The QNX® Neutrino® Cookbook

Recipes for Programmers

©2003–2014, QNX Software Systems Limited, a subsidiary of BlackBerry. All rights reserved.

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/

QNX, QNX CAR, Neutrino, Momentics, Aviage, and Foundry27 are trademarks of BlackBerry Limited that are registered and/or used in certain jurisdictions, and used under license by QNX Software Systems Limited. All other trademarks belong to their respective owners.

Electronic edition published: Thursday, February 20, 2014

Table of Contents

About This Guide

The QNX Neutrino Cookbook: Recipes for Programmers provides “recipes” that will help you understand how to design and write programs that run on the QNX Neutrino RTOS. There's a separate archive of the source code for the programs that the book describes.

This book was originally written by Rob Krten in 2003. In 2011, QNX Software Systems bought the rights to the book; this edition has been updated by the staff at QNX Software Systems.

The following table may help you find information quickly:

To find out about:	Go to:
Brian Stecher's foreword	*Foreword to the First Edition* (p. 11)
Rob Krten's preface	*Preface to the First Edition* (p. 13)
Code that's common to all the recipes	*The Basics* (p. 23)
How to make your system highly available	*High Availability* (p. 29)
Designing a system that's based on message passing	*Design Philosophy* (p. 51)
Using a resource manager to implement a web counter	*Web Counter Resource Manager* (p. 67)
Writing a data-acquisition server	*ADIOS — Analog/Digital I/O Server* (p. 99)
Writing a basic filesystem	*RAM-disk Filesystem* (p. 135)
Writing a filesystem that manages `.tar` files	*TAR Filesystem* (p. 177)
Additional information about the basics of filesystems	*Filesystems*
Useful information that QNX Neutrino stores in `/proc`	*The `/proc` Filesystem*
Getting the source code discussed in this book	*Sample Programs*
Terms used in QNX docs	*Glossary*

Copyright © 2014, QNX Software Systems Limited

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`
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.

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

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 use a forward slash (/) as a delimiter in all 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.

Copyright © 2014, QNX Software Systems Limited

Foreword to the First Edition by Brian Stecher

For those of you who haven't yet read Rob's earlier book, *Getting Started with QNX Neutrino — A Guide for Realtime Programmers* [now called *Get Programming with the QNX Neutrino RTOS*], let me reiterate some of the comments made by my foreword predecessor, Peter van der Veen. Not only does Rob explain the *how* and *why* of the QNX philosophy, he also makes excellent points that all programmers should take note of, no matter what they're doing.

This book differs from the previous one by being — for the lack of a better word — more "concrete." Not to say that there weren't plenty of examples in Rob's previous tome, but in this one they're presented as complete, fully functional programs, a number of which are currently whirring away at actual customer sites. Most unusual for examples in this kind of book, you're not just presented with the finished product — you also get to see the steps that were taken to get there in the design process.

If you want, you can treat QNX Neutrino as JAPOS (Just Another POSIX Operating System) and write your programs that way, but that misses the point (and fun!) of QNX. Rob's code shows you the other way — the QNX way. QNX extends the UNIX philosophy of "each program should do only one thing, and do it well" from user programs to the operating system itself and the examples in these pages take that to heart.

What programmer in the deep recesses of his or her heart hasn't wanted to write their own OS once or twice? QNX lets you have that fun without the pain, and Rob shows you how here with the code for two complete file systems and overviews of several more — some of which would send programmers working with other operating systems howling to a madhouse! `:-)`

The QNX OS source code has gone through a lot of changes since Dan Dodge and Gord Bell started working on it over twenty years ago — with several complete re-implementations — but the QNX design philosophy has never wavered. Start turning pages and let Rob take you on a tour through our house. It's a comfy place and there's room for everybody.

Brian Stecher, Architect, QNX Software Systems, September 2003

Preface to the First Edition by Rob Krten

I've been using QNX operating systems since 1982 when I was in my late teens, and started developing commercial applications under QNX a few years later. During those years, I've seen QNX's user base grow from just dozens of individuals to the large following it has today.

In the early days, people used QNX because they had a keen appreciation for the simplicity and elegance that the QNX operating system offered. It was, I believe, this simplicity and elegance that allowed those individuals to excel in their work. In those days you'd hear stories about how a team of programmers working with a competing OS were unable to complete projects after months of struggling, where a much smaller team of programmers who knew QNX finished the project in a much shorter time. And the end result was smaller and faster.

For clarity, there are several QNX operating system versions. The original operating system, known simply as "QUNIX" was followed by "QUNIX 2". Both of these ran on the early x86 processors. The next version was QNX 4 (note the name change from "QUNIX"). Finally, the latest version is known as the QNX Neutrino RTOS or (incorrectly) as QNX 6. In this book, we refer to this latest version, and call it simply "QNX Neutrino".

Over the years, as I've worked with QNX, I've tried to adhere to the "simpler is better" philosophy. This is summed up most elegantly by J. Gall in his 1986 book, *Systemantics*:

> A complex system that works is invariably found to have evolved from a simple system that worked.... A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over, beginning with a working simple system.

This is the same idea behind object-oriented design, and the fundamental "design philosophy" of UNIX — lots of small tools with well-defined tasks, joined together to make a complex system.

Confirming the points made in all kinds of software design books, it's been my experience that trying to design a complex working system from scratch just doesn't work. As Gall says, you have to start with a simple system. And, as Frederick P. Brooks, Jr. says in the original version of *The Mythical Man-Month*, you should build a prototype, and be prepared to throw it away. Unfortunately, even some of today's management doesn't quite understand this basic philosophy. I personally think that *The Mythical Man-Month* should be *required* reading — by managers and software developers alike.

In this book I want to share my experience with the QNX Neutrino operating system, and show you lots of simple systems that work. These examples were developed over

Copyright © 2014, QNX Software Systems Limited

the years; some of them are even in commercial use today. The companies that I wrote them for on contract have graciously let me publish the source code (please see the “Thanks!” section, below). A few of the utilities were developed especially for this book — they were projects that I ported from QNX 4 to QNX Neutrino, or were written from scratch to illustrate interesting features of the OS.

What you'll find here are, for all intents and purposes, recipes — small, well-defined pieces of code, or utilities that do something on their own, and can be used as part of a larger system. I will explain step-by-step what the code does, why it was written the way it was (for example, what were the constraints? what was I thinking? `:-)`), how the code can be expanded, and so on.

Copyright © 2014, QNX Software Systems Limited

What's in this book?

This book is divided into these sections:

- Philosophy
- Recipes
- References

Philosophy

In this first section of the book, I discuss the various "big picture" and architectural issues that you need to keep in mind. This is a condensation (some would say "brain dump") of the work I've done with QNX operating systems over the years — I've seen what works, and what doesn't. I'll share both, because I think it's important to know what doesn't work too, so you can understand *why* it didn't work out.

The Basics

In this chapter, the *main()* function and its friend, the option processing function *optproc()*, are discussed. It's in its own chapter so we don't have to talk about the things I use in almost every project.

High Availability

High availability is a very interesting topic. In this chapter, I discuss the concept of high availability (what it is, how it's measured, and how to achieve it). We'll discuss such things as Mean Time Between Failures (MTBF), Mean Time To Repair (MTTR), and the formula used to calculate availability.

I'll also talk about how you can design your systems to be "highly-available" and some of the problems that you'll run into. Unfortunately, in a lot of today's designs, high availability is done as an afterthought — this almost always leads to disaster.

By thinking about high availability up front, you'll be able to benefit from having the architectural insight necessary to design highly-available systems.

Design Philosophy

Next, I present an article about the basic philosophy that's useful when building a system based on message passing — the fundamental Inter Process Communications (IPC) model used by all QNX operating systems. We'll take a hypothetical security system example, and examine how the design is derived to meet various goals. We'll look at the design of the individual pieces (things like swipe card readers and door lock actuators) and see how they fit into a larger system. We'll also discuss things like scalability — how to make your software design scale from a small security system to a large, multi-site security system.

Copyright © 2014, QNX Software Systems Limited

Recipes

The second section of the book contains the "meat" — a smattering of small, useful, and well-documented utilities. Each has been tested on a live QNX Neutrino system. Where there are deficiencies, they're noted — software is an ever-evolving creation. I've also tried to pick utilities that demonstrate something interesting that's "special" about QNX Neutrino, and perhaps not well-understood.

You won't find a graphics ray-trace utility here, because that's not QNX Neutrino-specific (even though it may be poorly understood by the general population of programmers). What you will find are examples of resource managers, high-availability, IPC, and other topics.

Each of the headings below is one chapter in the "recipes" section.

Web-Counter Resource Manager

This chapter describes a utility that illustrates how to generate graphical images on-the-fly within a resource manager. You've seen web counters on various web pages; the count goes up every time someone accesses the web page. In this chapter, I'll show you how this can be done with a neat twist — the web counter looks and acts just like a plain, ordinary file. The "magic" is all done via a resource manager. You'll see how to maintain context on a per-open and per-device basis, how to handle the file content generation, and so on. The chapter presents the project in three phases — a kind of "building-block" approach.

ADIOS — Analog / Digital I/O Server

This project is a data acquisition server, written for Century Aluminum in Kentucky. There are two major parts to this project: card drivers (for the PCL-711, ISO-813, and DIO-144 analog/digital I/O cards), and a master server that collects data from the cards and puts it into shared memory.

Several other utilities are discussed as well, such as `showsamp`, which gets the data from shared memory. This chapter is a good insight on how to handle I/O, as well as shared-memory management.

RAM-disk Filesystem Manager

Many people want to write filesystems, or things that look like filesystems, for QNX Neutrino. The easiest filesystem to understand is a RAM-disk, because we don't need to deal with the "on-media" format — all of our data is stored in RAM, and the data itself is simply allocated from the pool of available memory. Reading, writing, seeking, block management, pathname parsing, directory management, etc. are discussed. This is an extensive chapter that serves as a foundation for the `tar` Filesystem Manager

Copyright © 2014, QNX Software Systems Limited

chapter (immediately following) and also serves as a good basis for any projects you may wish to pursue that need a filesystem (or a filesystem-like) interface.

The `tar` Filesystem Manager

This chapter presents another filesystem, one that manages `.tar` files. It builds on the ideas and content of the RAM-disk chapter (above) and shows how to manage an indexed file — a virtual file that is hosted as a portion of a real, disk-based file.

Additional topics at the end give you some ideas of other types of filesystems that can be constructed.

Copyright © 2014, QNX Software Systems Limited

References

Finally, the last section of the book contains appendixes with useful reference material, as well as some additional general topics:

Filesystems

This appendix provides additional information about the basics of filesystems, not only how files, directories, and symlinks are organized, but also how they map to the resource manager OCB and attributes structures. Fundamentally, all filesystems are a mapping between some physical (or abstract) hierarchical data representation onto the native resource manager structures.

The `/proc` Filesystem

The `/proc` filesystem is where QNX Neutrino stores all of the information about processes and threads — how much CPU time they've used, how much memory they have allocated, how many threads are running, what state they are in, etc. This appendix serves as a handy reference for the `/proc` filesystem, and shows you what information is available, where it is, and how to get it.

Sample Programs

This appendix tells you where to get an archive of the programs discussed in this book, and describes the contents and structure of the archive.

Glossary

Finally, you'll find a glossary that defines the terms used in this book.

What's not in this book?

The general rule when writing is, “Write what you know about.” As a result of this rule, you *won't* find anything in this book about TCP/IP, the Integrated Development Environment (I use `make` and `vi`), the Graphical User Interface (GUI), and so on.

Copyright © 2014, QNX Software Systems Limited

Other references

As I've programmed over the years, I've found the following books and references to be quite useful and enlightening:

***Get Programming with the QNX Neutrino RTOS* (formerly *Getting Started with QNX Neutrino*) by Robert Krten**

This is a pre-requisite for the book you are reading now — it covers the fundamental concepts of QNX Neutrino, such as message passing, and gives you the foundation for understanding things like Resource Managers.

***The Mythical Man-Month* by Frederick P. Brooks, Jr.**

(Addison-Wesley, 1995, ISBN 0-201-83595-9) The intriguing thing about this book is that while it is *ancient* (as far as "computer science" wants us to think), I'd say about 95% of it still applies (in a frighteningly accurate way) to software development today. The 5% that doesn't apply has to do with things like scheduling batch system usage and some antique computer system related issues. An excellent read, and should be read by both management and developers alike.

***Compilers — Principles, Techniques, and Tools* by Alfred V. Aho, Revi Sethi, and Jeffrey D. Ullman.**

(Addison-Wesley, 1986, ISBN 0-201-10088-6) This book is the "de facto standard" from which I learned how to write parsers.

Thanks!

I'd like to extend a gracious "Thank You" to many people who helped out with this book.

First, the QNX Software Systems folks. John Garvey's support when I was writing the RAM-disk section was invaluable. A lot of the behavior of the functions is obscure, and John patiently answered all my questions in the conferences, especially about symlinks. Brian Stecher reviewed copies of the book, provided the foreword, and supplied many details of the `/proc` filesystem that I wouldn't have been able to fathom on my own. Peter van der Veen also reviewed the book, and supplied insightful comments on the `/proc` filesystem and resource managers. Dan Dodge, David Gibbs, Adam Mallory, Peter Martin, Kirk Russell, and Xiaodan Tang reviewed the book and pointed out key omissions and clarified many of the issues involved. Kirk also helped out with the "horror" of `mkisofs :-)`

Other reviewers included David Alessio, Kirk A. Bailey, Bill Caroselli, Mario Charest, Rick Duff, and Alexander Koppel (who took me to task on several topics, thank you!). Thanks for taking the time to read through the book and for your comments; this book is better because of your efforts!

Once again, (for the third time now), Chris Herborth was tricked into editing this book. Thanks for putting up with my writing style and my demands on your time, Chris!

Finally, John Ostrander reviewed the final cut of the book, and once again provided his outstanding and patient copy-editing finesse.

My wife put up with my crawling off to the basement (again) but at least this time she was busy with school `:-)` Thanks Christine!

Century Aluminum

I worked for Century Aluminum on contract during the spring/summer of 2003. They were converting their aluminum smelter lines to use QNX Neutrino, and needed a few device drivers written to manage data acquisition. Out of this was born the ADIOS chapter — my sincere thanks to the people involved with the project for letting me use the source code I developed.

Copyright © 2014, QNX Software Systems Limited

Chapter 1
The Basics

In this chapter, just like with any good cookbook, we'll look at the basic techniques that are used in the recipes. I use a certain programming style, with conventions that I've found comfortable, and more importantly, *useful* and time-saving over the years. I want to share these conventions with you here so I don't have to describe them in each chapter.

If you're an experienced UNIX and C programmer, you can just skip this chapter, or come back to it if you find something confusing later.

In the beginning...

In the beginning, I'd begin each project by creating a new directory, typing `e main.c`, and entering the code (`e` is a custom version of `vi` that I use). Then it hit me that a lot of the stuff that occurs at the beginning of a new project is always the same. You need:

- a `Makefile` to hold the project together
- a `main.c` that contains *main()*, and the command-line processing,
- a usage message (explained below), and
- some way of tracking versions.

For projects dealing with a resource manager (introduced in the previous book), other common parts are required:

- `main.c` should set up the resource manager attributes structure(s), register the mount point, determine if it's going to be single-threaded or multi-threaded, and contain the main processing loop, and
- various I/O and connect functions.

This resulted in two simple scripts (and associated data files) that create the project for me: `mkmain` and `mkresmgr` (see *Threaded Resource Managers* (p. 27), below). You invoke them in an empty directory, and they un-`tar` template files that you can edit to add your own specific functionality.

You'll see this template approach throughout the examples in this book.

The *main()* function

The first thing that runs is, of course, *main()*. You'll notice that my *main()* functions tend to be very short — handle command-line processing (always with *optproc()*), call a few initialization functions (if required), and then start processing.

In general, I try to deallocate any memory that I've allocated throughout the course of the program. This might seem like overkill because the operating system cleans up your resources when you drop off the end of *main()* (or call *exit()*). Why bother to clean up after yourself? Some of the subsystems might find their way into other projects — and they might need to start up and shut down many times during the life of the process. Not being sloppy about the cleanup phase makes my life just that much easier when I reuse code. It also helps when using tools like the debug *malloc()* library.

Copyright © 2014, QNX Software Systems Limited

Command-line processing — *optproc()*

The command-line processing function is a bit more interesting. The majority of variables derived from the command line are called *opt**, where *** is the option letter. You'll often see code like this:

```
if (optv) {
    // do something when -v is present
}
```

By convention, the -v option controls verbosity; each -v increments *optv*. Some code looks at the value of *optv* to determine how much “stuff” to print; other code just treats *optv* as a Boolean. The -d option is often used to control debug output.

Command-line processing generally follows the POSIX convention: some options are just flags (like -v) and some have values. Flags are usually declared as `int`. Valued options are handled in the command-line handler's `switch` statement, including range checking.

One of the last things in most *optproc()* handlers is a final sanity check on the command-line options:

- do we have all of the required options?
- are the values valid?
- are there any conflicting options?

The last thing in *optproc()* is the command-line argument handling. POSIX says that all command-line options come first, followed by the argument list. An initial pass of command-line validation is done right in the `switch` statement after the *getopt()* call. Final validation is done after all of the parameters have been scanned from the command-line.

Common globals

There are a few common global variables that I use in most of my projects:

version

This pointer to a character string contains the version string. By convention, the version is always five characters — `A.BCDE`, with `A` being the major version number, and `BCDE` being the build number. The version is the only thing that's stored in the `version.c` file.

progname

A pointer to a character string containing the name of the program. (This is an old convention of mine; QNX Neutrino now has the *__progname* variable as well.)

blankname

A pointer to a character string the same length as *progname*, filled with blank characters (it gets used in multi-line messages).

I strive to have all messages printed from every utility include the *progname* variable. This is useful if you're running a bunch of utilities and redirecting their output to a common log file or the console.

You'll often see code like this in my programs:

```
fprintf (stderr, "%s:  error message...\n", progname, ...);
```

Usage messages

Both QNX 4 and QNX Neutrino let an executable have a built-in usage message — just a short reminder of the command-line options and some notes on what the executable does.

