenting an interface to various types of devices, whether actual (e.g., serial ports, parallel ports, network cards, disk drives) or virtual (e.g., **/dev/null**, a network filesystem, and pseudo-ttys).

In other operating systems, this functionality is traditionally associated with *device drivers*. But unlike device drivers, QNX Neutrino resource managers don't require any special arrangements with the kernel. In fact, a resource manager looks just like any other user-level program. See also *device driver*.

root

The superuser, which can do anything on your system. The superuser has what Windows calls administrator's rights.

round robin

A scheduling policy whereby a thread is given a certain period of time (the *timeslice*) to run. Should the thread consume CPU for the entire period of its timeslice, the thread will be placed at the end of the ready queue for its priority, and the next available thread will be made READY. If a thread is the only thread READY at its priority level, it will be able to consume CPU again immediately. See also *FIFO* and *sporadic*.

RTOS

Realtime operating system.

runtime loading

The process whereby a program decides *while it's actually running* that it wishes to load a particular function from a library. Contrast *static linking*.

scheduling latency

The amount of time that elapses between the point when one thread makes another thread READY and when the other thread actually gets some CPU time. Note that this latency is almost always at the control of the system designer.

Also designated as "T_{sl}". Contrast *interrupt latency*.

session

A collection of process groups established for job-control purposes. Each process group is a member of a session. A process belongs to the session that its process group belongs to. A newly created process joins the session of its creator. A process can alter its session membership via *setsid()*. A session can contain multiple process groups.

session leader

A process whose death causes all processes within its process group to receive a SIGHUP signal.

shell

A process that parses what you type on the command line; also known as a *command interpreter*.

shell script

A file that contains shell commands.

simple command

A command line that contains a single command, usually a program that you want to run (e.g., `less my_file`). Contrast *compound command*.

socket

A logical drive in a flash filesystem, consisting of a contiguous and homogeneous region of flash memory.

socket

In TCP/IP, a combination of an IP address and a port number that uniquely identifies a single network process.

software interrupt

Similar to a hardware interrupt (see *interrupt*), except that the source of the interrupt is software.

sporadic

A scheduling policy whereby a thread's priority can oscillate dynamically between a "foreground" or normal priority and a "background" or low priority. A thread is given an execution *budget* of time to be consumed within a certain *replenishment* period. See also *FIFO* and *round robin*.

startup code

The software component that gains control after the IPL code has performed the minimum necessary amount of initialization. After gathering information about the system, the startup code transfers control to the OS.

static linking

The process whereby you combine your programs with the modules from the library to form a single executable that's entirely self-contained. The word "static" implies that it's not going to change—*all* the required modules are already combined into one. Contrast runtime loading.

superuser

The **root** user, which can do anything on your system. The superuser has what Windows calls administrator's rights.

symbolic link

A special file that usually has a pathname as its data. Symbolic links are a flexible means of pathname indirection and are often used to provide multiple paths to a single file. Unlike hard links, symbolic links can cross filesystems and can also create links to directories.

system page area

An area in the kernel that is filled by the startup code and contains information about the system (number of bytes of memory, location of serial ports, etc.) This is also called the SYSPAGE area.

thread

The schedulable entity under QNX Neutrino. A thread is a flow of execution; it exists within the context of a *process*.

timer

A kernel object used in conjunction with time-based functions. A timer is created via *timer_create()* and armed via *timer_settime()*. A timer can then deliver an *event*, either periodically or on a one-shot basis.

timeslice

A period of time assigned to a *round-robin* scheduled thread. This period of time is small (four times the clock period in QNX Neutrino); programs shouldn't rely on the actual value (doing so is considered bad design).

Index

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X