You can try this out right now at a command-line — type `use cat` to get information about the `cat` command. You'll see the following:

```
cat - concatenate and print files (POSIX)

cat [-cu] [file...]
Options:
 -c      Compress, do not display empty lines.
 -u      Unbuffered, for interactive use.
 -n      Print line numbers without restarting.
 -r      Print line numbers restarting count for each file.
```

You can add these messages into your own executables. My standard `mkmain` script generates the `Makefile` and `main.use` files required for this.

With QNX 4 it was possible to have the usage message embedded within a C source file, typically `main.c`.

This doesn't work the same way under QNX Neutrino. For example, if the usage message has a line containing an odd number of single-quote characters (`'`), the compiler gives you grief, even though the offending line is walled-off in an `#ifdef` section. The solution was to move the usage message out of `main.c` and into a separate file (usually `main.use`). To maintain the usage message within the C code, you can get around this by putting the usage message in comments first, and *then* using the `#ifdef` for the usage message:

```
/*
#ifdef __USAGE
%C A utility that's using a single quote
#endif
*/
```

It's an ANSI C compiler thing. `:-)`

Threaded resource managers

The resource managers presented in this book are single-threaded. The resource manager *part* of the code runs with one thread — other threads, not directly related to the resource manager framework, may also be present.

This was done for simplicity, and because the examples presented here don't need multiple resource manager threads because:

1. the QNX Neutrino resource manager library enforces singled-threaded access to any resource that shares an attributes structure,
2. all of the I/O and Connect function outcalls are run to completion and do not block, and
3. simplicity.

Single-threaded access to resources that share an attributes structure means that if two or more client threads attempt to access a resource, only one of the threads can make an outcall at a time. There's a lock in the attributes structure that the QNX Neutrino library acquires before making an outcall, and releases when the call returns.

The outcalls “running to completion” means that every outcall is done “as fast as possible,” without any delay or blocking calls. Even if the resource manager was multi-threaded, and different attributes structures were being accessed, things would not be any faster on a single-processor box (though they might be faster on an SMP box).

A multi-threaded resource manager may be a *lot* more complex than a single-threaded resource manager. It's worthwhile to consider if you *really* need a multi-threaded resource manager. The complexity comes from:

1. handling unblock pulses (see *Get Programming with the QNX Neutrino RTOS*),
2. terminating the resource manager, and
3. general multi-threaded complexity.

Handling unblock pulses was covered extensively in my previous book — the short story is that the thread that's actively working on the client's request probably won't be the thread that receives the unblock pulse, and the two threads may need to interact in order to abort the in-progress operation. Not impossible to handle, just complicated.

When it comes time to terminate the resource manager, all of the threads need to be synchronized, no new requests are allowed to come in, and there needs to be a graceful shutdown. In a high-availability environment, this can be further complicated by the need to maintain state with a hot standby (see the *High Availability* (p. 29) chapter).

Finally, multi-threaded designs are generally more complex than single-threaded designs. Don't get me wrong, I love to use threads *when appropriate*. For performing multiple, concurrent, and independent operations, threads are wonderful. For speeding up performance on an SMP box, threads are great. But having threads for the sake of having threads, and the accompanying synchronization headaches, should be avoided.

Copyright © 2014, QNX Software Systems Limited

Chapter 2
High Availability

In this chapter, we'll take a look at the concept of high availability (HA). We'll discuss the definition of *availability*, examine the terms and concepts, and take a look at how we can make our software more highly available.

All software has bugs, and bugs manifest themselves in a variety of ways. For example, a module could run out of memory and not handle it properly, or leak memory, or get hit with a `SIGSEGV`, and so on. This leads to two questions:

- How do you recover from those bugs?
- How do you upgrade the software once you've found and fixed bugs?

Obviously, it's not a satisfactory solution to simply say to the customer, “What? Your system crashed? Oh, no problem, just reboot your computer!”

For the second point, it's also not a reasonable thing to suggest to the customer that they shut everything down, and simply “upgrade” everything to the latest version, and then restart it.

Some customers simply cannot afford the downtime presented by either of those “solutions.”

Let's define some terms, and then we'll talk about how we can address these (very important) concerns.

Copyright © 2014, QNX Software Systems Limited

Terminology

You can measure the amount of time that a system is up and running, before it fails. You can also measure the amount of time that it takes you to repair a failed system.

The first number is called *MTBF*, and stands for Mean Time Between Failures. The second number is called *MTTR*, and stands for Mean Time To Repair.

Let's look at an example. If your system can, on average, run for 1000 hours (roughly 41 days), and then fails, and then if it takes you one hour to recover, then you have a system with an MTBF of 1000 hours, and an MTTR of one hour. These numbers are useful on their own, but they are also used to derive a ratio called the *availability* — what percentage of the time your system is available.

This is calculated by the formula:

$$\text{availability} = \frac{\text{MTBF}}{(\text{MTBF} + \text{MTTR})}$$

Figure 1: Calculating MTBF.

If we do the math, with an MTBF of 1000 hours and an MTTR of one hour, your system will have an availability of:

$$\text{availability} = \frac{1000}{(1000 + 1)}$$

Figure 2: MTBF of 1000 hours, MTTR of one hour.

or 0.999 (which is usually expressed as a percentage, so 99.9%). Since ideally the number of leading nines will be large, availability numbers are generally stated as the number of nines — 99.9% is often called “three nines.”

Is three nines good enough? Can we achieve 100% reliability (also known as “continuous availability”)?

Both answers are “no” — the first one is subjective, and depends on what level of service your customers expect, and the second is based on simple statistics — *all* software (and hardware) has bugs and reliability issues. No matter how much redundancy you put in, there is *always* a non-zero probability that you will have a catastrophic failure.

Lies, damn lies, and statistics

It's an interesting phenomenon to see how the human mind perceives reliability. A survey done in the mid 1970s sheds some light. In this survey, the average person on the street was asked, “Would you be satisfied with your telephone service working 90% of the time?” You'd be surprised how many people looked at the number 90, and thought to themselves, “Wow, that's a pretty high number! So, yes, I'd be happy with that!” But when the question was reversed, “Would you be satisfied if your telephone service didn't work 10% of the time?” they were inclined to change their answer even though it's the exact same question!

To understand three, four, or more nines, you need to put the availability percentage into concrete terms — for example, a “down times per year” scale. With a three nines system, your unavailability is 1 - 0.999, or 0.001%. A year has 24 × 365 = 8760 hours. A three nines system would be unavailable for 0.001 × 8760 = 8.76 hours per year.

Some ISPs boast that their end-user availability is an “astounding” 99% — but that's 87.6 hours per year of downtime, over 14 minutes of downtime *per day*!

This confirms the point that you need to pay careful attention to the numbers; your gut reaction to 99% availability might be that it's pretty good (similar to the telephone example above), but when you do the math, 14 minutes of downtime per day may be unacceptable.

The following table summarizes the downtime for various availability percentages:

Availability %	Downtime per Year
99	3.65 days
99.9	8.76 hours
99.99	52.56 minutes
99.999	5.256 minutes
99.9999	31.5 seconds

This leads to the question of how many nines are required. Looking at the other end of the reliability spectrum, a typical telephone central office is expected to have six nines availability — roughly 20 minutes of downtime every 40 years. Of course, each nine that you add means that the system is ten times more available.

Increasing availability

There are two ways to increase the availability:

- increase the MTBF
- decrease the MTTR

If we took the three nines example above and made the MTTR only six minutes (0.1 hours for easy calculations) our availability would now be:

$$\text{availability} = \frac{1000}{(1000 + 0.1)}$$

Figure 3: MTBF of 1000 hours, MTTR of six minutes.

which is 0.9999, or four nines — ten times better!

Increasing the MTBF

Increasing the MTBF, or the overall reliability of the system, is an expensive operation. That doesn't mean that you shouldn't do it, just that it involves a lot of testing, defensive programming, and hardware considerations.

Effectively, the goal here is to eliminate *all* bugs and failures. Since this is generally unfeasible (i.e. will take a near-infinite amount of time and money) in any reasonably sized system, the best you can do is approach that goal.

When my father worked at Bell-Northern Research (later part of Nortel Networks), he was responsible for coming up with a model to predict bug discovery rates, and a model for estimating the number of bugs remaining. As luck would have it, a high-profile prediction of when the next bug would be discovered turned out to be bang on and astonished everyone, especially management who had claimed “There are no more bugs left in the software!”

Once you've established a predicted bug discovery rate, you can then get a feeling for how much it is going to cost you in terms of test effort to discover some percentage of the bugs. Armed with this knowledge, you can then make an informed decision based on a cost model of when it's feasible to declare the product shippable. Note that this will also be a trade-off between when your initial public offering (IPO) is, the status of your competition, and so on. An important trade-off is the cost to fix the problem once the system is in the field. Using the models, you can trade off between testing cost and repair cost.

Decreasing the MTTR

A much simpler and less expensive alternative, though, is to decrease the MTTR. Recall that the MTTR is in the denominator of the availability formula and is what is really driving the availability away from 100% (i.e. if the MTTR was zero, then the

Copyright © 2014, QNX Software Systems Limited

availability would be MTBF / MTBF, or 100%, regardless of the actual *value* of the MTBF.) So anything you can do to make the system recover faster goes a long way towards increasing your availability number.

Sometimes, speed of recovery is not generally thought about until later. This is usually due to the philosophy of “Who cares how long it takes to boot up? Once it's up and running it'll be fast!” Once again, it's a trade off — sometimes taking a long time to boot up is a factor of doing some work “up front” so that the application or system runs faster — perhaps precalculating tables, doing extensive hardware testing up front, etc.

Another important factor is that decreasing MTTR generally needs to be designed into the system right up front. This statement applies to HA in general — it's a *lot* more work to patch a system that doesn't take HA into account, than it is to design one with HA in mind.

Parallel versus serial

The availability numbers that we discussed are for individual components in a system. For example, you may do extensive testing and analysis of your software, and find that a particular component has a certain availability number. But that's not the whole story — your component is part of a larger system, and will affect the availability of the system as a whole. Consider a system that has several modules. If module A relies on the services of module B, and both modules have a five nines availability (99.999%), what happens when you combine them? What's the availability of the system?

Series calculations

When one module depends on another module, we say that the modules are connected in series — the failure of one module results in a failure of the system:

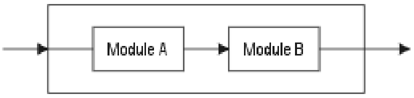


Figure 4: Aggregate module formed by two modules in series.

If module A has an availability of X_a, and module B has an availability of X_b, the combined availability of a subsystem constructed of modules A and B connected in series is:

availability = $X_a \times X_b$

Practically speaking, if both modules have a five nines availability, the system constructed from connecting the two modules in series will be 99.998%:

$$\begin{aligned} \text{availability} &= 0.99999 \times 0.99999 \\ &= 0.99998 \end{aligned}$$

Figure 5: Five nines availability in serial.

You need to be careful here, because the numbers don't *look* too bad, after all, the difference between 0.99999 and 0.99998 is only 0.00001 — hardly worth talking about, right? Well, that's not the case — the system now has *double* the amount of downtime! Let's do the math.

Suppose we wish to see how much downtime we'll have during a year. One year has 365 × 24 × 60 × 60 seconds (31 and a half million seconds). If we have an availability of five nines, it means that we have an unavailability factor of 0.00001 (1 minus 0.99999)

Therefore, taking the 31 and a half million seconds times 0.00001 gives us 315 seconds, or just over five minutes of downtime per year. If we use our new serial availability, 0.99998, and multiply the unavailability (1 minus 0.99998, or 0.00002), we come up with 630 seconds, or 10.5 minutes of downtime — double the amount of downtime!

The reason the math is counter-intuitive is because in order to calculate downtime, we're using the *unavailability* number (that is, one minus the availability number).

Parallel calculations

What if your systems are in parallel? How does that look?

In a parallel system, the picture is as follows:

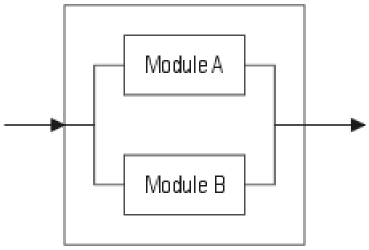


Figure 6: Aggregate module formed by two modules in parallel.

If module A has an availability of X_a, and module B has an availability of X_b, the combined availability of a subsystem constructed of modules A and B connected in parallel is:

availability = $1 - (1 - X_a) \times (1 - X_b)$

Practically speaking, if both modules have a five nines availability, the system constructed from connecting the two modules in parallel will be:

$$\begin{aligned} \text{availability} &= 1 - (1 - 0.99999) \times (1 - 0.99999) \\ &= 1 - 0.00001 \times 0.00001 \\ &= 1 - 0000000001 \\ &= 0.9999999999 \end{aligned}$$

Figure 7: Ten nines!

That number is *ten* nines!

The thing to remember here is that you're not extensively penalized for serial dependencies, but the rewards for parallel dependencies are very worthwhile! Therefore, you'll want to construct your systems to have as much parallel flow as possible and minimize the amount of serial flow.

In terms of software, just what is a parallel flow? A parallel flow is one in which either module A or module B (in our example) can handle the work. This is accomplished by having a redundant server, and the ability to seamlessly use either server — whichever one happens to be available. The reason a parallel flow is more reliable is that a single fault is more likely to occur than a double fault.

A double fault isn't impossible, just *much* less likely. Since the two (or more) modules are operating in parallel, meaning that they are independent of each other, and either will satisfy the request, it would take a double fault to impact *both* modules.

A hardware example of this is powering your machine from two independent power grids. The chance that *both* power grids will fail simultaneously is far less than the chance of either power grid failing. Since we're assuming that the hardware can take power from either grid, and that the power grids are *truly independent of each other*, you can use the availability numbers of both power grids and plug them into the formula above to calculate the likelihood that your system will be without power. (And then there was the North American blackout of August 14, 2003 to reassure everyone of the power grid's stability! `:-)`)

For another example, take a cluster of web servers connected to the same filesystem (running on a RAID box) which can handle requests in parallel. If one of the servers fails, the users will still be able to access the data, even if performance suffers a little. They might not even notice a performance hit.

You can, of course, extend the formula for sub-systems that have more than two components in series or parallel. This is left as an exercise for the reader.

Aggregate calculations

Real, complex systems will have a variety of parallel and serial flows within them. The way to calculate the availability of the entire system is to work with subcomponents. Take the availability numbers of each component, and draw a large diagram with these numbers, making sure that your diagram indicates parallel and serial flows. Then take the serial flows, and collapse them into an aggregate sub-system using the formula. Do the same for parallel flows. Keep doing this until your system has just one flow — that flow will now have the availability number of the entire system.

Copyright © 2014, QNX Software Systems Limited

Failure modes and recovery models

To create a highly available system, we need to consider the system's failure modes and how we'll maximize the MTBF and minimize the MTTR. One thing that won't be immediately obvious in these discussions is the implementation, but I've included an HA example in this book.

Cascade failures

In a typical system, the software fits into several natural layers. The GUI is at the topmost level in the hierarchy, and might interact with a database or a control program layer. These layers then interact with other layers, until finally, the lowest layer controls the hardware.

What happens when a process in the lowest layer fails? When this happens, the next layer often fails as well — it sees that its driver is no longer available and faults. The layer above that notices a similar condition — the resource that it depends on has gone away, so it faults. This can propagate right up to the highest layer, which may report some kind of diagnostic, such as “database not present.” One of the problems is that this diagnostic masks the true cause of the problem — it wasn't *really* a problem with the database, but rather it was a problem with the lowest-level driver.

We call this a *cascade failure* — lower levels causing higher levels to fail, with the failure propagating higher and higher until the highest level fails.

In this case, maximizing the MTBF would mean making not only the lower-level drivers more stable, but also preventing the cascade failure in the first place. This also decreases the MTTR because there are fewer things to repair. When we talk about in-service upgrades, below, we'll see that preventing cascade failures also has some unexpected benefits.

To prevent a cascade failure, you can:

- provide a backup mechanism for failing drivers, so that when a driver fails, it almost immediately cuts over to a standby, and
- provide a fault-tolerance mechanism in each layer that can deal with a momentary outage of a lower-level layer.

What might not be immediately obvious is that these two points are interrelated. It does little good to have a higher-level layer prepared to deal with an outage of a lower-level layer, if the lower-level layer takes a long time to recover. It also doesn't help much if the low-level driver fails and its standby takes over, but the higher-level layer isn't prepared to gracefully handle that momentary outage.

Copyright © 2014, QNX Software Systems Limited

System startup and HA

A point that arises directly out of our cascade failure discussion has to do with system startup. Often, even an HA system is designed such that starting up the system and the normal running operation are two distinct things.

When you stop to think about this, they really don't need to be — what's the difference between a system that's starting up, and a system where every component has crashed? If the system is designed properly, there might not be *any* difference. Each component restarts (and we'll see how that's done below). When it starts up, it treats the lack of a lower-layer component as if the lower-layer component had just failed. Soon, the lower-layer component will start up as well, and operation can resume as if the layer below it suffered a brief outage.

Overlords, or Big Brother is watching you

An important component in an HA system is an *overlord* or *Big Brother* process (as in Orwell, not the TV show). This process is responsible for ensuring that all of the other processes in the system are running. When a process faults, we need to be able to restart it or make a standby process active.

That's the job of the overlord process. It monitors the processes for basic sanity (the definition of which is fairly broad — we'll come back to this), and performs an orderly shutdown, restart, fail-over, or whatever else is required for the failed (or failing) component.

One remaining question is “who watches the watcher?” What happens when the overlord process faults? How do we recover from that? There are a number of steps that you should take with the overlord process regardless of anything I'll tell you later on:

- since it's a *critical* part of the system, it warrants *extensive* testing (this maximizes MTBF).
- in order to minimize the amount of testing required, the overlord should be as simple as possible.

However, since the overlord is a piece of software that's more complex than “Hello, world” it *will* have bugs and it *will* fail.

It would be a chicken-and-egg problem to simply say that we need an overlord to watch the overlord — this would result in a never-ending chain of overlords.

What we really need is a standby overlord that is waiting for the primary overlord to die or become unresponsive, etc. When the primary fails, the standby takes over (possibly killing the faulty primary), becomes primary, and starts up its own standby version. We'll discuss this mechanism next.

Cold, warm, and hot standby

So far, we've said that to make an HA system, we need to have some way of restarting failed components. But we haven't discussed how, or what impact it has.

Recall that when a failure happens, we've just blown the MTBF number — regardless of what the MTBF number is, we now need to focus on minimizing the MTTR. Repairing a component, in this case, simply means replacing the service that the failed component had been providing. There are number of ways of doing this, called *cold standby*, *warm standby*, and *hot standby*.

Mode	In this standby mode:
cold	Repairing the service means noticing that the service has failed and bringing up a new module (i.e., starting an executable by loading it from media), initializing it, and bringing it into service.
warm	Repairing the service is the same as in cold standby mode, except the new the service is already loaded in memory, and may have some idea of the state of the service that just failed.
hot	The standby service is already running. It notices immediately when the primary service fails, and takes over. The primary and the standby service are in constant communication; the standby receives updates from the primary every time a significant event occurs. In hot standby mode, the standby is available almost immediately to take over — the ultimate reduction in MTTR.

Cold, warm, and hot standby are points on a spectrum:

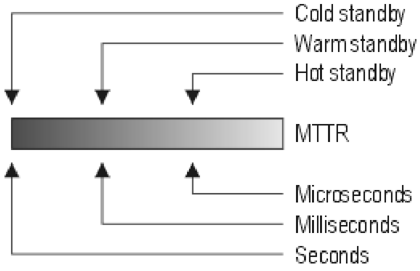


Figure 8: The MTTR spectrum.

Copyright © 2014, QNX Software Systems Limited

The times given above are for discussion purposes only — in your particular system, you may be able to achieve hot standby only after a few hundred microseconds or milliseconds; or you may be able to achieve cold standby after only a few milliseconds.

These broad ranges are based on the following assumptions:

cold standby — seconds

I've selected “seconds” for cold standby because you may need to load a process from some kind of slow media, and the process may need to perform lengthy initializations to get to an operational state. In extreme cases, this scale could go to minutes if you need to power-up equipment.

warm standby — milliseconds

Milliseconds were selected for warm standby because the process is already resident in memory; we're assuming that it just needs to bring itself up to date with the current system status, and then it's operational.

hot standby — microseconds

Ideally, the hot standby scenario can result in an operational process within the time it takes the kernel to make a context switch and for the process to make a few administrative operations. We're assuming that the executable is running on the system, and has a complete picture of the state — it's immediately ready to take over.

Achieving cold standby

For some systems, a cold standby approach may be sufficient. While the cold standby approach does have a higher MTTR than the other two, it is *significantly* easier to implement. All the overlord needs to do is notice that the process has failed, and then start a new version of the process.

Usually this means that the newly started process initializes itself in the same way that it would if it was just starting up for the first time — it may read a configuration file, test its dependent subsystems, bind to whatever services it needs, and then advertise itself to higher level processes as being ready to service their requests.

A cold standby process might be something like a serial port driver. If it faults, the overlord simply starts a new version of the serial port driver. The driver initializes the serial ports, and then advertises itself (for example, by putting `/dev/ser1` and `/dev/ser2` into the pathname space) as being available. Higher-level processes may notice that the serial port seemed to go away for a little while, but that it's back in operation, and the system can proceed.

Copyright © 2014, QNX Software Systems Limited

Achieving warm standby

In warm standby, another instance of the process is already resident in memory, and may be partially initialized. When the primary process fails, the overlord or the standby notices that the primary process has failed, and informs the standby process that it should now assume the duties of the primary process. For this system to work, the newly started process should arrange to create another warm standby copy of itself, in case it meets with an untimely end.

Generally, a warm standby process would be something that might take a long time to initialize (perhaps precalculating some tables), but once called into action can switch over to active mode quickly.

The MTTR of a warm standby process is in between the MTTR of cold standby and hot standby. The implementation of a warm standby process is still relatively straightforward; it works just like a newly started process, except that after it reaches a certain point in its processing, it lies dormant, waiting for the primary process to fail. Then it wakes up, performs whatever further initialization it needs to, and runs.

The reason a warm standby process may need to perform further initialization only after it's been activated is that it may depend on being able to determine the current state of the system before it can service requests, but such determination cannot be made *a priori*; it can only be made when the standby is about to service requests.

Achieving hot standby

With hot standby, we see a process that minimizes the MTTR, but is also (in the majority of cases) a *lot* more complicated than either the cold or warm standby.

The reason for the additional complexity is due to a number of factors. The standby process may need to *actively*:

- monitor the health of its primary process — this could be anything from establishing a connection to the primary process and blocking until it dies, or it may involve more complex heuristics.
- receive current transaction-by-transaction updates, so that it *always* stays in sync with the primary. This is the basis for the hot standby process's ability to “instantly” take over the functionality — it's already up to date and running.

Of course, as with the warm standby process, the hot standby process needs to create another copy of itself to feed updates to when it becomes primary, in case it fails.

An excellent example of a hot standby process is a database. As transactions to the primary version of the database are occurring, these same transactions are fed to the hot standby process, ensuring that it is synchronized with the primary.

Problems

The major problem with any standby software that gets data from an active primary is that, because it's the exact same version of software, any bad data that kills the primary may also kill the secondary, because it will tickle the same software bug.

If you have near-infinite money, the proper way to architect this is to have the primary and the standby developed by two independent teams, so that there will at least be different bugs in the software. This also implies that you have near-infinite money and time to test all possible fail-over scenarios. Of course, there is still a common point of failure, and that's the specification itself that's given to the two independent teams...

Detecting failure

There are a number of ways to detect process failure. The overlord process can do this, or, in the case of hot or warm standby, the standby process can do this.

If you don't have the source code for the process, you must resort to either polling periodically (to see if the process is still alive), or arranging for death notification via an *obituary.*

If you do have the source code for the process, and are willing to modify it, you can arrange for the process to send you obituaries automatically.

Obituaries

Obituaries are quite simple. Recall that a client creates a connection to a server. The client then asks the server to perform various tasks, and the client blocks until the server receives the message and replies. When the server replies, the client unblocks.

One way of receiving an obituary is to have the client send the server a message, stating "please do not reply to this message, ever." While the server is running, the client thread is blocked, and when the server faults, the kernel will automatically unblock the client with an error. When the client thread unblocks, it has implicitly received an obituary message.

A similar mechanism works in the opposite direction to notify the server of the client's death. In this case, the client calls *open()* to open the server, and never closes the file descriptor. If the client dies, the kernel will synthesize a *close()* message, which the server can interpret as an obituary. The kernel's synthetic *close()* looks just like the client's *close()* — except that the client and server have agreed that the *close()* message is an obituary message, and not just a normal *close()*. The client would never issue a normal *close()* call, so if the server gets one, it *must* mean the death of the client.

Putting this into perspective, in the warm and hot standby cases, we can arrange to have the two processes (the primary and the standby) work in a client/server relationship. Since the primary will always create the standby (so the standby can take over in the event of the primary's death), the standby can be a client of, or a server for, the primary. Using the methods outlined above, the standby can receive an instant obituary message when the primary dies.

Should the standby be a client or a server? That depends on the design of your system. In most cases, the primary process will be a server for some other, higher-level processes. This means that the standby process had better be a server as well, because it will need to take over the server functionality of the primary. Since the primary is a server, then we need to look at the warm and hot standby cases separately.

In the warm standby case, we want the secondary to start up, initialize, and then go to sleep, waiting for the primary to fail. The easiest way to arrange this is for the secondary to send a message to the primary telling it to never reply to the message. When the primary dies, the kernel unblocks the secondary, and the secondary can then proceed with becoming a primary.

In the hot standby case, we want the secondary to start up, initialize, and then actively receive updates from the primary, so that it stays synchronized. Either method will work (the secondary can be a client of the primary, as in the warm standby case, or the secondary can be a server for the primary).

Implementing the secondary as a client of the primary is done by having the secondary make requests like “give me the next update,” and block, until the primary hands over the next request. Then, the secondary digests the update, and sends a message asking for the next update.

Implementing the secondary as a server for the primary means that the secondary will be doing almost the exact same work as it would as primary — it will receive requests (in this case, only updates) from the primary, digest them, and then reply with the result. The result could be used by the primary to check the secondary, or it could simply be ignored. The secondary does need to reply in order to unblock the primary.

If the secondary is a client, it won't block the primary, but it does mean that the primary needs to keep track of transactions in a queue somewhere in case the secondary lags behind. If the secondary is a server, it blocks the primary (potentially causing the primary's clients to block as well), but it means that the code path that the secondary uses is the same as that used when it becomes primary.

Whether the secondary is a client or a server is your choice; this is one aspect of HA system design you will need to think about carefully.

Graceful fail-over

To avoid a cascade failure, the clients of a process must be coded so they can tolerate a momentary outage of a lower-level process.

Copyright © 2014, QNX Software Systems Limited

It would almost completely defeat the purpose of having hot standby processes if the processes that used their services couldn't gracefully handle the failure of a lower-level process. We discussed the impacts of cascade failures, but not their solution.

In general, the higher-level processes need to be aware that the lower-level process they rely on may fault. The higher-level processes need to maintain the state of their interactions with the lower-level process — they need to know what they were doing in order to be able to recover.

Let's look at a simple example first. Suppose that a process were using the serial port. It issues commands to the serial port when it starts up:

- set the baud rate to 38400 baud
- set the parity to 8,1,none
- set the port to raw mode

Suppose that the serial port is supervised by the overlord process, and that it follows the cold standby model.

When the serial port driver fails, the overlord restarts it. Unfortunately, the overlord has no idea of what settings the individual ports should have; the serial port driver will set them to whatever defaults it has, which may not match what the higher-level process expects.

The higher-level process may notice that the serial port has disappeared when it gets an error from a *write()*, for example. When that happens, the higher-level process needs to determine what happened and recover. This would case a cascade failure in non-HA software — the higher-level process would get the error from the *write()*, and would call *exit()* because it didn't handle the error in an HA-compatible manner.

Let's assume that our higher-level process is smarter than that. It notices the error, and because this is an HA system, assumes that someone else (the overlord) will notice the error as well and restart the serial port driver. The main trick is that the higher-level process needs to restore its operating context — in our example, it needs to reset the serial port to 38400 baud, eight data bits, one stop bit, and no parity, and it needs to reset the port to operate in raw mode.

Only after it has completed those tasks can the higher-level process continue where it left off in its operation. Even then, it may need to perform some higher-level reinitialization — not only does the serial port need to be set for a certain speed, but the peripheral that the high-level process was communicating with may need to be reset as well (for example, a modem may need to be hung up and the phone number redialed).

This is the concept of *fault tolerance*: being able to handle a fault and to recover gracefully.

If the serial port were implemented using the hot standby model, some of the initialization work may not be required. Since the state carried by the serial port is minimal (i.e. the only state that's generally important is the baud rate and configuration), and the serial port driver is generally very small, a cold standby solution may be sufficient for most applications.

Using shadows

QNX Neutrino's pathname space gives us some interesting choices when the time comes to design our warm or hot standby servers. Recall that the pathname space is maintained by the process manager, which is also the entity responsible for creation and cleanup of processes. One thing that's not immediately apparent is that you can have multiple processes registered for the same pathname, and that you can have a specific order assigned to pathname resolution.

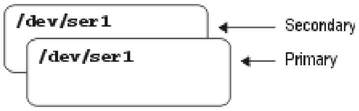


Figure 9: Primary and secondary servers registered for the same pathname.

A common trick for designing warm and hot standby servers is for the secondary to register the same pathname as the primary, but to tell the process manager to register it *behind* the existing pathname. Any requests to the pathname will be satisfied by the primary (because its pathname is "in front" of the secondary's pathname). When the primary fails, the process manager cleans up the process and *also* cleans up the pathname registered by the primary — this uncovers the pathname registered by the secondary.

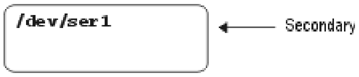


Figure 10: The secondary server is exposed when the primary fails.

When we say that a client reconnects to the server, we mean that literally. The client may notice that its connection to `/dev/ser1` has encountered a problem, and as part of its recovery it tries to open `/dev/ser1` again — it can assume that the standby module's registered pathname will be exposed by the failure of the primary.

In-service upgrades

The most interesting thing that happens when you combine the concepts of fault tolerance and the various standby models is that you get *in-service upgrades* almost for free.

An in-service upgrade means that you need to be able to modify the version of software running in your system without affecting the system's ability to do whatever it's doing.

As an interesting implementation, and for a bit of contrast, some six nines systems, like central office telephone switches, accomplish this in a unique manner. In the switch, there are two processors running the main software. This is for reliability —

Copyright © 2014, QNX Software Systems Limited

the two processors are operating in lock-step synchronization, meaning that they execute the exact same instruction, from the exact same address, at the exact same time. If there is ever any discrepancy between the two CPUs, service is briefly interrupted as both CPUs go into an independent diagnostic mode, and the failing CPU is taken offline (alarm bells ring, logs are generated, the building is evacuated, etc.).

This dual-CPU mechanism is also used for upgrading the software. One CPU is manually placed offline, and the switch runs with only the other CPU (granted, this is a small “asking for trouble” kind of window, but these things are generally done at 3:00 AM on a Sunday morning). The offline CPU is given a new software load, and then the two CPUs switch roles — the currently running CPU goes offline, and the offline CPU with the new software becomes the controlling CPU. If the upgrade passes sanity testing, the offline processor is placed online, and full dual-redundant mode is reestablished. Even scarier things can happen, such as live software patches!

We can do something very similar with software, using the HA concepts that we've discussed so far (and good design — see the *Design Philosophy* (p. 51) chapter).

What's the real difference between killing a driver and restarting it with the same version, versus killing a driver and restarting a newer version (or, in certain special cases, an older version)? If you've made the versions of the driver compatible, there is no difference. That's what I meant when I said that you get in-service upgrades for free! To upgrade a driver, you kill the current version. The higher level software notices the outage, and expects something like the overlord process to come in and fix things. However, the overlord process not only fixes things, but *upgrades the version that it loads*. The higher-level software doesn't really notice; it retries its connection to the driver, and eventually discovers that a driver exists and continues running.

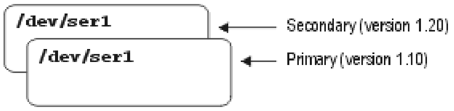


Figure 11: Preparation for in-service upgrade; the secondary server has a higher version number.

Of course, I've deliberately oversimplified things to give you the big picture. A botched in-service upgrade is an *excellent* way to get yourself fired. Here are just some of the kinds of things that can go wrong:

- new version is susceptible to the same problem as the old version,
- new version introduces newer, bigger, and nastier bugs, and
- new version is incompatible with the older version.

These are things that require testing, testing, and more testing.

Policies

Generally speaking, the HA infrastructure presented here is good but there's one more thing that we need to talk about. What if a process dies, and when restarted, dies again, and keeps dying? A good HA system will cover that aspect as well, by providing per-process *policies*. A policy defines things such as:

- how often a process is allowed to fault,
- at what rate it's restarted, and
- what to do when these limits are exceeded.

While it's a good idea to restart a process when it faults, some processes can be very expensive to restart (perhaps in terms of the amount of CPU the process takes to start up, or the extent to which it ties up other resources).

An overlord process needs to be able to limit how fast and how often a process is restarted. One technique is an *exponential back-off* algorithm. When the process dies, it's restarted immediately. If it dies again with a certain time window, it's restarted after 200 milliseconds. If it dies again with a certain time window, it's restarted after 400 milliseconds, then 800 milliseconds, then 1600 milliseconds, and so on, up to a certain limit. If it exceeds the limit, another policy is invoked that determines what to do about this process. One possibility is to run the previous version of the process, in case the new version has some new bug that the older version doesn't. Another might be to raise alarms, or page someone. Other actions are left to your imagination, and depend on the kind of system that you're designing.

Copyright © 2014, QNX Software Systems Limited

Implementing HA

Modifying a system to be HA after the system is designed can be stupidly expensive, while designing an HA system in the first place is merely moderately expensive.

The question you need to answer is, how much availability do you need? To a large extent this is a business decision (i.e., do you have service-level agreements with your customers? Are you going to be sued if your system faults in the field? What's the availability number for your competition's equipment?); often just thinking about HA can lead to a system that's good enough.

RK drones on about his home systems again

On my home system, I had a problem with one of the servers periodically dying. There didn't seem to be any particular situation that manifested the problem. Once every few weeks this server would get hit with a `SIGSEGV` signal. I wasn't in a position to fix it, and didn't really have the time to analyze the problem and submit a proper bug report. What I did have time to do, though, was hack together a tiny shell script that functions as an overlord. The script polls once per second to see if the server is up. If the server dies, the script restarts it. Client programs simply reconnect to the server once it's back up. Dead simple, ten lines of shell script, an hour of programming and testing, and the problem is now solved (although masked might be a better term).

Even though I had a system with a poor MTBF, by fixing the situation in a matter of a second or two (MTTR), I was able to have a system that met my availability requirements.

Of course, in a proper production environment, the core dumps from the server would be analyzed, the fault would be added to the regression test suite, and there'd be no extra stock options for the designer of the server. `:-)`

Other HA systems

I've worked at a few companies that have HA systems.

QNX Software Systems has the HAT (High Availability Toolkit) and the HAM (High Availability Manager). HAT is a toolkit that includes the HAM, various APIs for client recovery, and many source code examples. HAM is the manager component that monitors processes on your system.

QNX Neutrino includes the `/proc` filesystem, which is where you get information about processes so you can write your own policies and monitor things that are of interest in your system.

There are several other HA packages available.

Chapter 3
Design Philosophy

This chapter was first published in QNX Software Systems' email magazine, *QNX Source*, as a two-part series.

Copyright © 2014, QNX Software Systems Limited

Decoupling design in a message-passing environment

When you're first designing a project using QNX Neutrino, a question that often arises is “how should I structure my design?” This chapter answers that question by examining the architecture of a sample application; a security system with door locks, card readers, and related issues. We'll discuss process granularity — what components should be put into what size of containers. That is, what the responsibilities of an individual process should be, and where you draw the line between processes.

QNX Neutrino is advertised as a “message-passing” operating system. Understanding the true meaning of that phrase when it comes to designing your system can be a challenge. Sure, you don't need to understand this for a standard “UNIX-like” application, like a web server, or a logging application. But it becomes an issue when you're designing an entire system. A common problem that arises is design decoupling. This problem often shows up when you ask the following questions:

1. How much work should one process do? Where do I draw the line in terms of functionality?
2. How do I structure the drivers for my hardware?
3. How do I create a large system in a modular manner?
4. Is this design future-proof? Will it work two years from now when my requirements change?

Here are the short answers, in order:

1. A process must focus on the task at hand; leave everything else to other processes.
2. Drivers must be structured to present an abstraction of the hardware.
3. To create a large system, start with several small, well-defined components and glue them together.
4. If you've done these things, you'll have a system made of reusable components that you can rearrange or reuse in the future, and that can accommodate new building blocks.

In this chapter, we're going to look into these issues, using a reasonably simple situation:

Say you're the software architect for a security company, and you're creating the software design for a security system. The hardware consists of swipe-card readers, door lock actuators, and various sensors (smoke, fire, motion, glass-break, etc.). Your company wants to build products for a range of markets — a small security system that's suitable for a home or a small office, up to gigantic systems that are suitable for large, multi-site customers. They want their systems to be upgradable, so that as a customer's site grows, they can just add more and more devices, without having to throw out the small system to buy a whole new medium or large system (go figure!).

Copyright © 2014, QNX Software Systems Limited

Finally, any systems should support any device (the small system might be a super-high security area, and require some high-end input devices).

Your first job is to sketch out a high-level architectural overview of the system, and decide how to structure the software. Working from our goals, the implied requirements are that the system must support various numbers of each device, distributed across a range of physical areas, and it must be future-proof so you can add new types of sensors, output devices, and so on as your requirements change or as new types of hardware become available (e.g. retinal scanners).

The first step is to define the functional breakdown and answer the question, “How much work should one process do?”

If we step back from our security example for a moment, and consider a database program, we'll see some of the same concepts. A database program manages a database — it doesn't worry about the media that the data lives on, nor is it worried about the organization of that media, or the partitions on the hard disk, etc. It certainly does *not* care about the SCSI or EIDE disk driver hardware. The database uses a set of abstract services supplied by the filesystem — as far as the database is concerned, everything else is opaque — it doesn't need to see further down the abstraction chain. The filesystem uses a set of abstract services from the disk driver. Finally, the disk driver controls the hardware. The obvious advantage of this approach is this: *because* the higher levels don't know the details of the lower levels, we can substitute the lower levels with different implementations, if we maintain the well-defined abstract interfaces.

Thinking about our security system, we can immediately start at the bottom (the hardware) — we know we have different types of hardware devices, and we know the next level in the hierarchy probably does *not* want to know the details of the hardware. Our first step is to draw the line at the hardware interfaces. What this means is that we'll create a set of device drivers for the hardware and provide a well-defined API for other software to use.

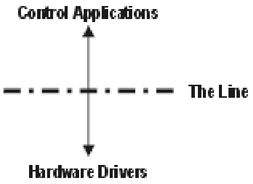


Figure 12: We've drawn the line between the control applications and the hardware drivers.

We're also going to need some kind of control application. For example, it needs to verify that Mr. Pink actually has access to door number 76 at 06:30 in the morning, and if that's the case, allow him to enter that door. We can already see that the control software will need to:

- access a database (for verification and logging)

- read the data from the swipe-card readers (to find out who's at the door)
- control the door locks (to open the door).

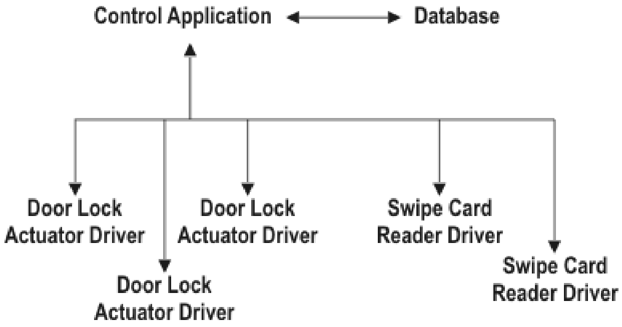

Control Application
Database
Door Lock Actuator Driver
Door Lock Actuator Driver
Door Lock Actuator Driver
Swipe Card Reader Driver
Swipe Card Reader Driver

Figure 13: In the next design phase, we've identified some of the hardware components, and refined the control application layer slightly.

Once we have these two levels defined, we can sort out the interface. Since we've analyzed the requirements at a higher level, we know what “shape” our interface will take on the lower level.

For the swipe-card readers, we're going to need to know:

- if someone has swiped their card at the card reader, and
- which card reader it was, so we can determine the person's location.

For the door-lock actuator hardware, we're going to need to open the door, and lock the door (so that we can let Mr. Pink get in, but lock the door after him).

Earlier, we mentioned that this system should scale — that is, it should operate in a small business environment with just a few devices, right up to a large campus environment with hundreds (if not thousands) of devices.

When you analyze a system for scalability, you're looking at the following:

- Is there enough CPU power to handle a maximum system?
- Is there enough hardware capacity (PCI slots, etc.)?
- How do you distribute your system?
- Are there any bottlenecks? Where? How do we get around them?

As we'll see, these scalability concerns are very closely tied in with the way that we've chosen to break up our design work. If we put too much functionality into a given process (say we had one process that controls *all* the door locks), we're limiting what we can do in terms of distributing that process across multiple CPUs. If we put too little functionality into a process (one process per function per door lock), we run into the problem of excessive communications and overhead.

So, keeping these goals in mind, let's look at the design for each of our hardware drivers.

Copyright © 2014, QNX Software Systems Limited

Door-lock actuators

We'll start with the door-lock actuators, because they're the simplest to deal with. The only thing we want this driver to do is to let a door be opened, or prevent one from being opened — that's it! This dictates the commands that the driver will take — we need to tell the driver which door lock it should manipulate, and its new state (locked or unlocked). For the initial software prototype, those are the only commands that I'd provide. Later, to offload processing and give us the ability to support new hardware, we might consider adding more commands.

Suppose you have different types of door-lock actuators — there are at least two types we should consider. One is a door release mechanism — when active, the door can be opened, and when inactive, the door cannot be opened. The second is a motor-driven door mechanism — when active, the motor starts and causes the door to swing open; when inactive, the motor releases the door allowing it to swing closed. These might look like two different drivers, with two different interfaces. However, they can be handled by the same interface (but *not* the same driver).

All we really want to do is let someone go through the door. Naturally, this means that we'd like to have some kind of timer associated with the opening of the door. When we've granted access, we'll allow the door to remain unlocked for, say, 20 seconds. After that point, we lock the door. For certain kinds of doors, we might wish to change the time, longer or shorter, as required.

A key question that arises is, “Where do we put this timer?” There are several possible places where it can go:

- the door lock driver itself,
- a separate timing manager driver that *then* talks to the door lock driver, or
- the control program itself.

This comes back to design decoupling (and is related to scalability).

If we put the timing functionality into the control program, we're adding to its workload. Not only does the control program have to handle its normal duties (database verification, operator displays, etc.), it now also has to manage timers.

For a small, simple system, this probably won't make much of a difference. But once we start to scale our design into a campus-wide security system, we'll be incurring additional processing in one central location. Whenever you see the phrase “one central location” you should immediately be looking for scalability problems. There are a number of significant problems with putting the functionality into the control program:

scalability

the control program must maintain a timer for each door; the more doors, the more timers.

security

if the communications system fails after the control program has opened the door, you're left with an unlocked door.

upgradability

if a new type of door requires different timing parameters (for example, instead of lock and unlock commands, it might require multiple timing parameters to sequence various hardware), you now have to upgrade the control program.

The short answer here is that the control program really shouldn't have to manage the timers. This is a low-level detail that's ideally suited to being offloaded.

Let's consider the next point. Should we have a process that manages the timers, and *then* communicates with the door-lock actuators? Again, I'd answer no. The scalability and security aspects raised above don't apply in this case (we're assuming that this timer manager could be distributed across various CPUs, so it scales well, and since it's on the same CPU we can eliminate the communications failure component). The upgradability aspect still applies, however.

But there's also a new issue — functional clustering. What I mean by that is that the timing function is tied to the hardware. You might have dumb hardware where you have to do the timing yourself, or you might have smart hardware that has timers built into it.

In addition, you might have complicated hardware that requires multi-phase sequencing. By having a separate manager handle the timing, it has to be aware of the hardware details. The problem here is that you've split the hardware knowledge across two processes, without gaining any advantages. On a filesystem disk driver, for example, this might be similar to having one process being responsible for reading blocks from the disk while another one was responsible for writing. You're certainly not gaining anything, and in fact you're complicating the driver because now the two processes must coordinate with each other to get access to the hardware.

That said, there are cases where having a process that's between the control program and the individual door locks makes sense.

Suppose that we wanted to create a meta door-lock driver for some kind of complicated, sequenced door access (for example, in a high security area, where you want one door to open, and then completely close, before allowing access to another door). In this case, the meta driver would actually be responsible for cycling the individual door lock drivers. The nice thing about the way we've structured our devices is that as far as the control software is concerned, this meta driver looks just like a standard door-lock driver — the details are hidden by the interface.

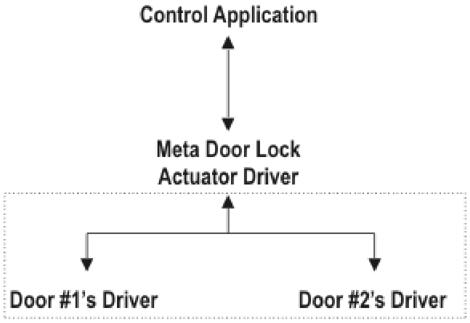


Figure 14: The meta door-lock driver presents the same interface as a regular door-lock driver.

At this point...

Our security system isn't complete! We've just looked at the door locks and illustrated some of the common design decisions that get made. Our goals for the door-lock driver were for the process to do as much work as was necessary to provide a clean, simple, abstract interface for higher levels of software to use. We used functional clustering (guided by the capabilities of the hardware) when deciding which functions were appropriate for the driver, and which functions should be left to other processes. By implication, we saw some of the issues associated with constructing a large system as a set of small, well-defined building blocks, which could all be tested and developed individually. This naturally led us to design a system that will allow new types of modules to be added seamlessly.

Managing message flow

Once you've laid out your initial design, it's important to understand the send hierarchy and its design impact. In this section, we'll examine which processes should be data sources and which should be data sinks, as well as the overall structure of a larger system.

Once the message flow is understood, we'll look at more scalability issues.

Let's focus on the swipe-card readers to illustrate some message flow and scalability concepts.

Swipe-card readers

The swipe-card reader driver is a fairly simple program — it waits for a card to be swiped through the hardware, and then collects the card ID. While the driver itself is reasonably simple, there's a major design point we need to consider that's different than what we've seen so far with the door lock driver. Recall that the door-lock driver acted as a data sink — the control process sent it commands, telling it to perform certain actions. With the swipe-card reader, we don't want it to be a data sink. We don't want to ask the swipe card reader if it has data yet; we want it to *tell us* when someone swipes their access card. We need the swipe-card reader to be a data source — it should provide data to another application when the data is available.

To understand this, let's think about what happens in the control program. The control program needs to know which user has swiped their card. If the control program kept asking the swipe card reader if anything had happened yet, we'd run into at a scalability problem (via polling), and possibly a blocking problem.

The scalability problem stems from the fact that the control program is constantly inquiring about the status of the swipe card reader. This is analogous to being on a road trip with kids who keep asking “Are we there yet?” It would be more efficient (but far more unlikely) for them to ask (once!) “Tell us when we get there.”

If the control program is continually polling, we're wasting CPU cycles — it costs us to ask, and it costs us to get an answer. Not only are we wasting CPU, but we could also be wasting network bandwidth in a network-distributed system. If we only have one or two swipe-card readers, and we poll them once per second, this isn't a big deal. However, once we have a large number of readers, and if our polling happens faster, we'll run into problems.

Ideally, we'll want to have the reader send us any data that it has as soon as it's available. This is also the intuitive solution; you expect something to happen *as soon as* you swipe your card.

Copyright © 2014, QNX Software Systems Limited

The blocking problem could happen if the control program sent a message to the reader driver asking for data. If the reader driver didn't have any data, it could block — that is, it wouldn't reply to the client (who is asking for the data) until data became available. While this might seem like a good strategy, the control program might need to ask *multiple* reader drivers for data. Unless you add threads to the control program (which could become a scalability problem), you'd have a problem — the control program would be stuck waiting for a response from one swipe-card reader. If this request was made on a Friday night, and no one swiped in through that particular card reader until Monday morning, the control program would be sitting there all weekend, waiting for an answer.

Meanwhile, someone could be trying to swipe in on Saturday morning through a different reader. Gratuitously adding threads to the control program (so there's one thread per swipe card reader) isn't necessarily an ideal solution either. It doesn't scale well. While threads are reasonably inexpensive, they still have a definite cost — thread-local memory, some internal kernel data structures, and context-switch overhead. It's the memory footprint that'll have the most impact (the kernel data structures used are relatively small, and the context-switch times are very fast). Once you add these threads to the control program, you'll need to synchronize them so they don't contend for resources (such as the database). The only thing that these threads buy you is the ability to have multiple blocking agents within your control program.

At this point, it seems that the solution is simple. Why not have the swipe-card reader send a message to the control program, indicating that data is available? This way, the control program doesn't have to poll, and it doesn't need a pool of threads. This means that the swipe-card reader is a “data source,” as we suggested above.

There's one subtle but critical point here. If the reader sends a message to the control program, *and* the control program sends a message to the reader at the same time, we'll run into a problem. Both programs sent messages to each other, and are now deadlocked, waiting for the other program to respond to their messages. While this might be something that only happens every so often, it'll fail during a demo rather than in the lab, or worse, it'll fail in the field.

Why would the control program want to send a message to the swipe-card reader in the first place? Well, some readers have an LED that can be used to indicate if access was allowed or not — the control program needs to set this LED to the correct color. (Obviously, in this example, there are many ways of working around this; however, in a general design, you should *never* design a system that even has a hint of allowing a deadlock!)

The easiest way around this deadlock is to ensure that it never happens. This is done with something called a *send hierarchy*. You structure your programs so those at one level always send messages to those at a lower level, but that those at lower levels *never* send messages to those at higher (or the same) levels. In this way, two programs will never send each other messages at the same time.

Unfortunately, this seems to break the simple solution that we had earlier: letting the swipe-card reader driver send a message to the control program. The standard solution to this is to use a non-blocking message (a QNX Neutrino-specific “pulse”). This message can be sent up the hierarchy, and doesn't violate our rules because it doesn't block the sender. Even if a higher level used a regular message to send data to a lower level at the same time the lower level used the pulse to send data to the higher level, the lower level would receive and process the message from the higher level, because the lower level wouldn't be blocked.

Let's put that into the context of our swipe-card reader. The control program would “prime” the reader by sending it a regular message, telling it to send a pulse back up whenever it has data. Then the control program goes about its business. Some time later, when a card is swiped through the reader, the driver sends a pulse to the control program. This tells the control program that it can now safely go and ask the driver for the data, knowing that data is available, and that the control program won't block. Since the driver has indicated that it has data, the control program can safely send a message to the driver, knowing that it'll get a response back almost instantly.

In the following diagram, there are a number of interactions:

1. The control program primes the swipe-card reader.
2. The reader replies with an OK.
3. When the reader has data, it sends a pulse up to the control program.
4. The control program sends a message asking for the data.
5. The reader replies with the data.

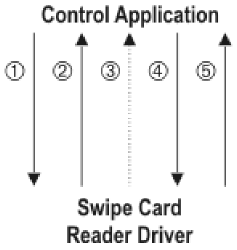


Figure 15: A well-defined message-passing hierarchy prevents deadlocks.

Using this send hierarchy strategy, we have the swipe-card driver at the bottom, with the control program above it. This means that the control program can send to the driver, but the driver needs to use pulses to transfer indications “up” to the control program. We could certainly have this reversed — have the driver at the top, sending data down to the control program. The control program would use pulses to control the LED status on the swipe-card hardware to indicate access granted or denied.

Why would we prefer one organization over the other? What are the impacts and trade-offs?

Let's look at the situation step-by-step to see what happens in both cases.

Copyright © 2014, QNX Software Systems Limited

Control program sends to the swipe-card reader

In the case where the Control Program (Control) is sending data to the Swipe-Card Reader (Swipe), we have the following steps:

1. Control sends to Swipe.
2. Swipe replies with “OK, event primed.”
3. A big delay occurs until the next person swipes their card.
4. Swipe uses a pulse to say “I have data available.”
5. Control sends a message, “What is your data?”
6. Swipe replies with “Swipe card ID # *xxxx* was swiped.”
7. Control sends message, “Change LED to GREEN.”
8. Swipe replies with “OK.”
9. A 20-second delay occurs, while the door is opened.
10. Control sends message, “Change LED to RED.”
11. Swipe replies with “OK.”

In this case, there was one pulse, and three messages. Granted, you would prime the event just once, so the first message can be eliminated.

Swipe-card reader sends to control program

If we invert the relationship, here's what happens:

1. A big delay occurs until the next person swipes their card.
2. Swipe sends message “Swipe card ID # *xxxx* was swiped.”
3. Control replies with “OK, change LED to GREEN.”
4. A 20-second delay occurs, while the door is opened.
5. Control sends a pulse “Change LED to RED.”

In this case, there was one pulse, and just one message.

An important design note is that the LED color change was accomplished with a *reply* in one case (to change it to green), and a pulse in another case (to change it to red). This may be a key point for your design, depending on how much data you need to carry in the pulse. In our example, simply carrying the color of the LED is well within the capabilities of the pulse.

Let's see what happens when we add a keypad to the system. The purpose of the keypad is for Control to issue a challenge to the person trying to open the door; they must respond with some kind of password.

Using a keypad challenge — control program sends to the swipe-card reader

In the case where Control is sending data to Swipe, in conjunction with a keypad challenge, we have the following steps:

1. Control sends to Swipe.
2. Swipe replies with, “OK, event primed.”
3. A big delay occurs until the next person swipes their card.
4. Swipe uses a pulse to say, “I have data available.”
5. Control sends a message, “What is your data?”
6. Swipe replies with, “Swipe card ID # *xxxx* was swiped.”
7. Control sends message, “Challenge the user with *YYYY*.”
8. Swipe replies with, “OK, challenge in progress.”
9. A delay occurs while the user types in the response.
10. Swipe sends a pulse to Control with, “Challenge complete.” (We're assuming that the challenge response doesn't fit in a pulse.)
11. Control sends a message, “What is the challenge response?”
12. Swipe replies with, “Challenge response is *IZZY*.”
13. Control sends a message, “change LED to GREEN.”
14. Swipe replies with, “OK.”
15. A 20-second delay occurs, while the door is opened.
16. Control sends a message, “Change LED to RED.”
17. Swipe replies with, “OK.”

In this case, there were two pulses and six messages. Again, the initial priming message could be discounted.

Using a keypad challenge — swipe-card reader sends to control program

If we invert the relationship, here's what happens:

1. A big delay occurs until the next person swipes their card.
2. Swipe sends message “Swipe card ID # *xxxx* was swiped.”
3. Control replies with “Challenge the user with *YYYY*.”
4. A delay occurs while the user types in the response.
5. Swipe sends message “Challenge response is *IZZY*.”
6. Control replies with “OK, change LED to GREEN.”
7. A 20-second delay occurs, while the door is opened.
8. Control sends a pulse “Change LED to RED.”

In this case, there was one pulse and two messages.

So, by carefully analyzing the transactions between your clients and servers, and by being able to carry useful data within pulses, you can make your transactions much more efficient! This has ramifications over a network connection, where message speeds are much slower than locally.

Copyright © 2014, QNX Software Systems Limited

Scalability

The next issue we need to discuss is scalability. Scalability can be summarized by the question, “Can I grow this system by an order of magnitude (or more) and still have everything work?” To answer this question, you have to analyze a number of factors:

- How much CPU, memory, and other resources are you using?
- How much message passing are you performing?
- Can you distribute your application across multiple CPUs?

The first point is probably self-evident. If you're using half of the CPU time and half of the memory of your machine, then you probably won't be able to more than double the size of the system (assuming that the resource usage is linearly increasing with the size of the system).

Closely related to that is the second point — if you're doing a lot of message passing, you will eventually “max out” the message passing bandwidth of whatever medium you're using. If you're only using 10% of the CPU but 90% of the bandwidth of your medium, you'll hit the medium's bandwidth limit first.

This is tied to the third point, which is the real focus of the scalability discussion here.

If you're using a good chunk of the resources on a particular machine (also called a *node* under QNX Neutrino), the traditional scalability solution is to share the work between multiple nodes. In our security example, let's say we were scaling up to a campus-wide security system. We certainly wouldn't consider having one CPU responsible for hundreds (or thousands) of door lock actuators, swipe card readers, etc. Such a system would probably die a horrible death immediately after a fire drill, when everyone on the campus has to badge-in almost simultaneously when the all-clear signal is given.

What we'd do instead is set up zone controllers. Generally, you'd set these up along natural physical boundaries. For example, in the campus that you're controlling, you might have 15 buildings. I'd immediately start with 15 controller CPUs; one for each building. This way, you've effectively reduced most of the problem into 15 smaller problems — you're no longer dealing with one, large, monolithic security system, but instead, you're dealing with 15 individual (and smaller) security systems.

During your design phase, you'd figure out what the maximum capacity of a single CPU was — how many door-lock actuators and swipe-card readers it could handle in a worst-case scenario (like the fire drill example above). Then you'd deploy CPUs as appropriate to be able to handle the expected load.

While it's good that you now have 15 separate systems, it's also bad — you need to coordinate database updates and system-level monitoring between the individual systems. This is again a scalability issue, but at one level higher. (Many commercial off-the-shelf database packages handle database updates, high availability, fail-overs, redundant systems, etc. for you.)

You could have one CPU (with backup!) dedicated to being the master database for the entire system. The 15 subsystems would all ask the one master database CPU to validate access requests. Now, it may turn out that a single CPU handling the database would scale right up to 200 subsystems, or it might not. If it does, then your work is done — you know that you can handle a fairly large system. If it doesn't, then you need to once again break the problem down into multiple subsystems.

In our security system example, the database that controls access requests is fairly static — we don't change the data on a millisecond-to-millisecond basis. Generally, we'd update the data only when a new employee joins the company, one gets terminated, someone loses their card, or the access permissions for an employee change.

To distribute this database, we can simply have the main database server send out updates to each of its “mirror” servers. The mirror servers are the ones that then handle the day-to-day operations of access validation. This nicely addresses the issue of a centralized outage — if the main database goes down, all of the mirror servers will still have a fairly fresh update. Since you've designed the central database server to be redundant, it'll come back up real soon, and no one will notice the outage.

Distributed processing

For purposes of illustration, let's say that the data *did* in fact change very fast. How would we handle that? The standard solution here is “distributed processing.”

We'd have multiple database servers, but each one would handle only a certain subset of the requests. The easiest way to picture this is to imagine that each of the swipe cards has a 32-bit serial number to identify the swipe card.

If we needed to break the database server problem down into, say, 16 sub-problems, then you could do something very simple. Look at the four least significant bits of the serial number, and then use that as a server address to determine *which* server should handle the request. (This assumes that the 32-bit serial numbers have an even distribution of least-significant four-bit addresses; if not, there are plenty of other hashing algorithms.) Need to break the problem into 32 “sub-problems?” Just look at the five least significant bits of the serial number, and so on.

This kind of approach is done all the time at trade shows — you'll see several registration desks. One might have “Last Name A-H,” “Last Name I-P,” and “Last Name Q-Z” to distribute the hoards of people into three separate streams. Depending on where you live, your local driver's license bureau may do something similar — your driver's license expires on your birthday. While there isn't necessarily a completely even distribution of birthdays over the days of the year, it does distribute the problem somewhat evenly.

Copyright © 2014, QNX Software Systems Limited

Summary

I've shown you some of the basic implications of message passing, notably the send hierarchy and how to design with it in mind to avoid deadlocks. We've also looked at scalability and distributed processing, noting that the idea is to break the problem down into a number of sub-problems that can be distributed among multiple CPUs.

Chapter 4
Web Counter Resource Manager

The web counter resource manager was created specifically for this book, and the code is presented in a three-phase building-block approach. In addition, there are a few interesting diversions that occurred along the way.

Requirements

The requirements for the web counter resource manager are very simple — it dynamically generates a graphic file that contains a bitmap of the numbers representing the current hit count. The hit count is meant to indicate how many times the resource has been accessed. Further phases of the project refine this by adding font selection, directory management, and other features.

Using the web counter resource manager

The web counter is very simple to use. As `root`, you run it in the background:

```
# webcounter &
```

The web counter is now running, and has taken over the pathname `/dev/webcounter.gif`. To see if it's really doing its job, I used my `aview` (Animation Viewer) program. If you test it using a web browser, you'll need to hit the reload or refresh button to get the next count to display.

To use it as part of a web page, you'd include it in an `<img>` tag:

```
<html>
<head>
<title>My Web Page</title>
<body>
<h1>My Web Page</h1>
<p>
My web page has been accessed this many times:
<img src="/dev/webcounter.gif">
</p>
</body>
</html>
```

Of course, your web server must be set up to allow access to `/dev/webcounter.gif` — you can get around this by using a symlink, or by setting the name of the web counter output file on the command line (use -n):

```
# webcounter -n /home/rk/public_html/hits.gif &
```

Copyright © 2014, QNX Software Systems Limited

Design

The webcounter design was very ad-hoc — the requirements were that it display a graphical image to illustrate on-demand data generation in a resource manager. Most of the command-line options were defined right at the time that I edited `main.c` — things like the X and Y image size, the background and foreground colors, and so on. The -n parameter, to determine the name of the resource, was already part of the "standard" resource manager "framework" library that I use as a template. The -s option was added last. It controls the starting count, and after the first draft of the code was completed, I got bored with the fact that it always started at zero, and wanted to provide a way of getting an arbitrary count into the counter.

Generating the graphical image

The hardest part of the design was generating the graphical image. I chose a 7-segment LED style because I figured it would be the easiest to generate. Of course, a plain 7-segment representation, where each segment was a rectangle, proved to be too simple, so I had to make the segments with a nice diagonal corner so they'd fit together nicely. Sometimes, I just have too much time on my hands.

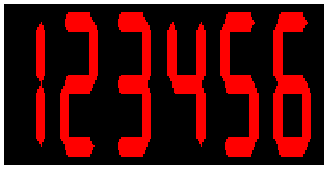


Figure 16: The simulated 7-segment digits.

Copyright © 2014, QNX Software Systems Limited

The code — phase 1

In this section, we'll discuss the code for the phase 1 implementation. In later sections, we'll examine the differences between this code and the phase 2 and phase 3 implementations.

The code consists of the following modules:

`Makefile`

This is a very basic `Makefile`, nothing special here.

`main.use`

Usage message for the web counter resource manager.

`main.c`

Main module. This contains pretty much everything.

`7seg.c`

7-segment LED simulator module.

You'll also need a GIF encoder.

Operation

Operation begins in the usual manner with *main()* in `main.c`. We call our option processor (*optproc()*) and then we enter the resource manager main loop at *execute_resmgr()*. The resource manager main loop never returns.

All of the work is done as callouts by the connect and I/O functions of the resource manager. We take over the *io_open()*, *io_read()*, and *io_close_ocb()* functions, and leave all the rest to the QNX Neutrino libraries. This makes sense when you think about it — apart from the magic of generating a graphical image on-the-fly, all we're really doing is handling the client's *open()* (so we can set up our data areas), giving the data to the client (via the client's *read()*), and cleaning up after ourselves (when we get the client's *close()*).

However, as you'll see shortly, even these simple operations have some interesting consequences that need to be considered.

Generating the graphical image

Part of the magic of this resource manager is generating the graphical image. The process is quite simple. We allocate an array to hold the graphical image as a bitmap. Then, we draw the 7-segment display into the bitmap, and finally we convert the bitmap into a GIF-encoded data block. It's the GIF-encoded data block — not the raw bitmap — that's returned by the *io_read()* handler to the client.

Step-by-step code walkthrough

Let's look at the source for the web counter resource manager, starting with the include files.

Include files and data structures

This project uses the following include files:

`7seg.h`

Nothing interesting here, just a function prototype.

`gif.h`

This contains the GIF compressor's work area data structure, `gif_context_t`. The work area is used for compressing the bitmap to GIF format. Since the topic of GIF compression is beyond the scope of this book I won't go details. All of the global variables that were present in the original GIF encoder were gathered up and placed into a data structure.

Source files

The web counter resource manager uses the following source files:

`main.c`

Standard *main()* and option-processing functions. Since this is such a small example, the three resource manager callouts (*io_open()*, *io_read()*, and *io_close_ocb()*) are present in this file as well.

`version.c`

This just contains the version number.

`7seg.c`

This file contains the *render_7segment()* routine. It draws the graphical representation of the 7-segment LED picture into the bitmap.

Apart from those files, there's also a fairly generic `Makefile` for building the executable.

Copyright © 2014, QNX Software Systems Limited

The code

As usual, execution begins at *main()* in `main.c`. We won't talk about *main()*; it only does very basic command-line processing (via our *optproc()* function), and then calls *execute_resmgr()*.

The *execute_resmgr()* function

One thing that you'll notice right away is that we extended the attributes (`iofunc_attr_t`) and the OCB (`iofunc_ocb_t`) structures:

```
typedef struct my_attr_s
{
    iofunc_attr_t       base;
    int                 count;
}   my_attr_t;

typedef struct my_ocb_s
{
    iofunc_ocb_t        base;
    unsigned char       *output;
    int                 size;
}   my_ocb_t;
```

It's a resource manager convention to place the standard structure as the first member of the extended structure. Thus, both members named *base* are the nonextended versions.

Extending the attributes and OCB structures is discussed in *Get Programming with the QNX Neutrino RTOS* in the Resource Managers chapter.

Recall that an instance of the attributes structure is created for each device. This means that for the device `/dev/webcounter1.gif`, there will be exactly one attributes structure. Our extension simply stores the current count value. We certainly could have placed that into a global variable, but that would be bad design. If it were a global variable, we would be *prevented* from manifesting multiple devices. Granted, the current example shows only one device, but it's good practice to make the architecture as flexible as possible, especially if it doesn't add undue complexity. Adding one field to an extended structure is certainly not a big issue.

Things get more interesting in the OCB extensions. The OCB structure is present on a per-open basis; if four clients have called *open()* and haven't closed their file descriptors yet, there will be four instances of the OCB structure.

This brings us to the first design issue. The initial design had an interesting bug. Originally, I reasoned that since the QNX Neutrino resource manager library allows only single-threaded access to the resource (our `/dev/webcounter.gif`), then it would be safe to place the GIF context (the *output* member) and the size of the resource (the *size* member) into global variables. This singled-threaded behavior is standard for resource managers, because the QNX Neutrino library locks the attributes structure before performing any I/O function callouts. Thus, I felt confident that there would be no problems.

Copyright © 2014, QNX Software Systems Limited

In fact, during initial testing, there were no problems — only because I tested the resource manager with a single client at a time. I felt that this was an insufficient test, and ran it with multiple simultaneous clients:

```
# aview -df1 /dev/webcounter.gif &
# aview -df1 /dev/webcounter.gif &
# aview -df1 /dev/webcounter.gif &
# aview -df1 /dev/webcounter.gif &
```

and that's when the first bug showed up. Obviously, the GIF context members needed to be stored on a per-client basis, because each client would be requesting a different version of the number (one client would be at `712`, the next would be at `713`, and so on). This was readily fixed by extending the OCB and adding the GIF context member *output*. (I really should have done this initially, but for some reason it didn't occur to me at the time.)

At this point, I thought all the problems were fixed; multiple clients would show the numbers happily incrementing, and each client would show a different number, just as you would expect.

Then, the second interesting bug hit. Occasionally, I'd see that some of the `aview` clients would show only part of the image; the bottom part of the image would be cut off. This glitch would clear itself up on the next refresh of the display, so I was initially at a loss to explain where the problem was. I even suspected `aview`, although it had worked flawlessly in the past.

The problem turned out to be subtle. Inside of the base attributes structure is a member called *nbytes*, which indicates the size of the resource. The “size of the resource” is the number of bytes that would be reported by `ls -l` — that is, the size of the “file” `/dev/webcounter1.gif`.

When you do an `ls -l` of the web counter, the size reported by `ls` is fetched from the attributes structure via `ls`'s *stat()* call. You'd expect that the size of a resource wouldn't change, but it does! Since the GIF compression algorithm squeezes out redundancy in the source bitmap image, it will generate different sizes of output for each image that's presented to it. Because of the way that the *io_read()* callout works, it requires the *nbytes* member to be accurate. That's how *io_read()* determines that the client has reached the end of the resource.

The first client would *open()* the resource, and begin reading. This caused the GIF compression algorithm to generate a compressed output stream. Since I needed the size of the resource to match the number of bytes that are returned, I wrote the number of bytes output by the GIF compression algorithm into the attributes structure's *nbytes* member, thinking that was the correct place to put it.

But consider a second client, preempting the first client, and generating a new compressed GIF stream. The size of the *second* stream was placed into the attributes structure's *nbytes* member, resulting in a (potentially) different size for the resource! This means that when the first client resumed reading, it was now comparing the new *nbytes* member against a *different* value than the size of the stream it was processing! So, if the second client generated a *shorter* compressed data stream, the *first* client would be tricked into thinking that the data stream was shorter than it should be. The net result was the second bug: only part of the image would appear — the bottom would be cut off because the first client hit the end-of-file prematurely.

The solution, therefore, was to store the size of the resource on a per-client basis in the OCB, and force the size (the *nbytes* member of the attributes structure) to be the one appropriate to the *current* client. This is a bit of a kludge, in that we have a resource that has a varying size depending on who's looking at it. You certainly wouldn't run into this problem with a traditional file — all clients using the file get whatever happens to be in the file when they do their *read()*. If the file gets shorter during the time that they are doing their *read()*, it's only natural to return the shorter size to the client.

Effectively, what the web counter resource manager does is maintain *virtual* client-sensitive devices (mapped onto the same name), with each client getting a slightly different view of the contents.

Think about it this way. If we had a standalone program that generated GIF images, and we piped the output of that program to a file every time a client came along and opened the image, then multiple concurrent clients would get inconsistent views of the file contents. They'd have to resort to some kind of locking or serialization method. Instead, by storing the actual generated image, and its size, in the per-client data area (the OCB), we've eliminated this problem by taking a snapshot of the data that's relevant to the client, without the possibility of having another client upset this snapshot.

That's why we need the *io_close_ocb()* handler — to release the per-client context blocks that we generated in the *io_open()*.

So what size do we give the resource when no clients are actively changing it? Since there's no simple way of pre-computing the size of the generated GIF image (short of running the compressor, which is a mildly expensive operation), I simply give it the size of the uncompressed bitmap buffer (in the *io_open()*).

Now that I've given you some background into why the code was designed the way it was, let's look at the resource manager portion of the code.

The *io_open()* function

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

    sts = iofunc_open (ctp, msg, &handle -> base, NULL, NULL);
    if (sts != EOK) {
        return (sts);
    }

    ocb = calloc (1, sizeof (*ocb));
    if (ocb == NULL) {
        return (ENOMEM);
    }

    // give them the unencoded size
    handle -> base.nbytes = optx * opty;

    sts = iofunc_ocb_attach (ctp, msg, &ocb -> base,
                             &handle -> base, NULL);
    return (sts);
}
```

The only thing that's unique in this *io_open()* handler is that we allocate the OCB ourselves (we do this because we need a non standard size), and then jam the *nbytes* member with the raw uncompressed size, as discussed above.

In an earlier version of the code, instead of using the raw uncompressed size, I decided to call the 7-segment render function and to GIF-encode the output. I thought this was a good idea, reasoning that every time the client calls *open()* on the resource, I should increment the number. This way, too, I could give a more "realistic" size for the file (turns out that the compressed file is on the order of 5% of the size of the raw image). Unfortunately, that didn't work out because a lot of things end up causing the *io_open()* handler to run — things like `ls` would *stat()* the file, resulting in an inaccurate count. Some utilities prefer to *stat()* the file first, and *then* open it and read the data, causing the numbers to jump unexpectedly. I removed the code that generates the compressed stream from *io_open()* and instead moved it down to the *io_read()*.

The *io_read()* function

```
static int
io_read (resmgr_context_t *ctp, io_read_t *msg, RESMGR_OCB_T *ocb)
{
    int     nbytes;
    int     nleft;
    int     sts;
    char    string [MAX_DIGITS + 1];

    // 1) we don't do any xtypes here...
    if ((msg -> i.xtype & _IO_XTYPE_MASK) != _IO_XTYPE_NONE) {
        return (ENOSYS);
    }

    // standard helper
    sts = iofunc_read_verify (ctp, msg, &ocb -> base, NULL);
    if (sts != EOK) {
        return (sts);
    }

    // 2) generate and compress an image
    if (!ocb -> output) {
        unsigned char *input;   // limited scope

        input = calloc (optx, opty);
        if (input == NULL) {
            return (ENOMEM);
        }
        ocb -> output = calloc (optx, opty);
        if (ocb -> output == NULL) {
            free (input);
            return (ENOMEM);
        }

        sprintf (string, "%0*d", optd, ocb -> base.attr -> count++);
        render_7segment (string, input, optx, opty);
        ocb -> size = encode_image (input, optx, opty, ocb -> output);
        free (input);
    }
    // 3) figure out how many bytes are left
    nleft = ocb -> size - ocb -> base.offset;

    // 4) and how many we can return to the client
    nbytes = min (nleft, msg -> i.nbytes);

    if (nbytes) {
        // 5) return it to the client
        MsgReply (ctp -> rcvid, nbytes,
                 ocb -> output + ocb -> base.offset, nbytes);

        // 6) update flags and offset
        ocb -> base.attr -> base.flags |=
            IOFUNC_ATTR_ATIME | IOFUNC_ATTR_DIRTY_TIME;
        ocb -> base.offset += nbytes;
    } else {
        // 7) nothing to return, indicate End Of File
        MsgReply (ctp -> rcvid, EOK, NULL, 0);
    }

    // 8) already done the reply ourselves
    return (_RESMGR_NOREPLY);
}
```

Let's look at this code step-by-step:

1. If there are any XTYPE directives, we return ENOSYS because we don't handle XTYPEs. XTYPEs are discussed in the *Get Programming with the QNX Neutrino RTOS* book in the “Resource Managers” chapter.
2. If we currently don't have a compressed image to work from (i.e. this is the first time that we've been called for this particular *open()* request), we allocate the temporary input (raw buffer) and OCB's output (compressed buffer) data areas, call *render_7segment()* to draw the picture into the raw buffer, and then *encode_image()* to compress it. Notice that *encode_image()* returns the number of bytes that it generated and we store that into the OCB's *size* member. Then we free the temporary *input* buffer area.
3. We calculate the number of bytes that are left, which is simply the difference between the number of bytes that we have in the compressed image buffer and our current offset within that buffer. Note that we use the OCB's *size* member rather than the attributes structure's *nbytes* member (see note after step 8 below).
4. We then calculate the number of bytes we can return to the client. This could be smaller than the number of bytes that we have left because the client may request fewer bytes than what we could give them.
5. If we have any bytes to return to the client (i.e. we haven't reached EOF), we perform the *MsgReply()* ourselves, giving the client *nbytes*' worth of data, starting at the output area plus the current offset.
6. As per POSIX, we update our ATIME flag, because we just accessed the device and returned more than zero bytes. We also move our offset to reflect the number of bytes we just returned to the client, so that we have the correct position within the file for the next time.
7. If, on the other hand, we were not returning any bytes to the client (i.e. we've reached EOF), we indicate this by doing a *MsgReply()* with zero bytes.
8. By returning _RESMGR_NOREPLY we're indicating to the QNX Neutrino resource manager framework that we've already called *MsgReply()* and that it should not.

Copyright © 2014, QNX Software Systems Limited

Notice that we used the OCB's *size* member rather than the attributes structure's *nbytes* member. This is because the image that we generated has a different size (shorter) than the size stored in the attributes structure's *nbytes* member. Since we want to return the correct number of bytes to the client, we use the smaller *size* number.

The *io_close_ocb()* function

```
static int
io_close_ocb (resmgr_context_t *ctp, void *reserved,
              RESMGR_OCB_T *ocb)
{
    if (ocb -> output) {
        free (ocb -> output);
        ocb -> output = NULL;
    }

    return (iofunc_close_ocb_default (ctp, reserved,
                                      &ocb -> base));
}
```

The *io_close_ocb()* function doesn't do anything special, apart from releasing the memory that may have been allocated in *io_read()*. The check to see if anything is present in the *output* member of the extended OCB structure is necessary because it's entirely possible that *io_read()* was never called and that member never had anything allocated to it (as would be the case with a simple *stat()* call — *stat()* doesn't cause *read()* to be called, so our *io_read()* would never get called).

The *render_7segment()* function

I won't go into great detail on the *render_7segment()* function, except to describe in broad terms how it works.

Here's the prototype for *render_7segment()*

```
void
render_7segment (char *digits,
                 unsigned char *r,
                 int xsize,
                 int ysize);
```

The parameters are:

digits

This is the ASCII string to render into the raw bitmap. Currently, the *render_7seg()* function understands the numbers 0 through 9, a blank, and the digits A through F (upper and lower case).

r

This is the raw graphics bitmap, allocated by the caller. It is stored as one byte per pixel, left to right, top to bottom (i.e. `r [7]` is X-coordinate 7, Y-coordinate 0).

xsize*, *ysize

This defines the size of the graphical bitmap. To write to an arbitrary (X, Y) location, add the X value to the product of the Y value and the *xsize* parameter.

As an exercise for the reader, you can extend the character set accepted by the *render_7segment()* function. You'll want to pay particular attention to the *seg7* array, because it contains the individual segment encodings for each character.

The *encode_image()* function

Finally, the *encode_image()* function is used to encode the raw bitmap array into a GIF compressed stream. This code was originally found in an antique version of FRACTINT (a Fractal graphics generator; see `http://fractint.org/`) that I had on my system from the BIX (Byte Information Exchange) days. I've simplified the code to deal only with a limited range of colors and a fixed input format. You'll need to provide your own version.

I won't go into great detail on the *encode_image()* function, but here's the prototype:

```
int
encode_image (unsigned char *raster,
              int x,
              int y,
              unsigned char *output);
```

The parameters are:

raster

The raw bitmap, in the same format as the input to the *render_7segment()* function, above.

x, y

The X and Y size of the raw bitmap.

output

The compressed output buffer, allocated by the caller.

Since we don't know *a priori* how big the compressed output buffer will be, I allocate one that's the same size as the input buffer. This is safe, because the nature of the 7-segment rendering engine is that there will be many “runs” (adjacent pixels with the same color), which are compressible. Also, the bitmap used is really only one bit (on or off), and the compressed output buffer makes full use of all 8 bits for the compressed streams. (If you modify this, beware that in certain cases, especially with “random” data, the compressed output size can be bigger than the input size.)

As another exercise, you can modify the webcounter resource manager to generate a JPEG or PNG file instead of, or in addition to, the GIF files that it currently generates.

The return value from the *encode_image()* function is the number of bytes placed into the compressed data buffer, and is used in *io_read()* to set the size of the resource.

Copyright © 2014, QNX Software Systems Limited

The code — phase 2

Now that we understand the basic functionality of the web counter, it's time to move on to phase 2 of the project.

In this second phase, we're going to add (in order of increasing complexity):

persistent count file

This stores the count to a file every time the count changes. It lets you shutdown and restart the resource manager without losing your count.

font selection

We're going to give the client the ability to choose from different fonts. Currently, we'll add only one additional font, because adding more fonts is a simple matter of font design.

ability to render in plain text

For simplicity, sometimes it's nice to get the data in a completely different manner, so this modification lets you use `cat` to view (and not increase) the count.

ability to *write()* to the resource

This lets you set the current value of the counter just by writing an ASCII text string to the counter resource.

Why these particular features, and not others? They're incremental, and they are more or less independent of each other, which means we can examine them one at a time, and understand what kind of impact they have on the code base. And, most of them are useful. `:-)`

The biggest modification is the ability to *write()* to the resource. What I mean by that is that we can set the counter's value by writing a value to it:

```
# echo 1234 >/dev/webcounter.gif
```

This will reset the counter value to 1234.

This modification will illustrate some aspects of accumulating data from a client's *write()* function, and then seeing how this data needs to be processed in order to interact correctly with the resource manager. We'll also look at extending this functionality later.

All of the modifications take place within `main.c` (with the exception of the font-selection modification, which also adds two new files, `8x8.c` and `8x8.h`).

Persistent count file

Soon after finishing the phase 1 web counter, I realized that it really needed to be able to store the current count somewhere, so that I could just restart the web counter and have it magically continue from where it left off, rather than back at zero.

The first change to accomplish this is to handle the -S option, which lets you specify the name of the file that stores the current web counter value. If you don't specify a -S then nothing different happens — the new version behaves just the same as the old version. Code-wise, this change is trivial — just add "`S:`" to the *getopt()* string list in the command line option processor (*optproc()* in `main.c`), and put in a handler for that option. The handler saves the value into the global variable *optS*.

At the end of option processing, if we have a -S option, we read the value from it (by calling *read_file()*), but only if we haven't specified a -s. The logic here is that if you want to reset the web counter to a particular value, you'd want the -s to override the count from the -S option. (Of course, you could always edit the persistent data file with a text editor once the resource manager has been shut down.)

Once the web counter has started, we're done reading from the persistent count file. All we need to do is update the file whenever we update the counter. This too is a trivial change. After we increment the counter's value in *io_read()*, we call *write_file()* to store the value to the persistent file (only if *optS* is defined; otherwise, there's no file, so no need to save the value).

Font selection

As it turns out, font selection is almost as trivial to implement as the persistent count file, above.

We had to add a -r option (for "render"), and a little bit of logic to determine which font was being specified:

```
case    'r':
    if (!strcmp (optarg, "8x8")) {
        optr = render_8x8;
    } else if (!strcmp (optarg, "7seg")) {
        optr = render_7segment;
    } else {
        // error message
        exit (EXIT_FAILURE);
    }
    break;
```

The specified font is selected by storing a function pointer to the font-rendering function in the *optr* variable. Once that's done, it's a simple matter of storing the function pointer into a new field (called *render*) within the OCB:

```
typedef struct my_ocb_s
{
  iofunc_ocb_t  base;
  unsigned char *output;
  int           size;
  void          (*render) (char *string,
                           unsigned char *bitmap,
                           int x, int y);
}   my_ocb_t;
```

Then, in *io_read()*, we replaced the hard-coded call to *render_7segment()*:

```
render_7segment (string, input, optx, opty);
```

with a call through the OCB's new pointer:

```
(*ocb -> render) (string, input, optx, opty);
```

Fairly painless. Of course, we needed to create a new font and font-rendering module, which probably took more time to do than the actual modifications. See the source in `8x8.c` for the details of the new font.

Plain text rendering

The next modification is to make the resource manager return an ASCII string instead of a GIF-encoded image. You could almost argue that this should have been "Phase 0," but that's not the way that the development path ended up going.

It would be simple to create a rendering module that copied the ASCII string sent to the "raw bitmap" array, and skip the GIF encoding. The code changes would be minimal, but there would be a lot of wasted space. The raw bitmap array is at least hundreds of bytes, and varies in size according to the X and Y sizes given on the command line. The ASCII string is at most 11 bytes (we impose an arbitrary limit of 10 digits maximum, and you need one more byte for the `NUL` terminator for the string). Additionally, having the resource manager register a name in the pathname space that ends in `.gif`, and then having text come out of the GIF-encoded "file" is a little odd.

Therefore, the approach taken was to create a second pathname that strips the extension and adds a different extension of "`.txt`" for the text version. We now have two attributes structures: one for the original GIF-encoded version and another for the text version.

This is a fairly common extension to resource managers in the field. Someone decides they need data to come out of the resource manager in a different format, so instead of overloading the meaning of the registered pathname, they add a second one.

The first change is to create the two attributes structures in *execute_resmgr()*. So, instead of:

```
static void
execute_resmgr (void)
{
    resmgr_attr_t           resmgr_attr;
    resmgr_connect_funcs_t  connect_func;
    resmgr_io_funcs_t       io_func;
    my_attr_t               attr;
    dispatch_t              *dpp;
    resmgr_context_t        *ctp;
    ...
```

we now have:

```
// one for GIF, one for text
static  my_attr_t   attr_gif;
static  my_attr_t   attr_txt;

static void
execute_resmgr (void)
{
    resmgr_attr_t           resmgr_attr;
    resmgr_connect_funcs_t  connect_func;
    resmgr_io_funcs_t       io_func;
    dispatch_t              *dpp;
    resmgr_context_t        *ctp;
    ...
```

Notice how we moved the two attributes structures out of the *execute_resmgr()* function and into the global space. You'll see why we did this shortly. Ordinarily, I'd be wary of moving something into global space. In this case, it's purely a scope issue — the attributes structure is a per-device structure, so it doesn't really matter where we store it.

Next, we need to register the two pathnames instead of just the one. There are some code modifications that we aren't going to discuss in this book, such as initializing both attributes structures instead of just the one, and so on. Instead of:

```
// establish a name in the pathname space
if (resmgr_attach (dpp, &resmgr_attr, optn, _FTYPE_ANY, 0,
                 &connect_func, &io_func, &attr) == -1) {
    perror ("Unable to resmgr_attach()\n");
    exit (EXIT_FAILURE);
}
```

we now have:

```
// establish a name in the pathname space for the .GIF file:
if (resmgr_attach (dpp, &resmgr_attr, optn, _FTYPE_ANY, 0,
                   &connect_func, &io_func, &attr_gif) == -1) {
    perror ("Unable to resmgr_attach() for GIF device\n");
    exit (EXIT_FAILURE);
}

// establish a name in the pathname space for the text file:
convert_gif_to_txt_filename (optn, txtname);
if (resmgr_attach (dpp, &resmgr_attr, txtname, _FTYPE_ANY, 0,
                   &connect_func, &io_func, &attr_txt) == -1) {
    perror ("Unable to resmgr_attach() for text device\n");
    exit (EXIT_FAILURE);
}
```

The *convert_gif_to_txt_filename()* function does the magic of stripping out the extension and replacing it with "`.txt`." It also handles other extensions by adding "`.txt`" to the filename.

At this point, we have registered two pathnames in the pathname space. If you didn't specify a name via -n, then the default registered names would be:

```
/dev/webcounter.gif
/dev/webcounter.txt
```

Copyright © 2014, QNX Software Systems Limited

Notice how we don't change the count based on reading the text resource. This was a conscious decision on my part — I wanted to be able to read the “current” value from the command line without affecting the count:

```
cat /dev/webcounter.txt
```

The rationale is that we didn't actually *read* the web page; it was an administrative read of the counter, so the count should not be incremented. There's a slight hack here, in that I reach into the GIF's attributes structure to grab the count. This is acceptable, because the binding between the two resources (the GIF-encoded resource and the text resource) is done at a high level.

Writing to the resource

The final change we'll make to our resource manager is to give it the ability to handle the client's *write()* requests. Initially, the change will be very simple — the client can write a number up to `MAX_DIGITS` (currently 10) digits in length, and when the client closes the file descriptor, that number will be jammed into the current count value. This lets you do the following from the command line:

```
echo 1433 >/dev/webcounter.gif
```

or

```
echo 1433 >/dev/webcounter.txt
```

We're not going to make any distinction between the GIF-encoded filename and the text filename; writing to either will reset the counter to the specified value, 1433 in this case (yes, it's a little odd to write a plain ASCII string to a `.gif` file).

Adding the *io_write()* handler

If we're going to be handling the client's *write()* function, we need to have an *io_write()* handler. This is added in *execute_resmgr()* to the table of I/O functions, right after the other functions that we've already added:

```
// override functions in "connect_func" and
// "io_func" as required here
connect_func.open = io_open;
io_func.read      = io_read;
io_func.close_ocb = io_close_ocb;
io_func.write     = io_write;     // our new io_write handler
```

What are the characteristics of the *io_write()* handler?

First of all, it must accumulate characters from the client. As with the *io_read()* handler, the client can “dribble” in digits, one character at a time, or the client can *write()* the entire digit stream in one *write()* function call. We have to be able to handle all of the cases. Just like we did with the *io_read()* handler, we'll make use of the OCB's *offset* member to keep track of where we are in terms of reading data from the client. The *offset* member is set to zero (by the *calloc()* in *io_open()*) when the resource is opened, so it's already initialized.

Copyright © 2014, QNX Software Systems Limited

We need to ensure that the client doesn't overflow our buffer, so we'll be comparing the *offset* member against the size of the buffer as well.

We need to determine when the client is done sending us data. This is done in the *io_close_ocb()* function, and not in the *io_write()* function.

The *io_write()* function

Let's look at the *io_write()* function first, and then we'll discuss the code:

```
static int
io_write (resmgr_context_t *ctp, io_write_t *msg, RESMGR_OCB_T *ocb)
{
  int     nroom;
  int     nbytes;

  // 1) we don't do any xtypes here...
  if ((msg -> i.xtype & _IO_XTYPE_MASK) != _IO_XTYPE_NONE) {
    return (ENOSYS);
  }

  // standard helper function
  if ((sts = iofunc_write_verify (ctp, msg, &ocb -> base, NULL)) != EOK) {
    return (sts);
  }

  // 2) figure out how many bytes we can accept in total
  nroom = sizeof (ocb -> wbuf) - 1 - ocb -> base.offset;

  // 3) and how many we can accept from the client
  nbytes = min (nroom, msg -> i.nbytes);

  if (nbytes) {
    // 4) grab the bytes from the client
    memcpy (ocb -> wbuf + ocb -> base.offset, &msg -> i + 1, nbytes);

    // 5) update flags and offset
    ocb -> base.attr -> base.flags |=
        IOFUNC_ATTR_MTIME | IOFUNC_ATTR_DIRTY_TIME;
    ocb -> base.offset += nbytes;
  } else {
    // 6) we're full, tell them
    if (!nroom) {
      return (ENOSPC);
    }
  }

  // 7) set the number of returning bytes
  _IO_SET_WRITE_NBYTES (ctp, nbytes);
  return (EOK);
}
```

The *io_write()* function performs the following steps:

1. Just like in the *io_read()* handler, we fail the attempt to perform any XTYPE operations.
2. We determine how much room we have available in the buffer by subtracting the current offset from its size.
3. Next, we determine how many bytes we can accept from the client. This is going to be the smaller of the two numbers representing how much room we have, and how many bytes the client wants to transfer.
4. If we are transferring any bytes from the client, we do that via *memcpy()*. (See note after step 7 below!)

Copyright © 2014, QNX Software Systems Limited

5. POSIX says that the MTIME time field must be updated if transferring more than zero bytes, so we set the “MTIME is dirty” flag.
6. If we can't transfer any bytes, and it's because we have no room, we give the client an error indication.
7. Finally, we tell the resource manager framework that we've processed *nbytes* worth of data, and that the status was EOK.

In step 4, we *assume* that we have all of the data! Remember that the resource manager framework doesn't necessarily read in all of the bytes from the client — it reads in only as many bytes as you've specified in your *resmgr_attr.msg_max_size* parameter to the *resmgr_attach()* function (and in the network case it may read in less than that). However, we are dealing with *tiny* amounts of data — ten or so bytes at the most, so we are safe in simply *assuming* that the data is present. For details on how this is done “correctly,” take a look at the RAM-disk Filesystem (p. 135) chapter.

The *io_close_ocb()* function

Finally, here are the modifications required for the *io_close_ocb()* function:

```
static int
io_close_ocb (resmgr_context_t *ctp, void *reserved, RESMGR_OCB_T *ocb)
{
    int     tmp;

    if (ocb -> output) {
        free (ocb -> output);
        ocb -> output = NULL;
    }

    // if we were writing, parse the input
    // buffer and possibly adjust the count
    if (ocb -> base.ioflag & _IO_FLAG_WR) {
        // ensure NUL terminated and correct size
        ocb -> wbuf [optd + 1] = 0;
        if (isdigit (*ocb -> wbuf)) {
            attr_gif.count = attr_txt.count = tmp = atoi (ocb -> wbuf);
            if (optS) {
                write_file (optS, tmp);
            }
        }
    }

    return (iofunc_close_ocb_default (ctp, reserved, &ocb -> base));
}
```

All that's different in the *io_close_ocb()* function is that we look at the OCB's *ioflag* to determine if we were writing or not. Recall that the *ioflag* is the “open mode” plus one — by comparing against the constant _IO_FLAG_WR we can tell if the “write” bit is set. If the write bit is set, we were writing, therefore we process the buffer that we had been accumulating in the *io_write()*. First of all, we NULL-terminate the buffer at the position corresponding to the number of digits specified on the command line (the -d option, which sets *optd*). This ensures that the *atoi()* call doesn't run off the end of the string and into Deep Outer Space (DOS) — this is redundant because we *calloc()* the entire OCB, so there is a NULL there anyway. Finally, we write the persistent counter file if *optS* is non-NULL.

We check to see if the first character is indeed a digit, and jam the converted value into both attribute structures' *count* members. (The *atoi()* function stops at the first non digit.)

This is where you could add additional command parsing if you wanted to. For example, you might allow hexadecimal digits, or you might wish to change the background/foreground colors, etc., via simple strings that you could echo from the command line:

```
echo "bgcolor=#FFFF00" >/dev/webcounter.gif
```

This is left as an exercise for the reader.

Copyright © 2014, QNX Software Systems Limited

The code — phase 3

In the last phase of our project, we're going to change from managing one file at a time to managing multiple counters.

What this means is that we'll take over a directory instead of a file or two, and we'll be able to present multiple counters. This is useful in the real world if, for example, you want to be able to maintain separate counters for several web pages.

Before we tackle that, though, it's worthwhile to note that you could achieve a similar effect by simply performing more *resmgr_attach()* calls; one pair of resources (the GIF-encoded and the text file) per web counter. The practical downside of doing this is that if you are going to be attaching a *lot* of pathnames, QNX Neutrino's process manager will need to search through a linear list in order to find the one that the client has opened. Once that search is completed and the resource is opened, however, the code path is identical. All we're doing by creating a directory instead of a pathname is moving the pathname-matching code into our resource manager instead of the process manager.

The main differences from the previous version will be:

- We specify the directory flag instead of the file flag to *resmgr_attach()*.
- There's more complex processing in *io_open()*.
- There's more complex processing when dealing with our persistent count files.

Filename processing tricks

There are a number of “tricks” that we can play when we manage our own pathname space. For example, instead of having a plain filename for the web resource, we can have a built-in command in the resource filename, like this:

```
/dev/webcounters/counter-00.gif/fg=#ffff00,bg=#00ffa0
```

(I would have really liked to put a “?” character instead of the last “/” character, but web-browsers strip off anything after (and including) the “?” character; plus it would be cumbersome to use within the shell). Here, we're accessing the `/dev/webcounters/counter-00.gif` resource, and “passing” it the arguments for the foreground and background colors.

The process manager doesn't care about the pathname after our registered mount point. In this example, our mount point is just `/dev/webcounters` — anything after that point is passed on to our *io_open()* as a text string. So in this case, our *io_open()* would receive the string:

```
counter-00.gif/fg=#ffffff0,bg=#00ffa0
```

How we choose to interpret that string is entirely up to us. For simplicity, we won't do any fancy processing in our resource manager, but I wanted to point out what *could* be done if you wanted to.

Our resource manager will accept a fixed-format string, as suggested above. The format is the string "`counter-`" followed by two decimal digits, followed by the string "`.gif`" and nothing further. This lets our *io_open()* code parse the string quite simply, and yet demonstrates what you can do.

This is one of the reasons that our pathname parsing will be faster than the generic linear search inside of the process manager. Since our filenames are of a fixed form, we don't actually "search" for anything, we simply convert the ASCII number to an integer and use it directly as an index.

The default number of counters is set to 100, but the command-line option -N can be used to set a different number.

We're also going to reorganize the storage file format of the persistent counter a little bit just to make things simpler. Rather than have 100 files that each contain one line with the count, instead we're going to have one file that contains 100 32-bit binary integers (i.e. a 400-byte file).

Changes

There are a number of architectural changes required to go from a single pair of file-type resources to a directory structure that manages a multitude of pairs of file-type resources.

Globals

The first thing I did was add a few new global variables, and modify others:

optN*, *optNsize

(new) This is the number of counters, and the number of digits required to represent the number of counters.

attr_gif*, *attr_txt

(modified) These are two attributes structures, one for GIF-encoded files and one for text files. I've modified them to be arrays rather than scalars.

attr

(new) This is the attributes structure for the directory itself.

You'll notice that the attributes structures and the OCB remain the same as before; no changes are required there.

Copyright © 2014, QNX Software Systems Limited

The new-and-improved *execute_resmgr()*

We need to change the *execute_resmgr()* function a little. We're no longer registering a pair of file-type resources, but rather just a single directory.

Therefore, we need to allocate and initialize the arrays *attr_gif* and *attr_txt* so that they contain the right information for the GIF-encoded and text files:

```
// initialize the individual attributes structures
for (i = 0; i < optN; i++) {
  iofunc_attr_init (&attr_gif [i].base, S_IFREG | 0666, 0, 0);
  iofunc_attr_init (&attr_txt [i].base, S_IFREG | 0666, 0, 0);

  // even inodes are TXT files
  attr_txt [i].base.inode = (i + 1) * 2;

  // odd inodes are GIF files
  attr_gif [i].base.inode = (i + 1) * 2 + 1;
}
```

It's important to realize that the attributes structure's *inode* (or “file serial number”) member plays a key role. First of all, the inode cannot be zero. To QNX Neutrino, this indicates that the file is no longer in use. Therefore, our inodes begin at 2. I've made it so that even-numbered inodes are used with the text files, and odd-numbered inodes are used with the GIF-encoded files. There's nothing saying *how* you use your inodes; it's completely up to you how you interpret them — so long as all inodes in a particular filesystem are unique.

We're going to use the inode to index into the *attr_gif* and *attr_txt* attributes structures. We're also going to make use of the even/odd characteristic when we handle the client's *read()* function.

Next, we initialize the attributes structure for the directory itself:

```
iofunc_attr_init (&attr.base, S_IFDIR | 0777, 0, 0);

// our directory has an inode of one.
attr.base.inode = 1;
```

Notice the `S_IFDIR | 0777` — this sets the *mode* member of the attributes structure to indicate that this is a directory (the `S_IFDIR` part) and that the permissions are 0777 — readable, writable, and seekable by all.

Finally, we register the pathname with the process manager:

```
if (resmgr_attach (dpp, &resmgr_attr, optn, _FTYPE_ANY,
    _RESMGR_FLAG_DIR, &connect_func, &io_func, &attr) == -1) {
    perror ("Unable to resmgr_attach()\n");
    exit (EXIT_FAILURE);
}
```

Notice that this time there is only one *resmgr_attach()* call — we're registering only one pathname, the directory. All of the files underneath the directory are managed by our resource manager, so they don't need to be registered with the process manager. Also notice that we use the flag `_RESMGR_FLAG_DIR`. This tells the process manager that it should forward any requests *at and below* the registered mount point to our resource manager.

Copyright © 2014, QNX Software Systems Limited

Option processing

The change to the option processing is almost trivial. We added "`N:`" to the list of options processed by *getopt()* and we added a `case` statement for the -N option. The only funny thing we do is calculate the number of digits that the number of counters will require. This is done by calling *sprintf()* to generate a string with the maximum value, and by using the return value as the count of the number of digits. We need to know the size because we'll be using it to match the filenames that come in on the *io_open()*, and to generate the directory entries.

Finally, we initialize the attributes structure in the option processor (instead of *execute_resmgr()* as in previous versions) because we need the attributes structures to exist before we call *read_file()* to read the counter values from the persistent file.

Handling *io_read()*

This is where things get interesting. Our *io_read()* function gets called to handle three things:

1. a *read()* of the text counter value
2. a *read()* of the GIF-encoded counter picture
3. a *readdir()* of the directory (e.g. `ls /dev/webcounters`)

The first two operate on a file, and the last operates on a directory. So that's the first decision point in our new *io_read()* handler:

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

  // use the helper function to decide if valid
  if ((sts = iofunc_read_verify (ctp, msg, &ocb -> base, NULL)) != EOK) {
    return (sts);
  }

  // decide if we should perform the "file" or "dir" read
  if (S_ISDIR (ocb -> base.attr -> base.mode)) {
    return (io_read_dir (ctp, msg, ocb));
  } else if (S_ISREG (ocb -> base.attr -> base.mode)) {
    return (io_read_file (ctp, msg, ocb));
  } else {
    return (EBADF);
  }
}
```

By looking at the attributes structure's *mode* field, we can tell if the request is for a file or a directory. After all, we set this bit ourselves when we initialized the attributes structures (the `S_IFDIR` and `S_IFREG` values).

Operating on a file

If we are handling a file, then we proceed to *io_read_file()*, which has changed slightly from the previous version:

```
...
```

Copyright © 2014, QNX Software Systems Limited

```
// 1) odd inodes are GIF files, even inodes are text files
if (ocb -> base.attr -> base.inode & 1) {
  if (!ocb -> output) {
    ...
    // 2) allocate the input and output structures as before
    ...

    sprintf (string, "%0*d", optd, ocb -> base.attr -> count++);
    (*ocb -> render) (string, input, optx, opty);
    ocb -> size = ocb -> base.attr -> base.nbytes = encode_image (
      input, optx, opty, ocb -> output);

    // 3) note the changes to write_file()
    if (optS) {
      write_file (optS, ocb -> base.attr -> base.inode / 2 - 1,
                 ocb -> base.attr -> count);
      }
  }
} else { // 4) even, so must be the text attribute
  int     tmp;  // limited scope

  ocb -> base.attr -> count =
      attr_gif [ocb -> base.attr -> base.inode / 2 - 1].count;
  tmp = sprintf (string, "%0*d\n", optd, ocb -> base.attr -> count);
  if (ocb -> output) {
    free (ocb -> output);
  }
  ocb -> output = strdup (string);
  ocb -> size = tmp;
}
```

Notice a few things:

1. We determine if we are dealing with the GIF-encoded file or the text file by looking at the *inode* member of the attributes structure. This is why we made the inodes odd for GIF-encoded and even for text, so that we could tell them apart easily.
2. Code not shown for brevity, no change from previous.
3. I've added an extra parameter to *write_file()*, namely the counter number. This lets *write_file()* seek into the correct spot in the persistent counter file and make a tiny *write()* rather than writing out the entire file.
4. If we are dealing with the text file, then we need to get the count. However, the "real" value of the count is only maintained in the GIF-encoded file's attributes structure. Therefore, we need to use the *inode* as an index into the array of GIF-encoded attributes structures in order to find the correct one. This is why we made the inodes sequential, so that there's a direct mapping between the inode number and the index for the array of either attributes structure. Also notice that we check to see if we already have memory allocated for the string. If so, we *free()* it first.

What might appear to be "clever" use of inodes is in fact standard programming practice. When you think about it, a disk-based filesystem makes use of the inodes in a similar manner; it uses them to find the disk blocks on the media.

Operating on a directory

Our resource manager needs to be able to handle an `ls` of the directory.

While this isn't an absolute requirement, it's a "nice-to-have." It's acceptable to simply lie and return nothing for the client's *readdir()*, but most people consider this tacky and lazy programming. As you'll see below, it's not rocket science.

I've presented the code for returning directory entries in the “Resource Managers” chapter of my previous book, *Get Programming with the QNX Neutrino RTOS*. This code is a cut-and-paste from the `atoz` resource manager example, with some important changes.

```
#define ALIGN(x) (((x) + 3) & ~3)
```

This is an alignment macro to help align things on a 32-bit boundary within the `struct dirent` that we are returning.

```
static int
io_read_dir (resmgr_context_t *ctp, io_read_t *msg,
             RESMGR_OCB_T *ocb)
{
    int     nbytes;
    int     nleft;
    struct  dirent *dp;
    char    *reply_msg;
    char    fname [PATH_MAX];

    // 1) allocate a buffer for the reply
    reply_msg = calloc (1, msg -> i.nbytes);
    if (reply_msg == NULL) {
        return (ENOMEM);
    }

    // 2) assign output buffer
    dp = (struct dirent *) reply_msg;

    // 3) we have "nleft" bytes left
    nleft = msg -> i.nbytes;

    while (ocb -> base.offset < optN * 2) {

        // 4) create the filename
        if (ocb -> base.offset & 1) {
            sprintf (fname, "counter-%0*d.gif",
                   optNsize, (int) (ocb -> base.offset / 2));
        } else {
            sprintf (fname, "counter-%0*d.txt",
                   optNsize, (int) (ocb -> base.offset / 2));
        }

        // 5) see how big the result is
        nbytes = dirent_size (fname);

        // 6) do we have room for it?
        if (nleft - nbytes >= 0) {

            // 7) fill the dirent, advance the dirent pointer
            dp = dirent_fill (dp, ocb -> base.offset + 2,
                           ocb -> base.offset, fname);

            // 8) move the OCB offset
            ocb -> base.offset++;

            // 9) account for the bytes we just used up
            nleft -= nbytes;
        } else {

            // 10) don't have any more room, stop
            break;
        }
    }

    // 11) return info back to the client
    MsgReply (ctp -> rcvid, (char *) dp - reply_msg,
             reply_msg, (char *) dp - reply_msg);
```

Copyright © 2014, QNX Software Systems Limited

```
    // 12) release our buffer
    free (reply_msg);

    // 13) tell resmgr library we already did the reply
    return (_RESMGR_NOREPLY);
}
```

Now I'll walk you through the code:

1. We're generating data, so we must allocate a place to store our generated data. The client has told us how big their buffer is (the *nbytes* member of the incoming message), so we allocate a buffer of that size.
2. For convenience, we're going to use a pointer to `struct dirent` to write our data into the buffer. Unfortunately, the `struct dirent` is a variable size (with implicit data fields following the end of the structure) so we can't just simply use an array; we'll need to do pointer arithmetic.
3. Just like when we are returning file data to the client, we need to see how many bytes we have available to us. In this case, we don't want to overflow our allocated memory. The `while` loop runs until we have returned all of the data items to the client, with an early out at step 10 in case the data buffer is full.
4. Once again we use the odd/even aspect of the inode to determine whether we are dealing with the GIF-encoded file or the text file. This time, however, we're generating the inodes ourselves (see note after step 13 below). Depending on what type of file we are returning, we call the appropriate version of *sprintf()*. Also note the use of the *optNsize* variable to generate the correct number of digits in the filename.
5. The *nbytes* variable holds the size of the new, proposed `struct dirent`. It might not fit, so we check that in the next step. The helper routine *dirent_size()* is discussed below.
6. If we have room for the new `struct dirent` we proceed; else we go to step 10.
7. Now that we know that the proposed size of the `struct dirent` is okay, we proceed to fill the information by calling the helper routine *dirent_fill()* (discussed below). Notice that we add 2 to the OCB's *offset* member. That's because our first inode number is 2 (1 is reserved for the directory entry itself, and 0 is invalid). Also notice that *dirent_fill()* returns a new pointer to where the next directory entry should go; we assign this to our *dp* pointer.
8. Next we increment the OCB's *offset* member. This is analogous to what we did when returning file data (in *io_read_file()*) in that we are making note of where we last were, so that we can resume on the next *readdir()* call.
9. Since we wrote *nbytes* through the *dp* pointer in step 7, we need to account for these bytes by subtracting them from the number of bytes we still have available.
10. This is the “early-out” step that just breaks out of the `while` loop in case we've run out of room.

11. Just like when we handle a file, we need to return the data to the client. Unlike when we handle a file, we're returning data from our own allocated buffer, rather than the text buffer or the GIF-encoded output buffer.
12. Clean up after ourselves.
13. Finally, we tell the resource manager library that we did the reply, so that it doesn't need to.

In step 4, we need to make note of the relationship between inode values and the *offset* member of the OCB. The meaning of the *offset* member is entirely up to us — all that QNX Neutrino demands is that it be consistent between invocations of the directory-reading function. In our case, I've decided that the *offset* member is going to be directly related to the array index (times 2) of the two arrays of attributes structures. The array index is directly related to the actual counter number (i.e. an array index of 7 corresponds to counter number 7).

```
static int
dirent_size (char *fname)
{
  return (ALIGN (sizeof (struct dirent) - 4 + strlen (fname)));
}

static struct dirent *
dirent_fill (struct dirent *dp, int inode, int offset, char *fname)
{
    dp -> d_ino = inode;
    dp -> d_offset = offset;
    strcpy (dp -> d_name, fname);
    dp -> d_namelen = strlen (dp -> d_name);
    dp -> d_reclen = ALIGN (sizeof (struct dirent)
                   - 4 + dp -> d_namelen);
    return ((struct dirent *) ((char *) dp + dp -> d_reclen));
}
```

These two utility functions calculate the size of the directory entry (*dirent_size()*) and then fill it (*dirent_fill()*).

The persistent counter file

Finally, the last thing to modify is the utilities that manage the persistent counter file, *read_file()* and *write_file()*.

The changes here are very simple. Instead of maintaining one counter value in the file (and in ASCII text at that), we maintain all of the counter values in the file, in binary (4 bytes per counter, native-endian format).

This implies that *read_file()* needs to read the entire file and populate all of the counter values, while *write_file()* needs to write the value of the counter that has changed, and leave the others alone. This also, unfortunately, implies that we can no longer modify the file "by hand" using a regular text editor, but instead must use something like `spatch` — this is fine, because we still have the ability to *write()* a number to the resource manager's "files."

I've used the buffered I/O library (*fopen()*, *fread()*, *fwrite()*)) in the implementation, and I *fopen()* and *fclose()* the file for each and every update. It would certainly be more efficient to use the lower-level I/O library (*open()*, *read()*, *write()*, etc.), and consider leaving the file open, rather than opening and closing it each time. I stayed with buffered I/O due to, yes, I'll admit it, laziness. The original used *fopen()* and *fprintf()* to print the counter value in ASCII; it seemed easier to change the *fprintf()* to an *fseek()*/*fwrite()*.

Enhancements

The ability to have each web counter be a different size (and that information stored in a persistent configuration file), or to have the size specified as a “trailer” in the pathname, is something that can be added fairly easily. Try it and see!

Copyright © 2014, QNX Software Systems Limited

References

The following references apply to this chapter.

Header files

- `<sys/dirent.h>`

Functions

See the following functions in the QNX Neutrino *C Library Reference*:

- *readdir()*
- *read()*
- *write()*

Books

Get Programming with the QNX Neutrino RTOS's “Resource Managers” chapter goes into exhaustive detail about handling resource managers.

Chapter 5
ADIOS — Analog/Digital I/O Server

There I was, on my summer break, preparing to do a bunch of painting and drywalling in the basement when the phone rings. It's a former student of mine asking if I'd like to help with a data acquisition project. Well, how could I resist? A few days later, FedEx showed up with a bunch of analog and digital I/O cards, and the festivities began.

This chapter documents the ADIOS project. ADIOS stands for *A*nalog/*D*igital *I*nput/*O*utput *S*erver. ADIOS consists of a series of inter-related modules, which I'll describe here.

ADIOS was developed under contract to Century Aluminum in Kentucky (USA), and I'd like to thank them for letting me publish the source code in this book!

Requirements

Century Aluminum has several production lines where they smelt aluminum. Each line has several analog and digital I/O points that they monitor so they can feed data into the proprietary process control software that controls the line.

I was involved on the low-level driver and data acquisition side, and have no idea what they do with the data once it leaves the QNX Neutrino box.

The requirements for this contract came in two sets. The first set of requirements was to support three different hardware cards. For those of you not familiar with the industrial automation/process control world, the next statement will come as a surprise. The buses on all three cards are ISA (*not* PCI). These folks use rack-mounted PC's (with modern Pentium 3-class CPUs), with a huge ISA bus — some of them have room for up to *twenty* ISA cards! The three cards are the DIO-144 (144 digital I/O pins), PCL-711 (16 digital I/O, 8 analog inputs, and 1 analog output), and the ISO-813 (32 analog inputs).

The second set of requirements was for something to periodically read the data from the cards, and put that data into shared memory. The idea was that Century Aluminum's control software would then grab samples out of shared memory, do the processing, and control the lines.

As with all software projects, there were some extras that got included into the contract just because they came in handy during development. One of these extras was a `showsamp` utility that displayed samples out of the shared memory maintained by ADIOS. Another one was called `tag` and it lets you query or set various analog or digital I/O channels. Finally, a master parser was created to handle the configuration file. This was incorporated into the drivers, ADIOS, and `tag`, so everything could be referred to by a symbolic name (like `OVEN_CONTROL_1`, rather than “card at address 0x220, channel 3, port B, bit 4”). For analog ports, the configuration file also contained range information (so that a 12-bit analog converter might actually be scaled to represent voltages from -5V to +5V, rather than a raw binary value from `0x000` to `0xFFF`).

So, in this chapter, we'll look at the following pieces of software:

- Card drivers: `pcl711`, `dio144`, and `iso813`
- ADIOS server: `adios`
- Show sample utility: `showsamp`
- Tag utility: `tag`

We'll also discuss some of the common code that's shared between modules.

This high-level diagram shows the pieces that we'll be discussing:

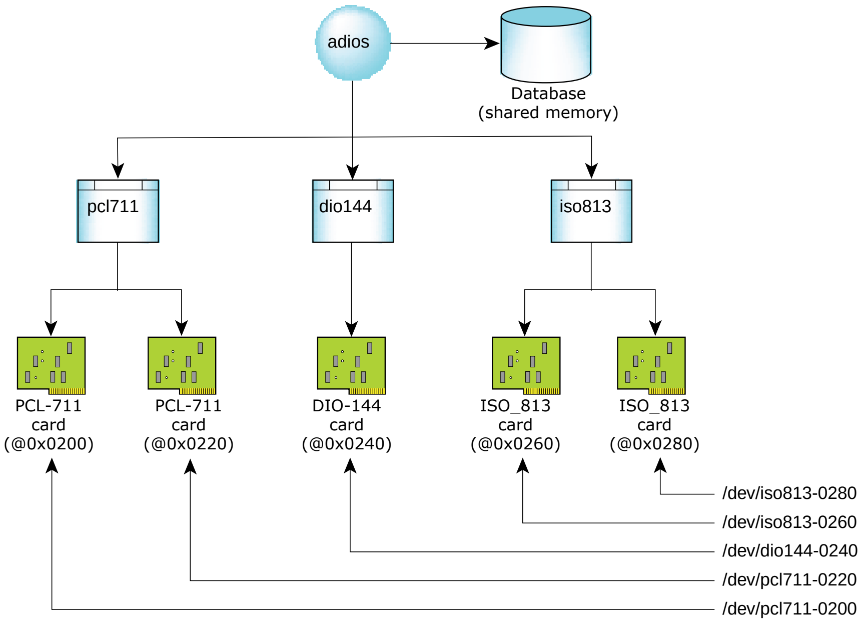


Figure 17: The relationship between ADIOS, the shared memory database, the hardware drivers, and their cards.

Design

Most of the design was fairly straightforward — we needed to configure each card, and we had to be able to get and set the analog and digital I/O points. For the shared memory interface, I had to figure out how to minimize the amount of work needed for the process control software to access data in the shared memory (an API library was created, along with some demo software).

Driver Design

The first design decision I made was that all of the device drivers should accept the same set of *devctl()* commands. This meant that to anyone using the device drivers, they'd all look the same. One could argue that sending a “turn on digital bit 1” command to a card that only has analog input doesn't make sense. That's true — in that case, the card simply returns ENOSYS. But the important thing is that we didn't *reuse* the command number for something else — all cards know about that command, but only some may support it.

The next design decision was that each driver would support one or more cards. This way, instead of having multiple pcl711 drivers running for multiple PCL-711 cards, there'd be only one driver running. Each driver creates multiple mount points under /dev — one for each card. So in a system with three PCL-711 cards and one DIO-144 card (at addresses 0x220, 0x240, 0x260 and 0x280, respectively) you'd see the following devices:

```
/dev/pcl711-0220
/dev/pcl711-0240
/dev/pcl711-0260
/dev/dio144-0280
```

The three PCL-711 devices are managed by the one pcl711 driver, and the one DIO-144 device is managed by the one dio144 driver. At one point I considered having just one mount point for each driver (e.g. /dev/pcl711), instead of multiple mount points, but it turned out to be much easier to consider each device as a separate entity.

You'll notice that we didn't take over /dev/pcl711 as a directory and create the individual devices underneath. This is because the number of devices is limited by the number of ISA slots, which, even in the industrial automation space, is still only a dozen or two. It wasn't worth the extra effort of maintaining a directory hierarchy in the driver.

As mentioned above, all three of the supported cards are ISA cards, so there's no automatic configuration to deal with — whatever you set the I/O port address to on the card is where the card will be. (The one thing that did get me, though, is that on

Copyright © 2014, QNX Software Systems Limited

the card's I/O port selector switch, “ON” is a zero and “OFF” is a one. That had me scratching my head for a few hours.)

DIO-144

The DIO-144 card has 144 bits of digital I/O, and you can configure them as inputs or outputs by 8-bit or (for some ports) 4-bit chunks. This gives you a fair bit of flexibility, but also presents some problems for the driver. There is no analog I/O on this card.

All of the programming for this card is done by reading and writing I/O ports.

For those of you not familiar with the x86 architecture, an x86 CPU maintains two separate address spaces. One is the traditional memory address space that every CPU in the world has, and the second is called an *I/O address space*. There are special CPU instructions to access this second address space. From C, you can access the I/O address space using the *in8()* and *out8()* (and related) functions (see `<hw/inout.h>` and *mmap_device_io()*).

The hardest part of the DIO-144 software driver design is setting up the bits to be input or output. Once you have the card configured, reading from it involves calling several *in8()* functions, and writing involves calling several *out8()* functions. The only trick to writing is that you often need to change just one bit — so the software will have to keep a current image of the other 7 bits, because the hardware allows only 8-bit-at-a-time writes.

ISO-813

The ISO-813 card has 32 12-bit analog input channels. In reality, this is implemented as a 32-channel multiplexer that feeds into one analog-to-digital (A/D) converter. The trick with this card is that you need to tell the multiplexer which channel it should be reading, set up the gain (1x, 2x, 4x, 8x, or 16x), wait for the multiplexer and A/D to stabilize, and *then* trigger an A/D conversion. When the A/D conversion is complete, you can read the 12-bit result from an I/O port.

PCL-711

The PCL-711 card is the most versatile of the three. It has 16 digital inputs and 16 digital outputs (these are simple; you just read or write an I/O port). It has one analog output (this too is fairly simple; you just write the 12-bit value to an I/O port). It also has 8 analog inputs (these function like the ISO-813's analog input ports). There is no configuration for the digital I/O; they're fixed. The configuration of the analog input is the same as the ISO-813 card — mainly the selection of the gain.

Copyright © 2014, QNX Software Systems Limited

Note that for both the ISO-813 and the PCL-711 cards, there are jumpers on the card to control the input range and whether the input is single-ended or differential. We're not going to discuss these jumpers further, because they have no impact on the software design.

Shared Memory Design

Another design consideration was the shared memory driver and the layout of the shared memory. We'll see the details when we look at the `adios` server itself.

The shared memory design involved setting up the layout of the data areas, as well as figuring out a method to ensure that the control process would get valid samples. To achieve this, I divided the shared memory into two sections, each on a 4k page boundary. The first section is a database section that has information about which drivers are being polled for data, how many analog and digital I/O points they have, as well as the current head and tail index into the data section. The second section is the data section, and it contains the actual data that's been acquired from the cards. Note that the shared memory contains only analog and digital inputs — when the software needs to write to an analog or digital output, it contacts the driver directly. We'll see an example of this when we look at the `tag` utility.

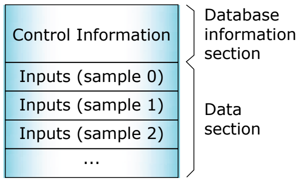


Figure 18: The shared memory layout.

Tags database design

The ability to associate ASCII labels with data points evolved during the term of the project. Initially, there was no requirement for it — as long as `adios` put the samples into shared memory, everyone was happy. When the time came to configure the analog and digital I/O ports, we decided we needed a way of saving that configuration. The tags database was really a text-file-based configuration database initially. Then I added the `tag` keyword, and the *tags database* was born.

Before we proceed with the details of the software, it's a good idea to explain the tags database because it's common to the drivers, `adios`, and the `tag` utility.

At a high level, the tags database is a flat ASCII text file. The default name is `adios.cfg`, and the utilities search for it in `/etc` first and then in the local directory (unless you override the name).

Copyright © 2014, QNX Software Systems Limited

Configuration (and tag) information is presented card-by-card and spans multiple lines. It's important to keep in mind that this information is used by both the drivers *and* utilities. The drivers use it to configure their I/O ports, and the utilities use it to make sense of the shared memory region maintained by `adios`.

The `device` keyword indicates the beginning of configuration for a particular card. Recall that the `dio144`, `pcl711`, and `iso813` drivers create device names of the form `/dev/`*driver*-*port* (e.g. `/dev/pcl711-0220` for a PCL-711 card at port address `0x220`). The `device` keyword uses the same format.

Once a `device` keyword has been read, all further configuration information in the file applies to that device, until another `device` keyword (or the end of file). Here's the formal grammar (I've broken the `ccfga` and `cfgd` keyword definitions over two lines):

```
cfgfile  :: [devdirs]... ["#" comment]

devdirs  :: devs [chandirs]...

devs     :: "device" devicename

chandirs :: "channel" channum [ [portspec] [cfg]... ]...

portspec :: "port" <"A" | "B" | "C">

cfg      :: <cfga | cfgd>

cfga     :: <"ain" | "aout"> ["span" loval","hival]
              ["gain" gainval] ["tag" tagname]

cfgd     :: "bit" bitnum [<"din" | "dout">]
            [<"positive" | "negative">] ["tag" tagname]
```

So, as a short example:

```
device /dev/pcl711-0220
  channel 1 ain span 0,70 tag BOILER_TEMPERATURE
  channel 2
    ain
    span -3,+4.2285
    gain 2
    tag VOLTAGE_1
  channel 3 ain span +100,-100 tag PRESSURE_1
  # channels 4 and 5 are not used here
  channel 6 ain tag RAW_ANALOG_1
  channel
    7 ain
      span 0,1 tag spud

  # Channel 8 is not used

  # Channel 9 is the output channel
  channel 9 aout span 0,8191 tag OUTPUT_TEMPERATURE_1

  # Channel 10 is the digital input channel with 8 bits
  channel 10
    bit 0 din negative tag CHECK_1
    bit 1 din positive tag CHECK_2
    din bit 2 positive tag SWITCH_1
    din bit 3 positive tag SWITCH_2
    # bits 4 through 7 are not used

  # Channel 11 is the digital output channel with 8 bits
  channel 11
    bit 0 dout positive tag CONVEYOR_MOTOR_1
    bit 1 negative tag CONVEYOR_START
    bit 2 negative tag ALARM_1_LED
    bit 3 dout negative tag ALARM_2_LED
    # bits 4 through 7 are not used
```

This configures a PCL-711 driver at port `0x220`.

- Channel 1 is an analog input (the `ain` keyword). Its 12-bit range is to be interpreted as a range between 0 and 70, and it has a tag associated with it called `BOILER_TEMPERATURE`.
- Channel 2 is also an analog input, has a range of -3 to +4.2285, and has a gain setting of 2. Its tag is called `VOLTAGE_1`. Notice that it spans multiple lines in the configuration file.
- Channel 3 is similar to channel 1.
- Channels 4 and 5 are not used, so we simply put in a comment.
- Channel 6 is a raw analog input value, with no scaling.
- Channel 7 is similar to channel 1.
- Channel 10 is a digital input channel. Notice how we specify each bit's characteristics. The `positive` and `negative` commands are used to determine if the bit's value is true or complemented.
- Channel 11 is the digital output channel. Notice how bits 1 and 2 don't explicitly say `dout` to indicate a digital output — this is inferred by the software.

The gain settings are simply numbers passed directly to the driver. So, a gain setting of 2 doesn't mean that the gain setting is 2 ×, it simply means that the PCL-711 driver will receive the constant `2` and interpret it accordingly (i.e. in this case it means gain setting number 2, which is actually a 4 × gain setting.)

Copyright © 2014, QNX Software Systems Limited

The Driver Code

The drivers are all similar, so we'll examine only the PCL-711 driver (`pcl711`) because it's a superset of the other two (the DIO-144 and the ISO-813).

Theory of operation

The first thing each driver does is read its configuration file. There is usually only one common configuration file, which describes the configuration of the entire system. Each driver must read through the configuration file, looking for entries that apply to it and ignoring entries for other drivers.

Once the configuration file is read and parsed, the individual fields are validated. That's because only the driver knows the intimate details of which ports can be configured how. It also lets you easily add other drivers, without having to add knowledge about the characteristics of the new driver.

Once validation is done, the driver becomes a resource manager and waits for requests.

Code walkthrough

We'll walk through the following modules:

- *main()*, option processing, and initialization
- the resource manager modules
- the card interface modules.

main() and friends

The *main()* function is typically short; it does the option processing and then calls the resource manager mainline. However, there's one important call in *main()* that should be pointed out:

```
ThreadCtl (_NTO_TCTL_IO, 0);
```

This function allows a `root`-owned process (or one that's *setuid()* to `root`) to access the hardware I/O ports. If you don't call this function, and attempt to do I/O port manipulation (via *in8()* for example), your process dies with a `SIGSEGV`.

Option processing is fairly standard as well, except that at the end of option processing we read in the configuration file. The configuration file drives the card installation (I've deleted some of the long error messages for clarity and brevity):

```
parser_t        *p;
parser_handle_t *ph;

...

if (ph = parser_open (optc)) {
  if (p = parser_parse (ph, "/dev/pcl711*")) {
   if (p -> status != PARSER_OK) {
      // error message
      exit (EXIT_FAILURE);
    }
    if (optv) {
      parser_dump (p);
    }
    install_cards_from_database (p);
    parser_free (p);
  }
  parser_close (ph);
} else {
  if (optc) {
    // error message
    exit (EXIT_FAILURE);
  } else {
    // warning message
  }
}
```

Basically, the logic is that we call *parser_open()* to get a parse handle, which we then pass to *parser_parse()*. Notice that *parser_parse()* is given a wild-card pattern of `/dev/pcl711*` to match — this is how we filter out only our card's information from the configuration file. This aspect of the driver names was one of the reasons that I create multiple mount points per driver, rather than just one. Finally, the real work is done in *install_cards_from_database()* (in `pcl711.c`).

Skipping vast amounts of code (the *parser_*()* functions — see the source code) and hiding the details, *install_cards_from_database()* boils down to:

```
int
install_cards_from_database (parser_t *data)
{
  int s;

  s = strlen ("/dev/pcl711-");

  for (nd = 0; nd < data -> ndevices; nd++) {
    card = install_card (d -> devname,
             strtol (d -> devname + s, NULL, 16));
    for (nc = 0; nc < d -> nchannels; nc++) {
      for (np = 0; np < c -> nports; np++) {
        for (naio = 0; naio < p -> naios; naio++) {
          // verify and optionally default configuration
        }
        for (ndio = 0; ndio < p -> ndios; ndio++) {
          // verify and optionally default configuration
        }
      }
    }
  }
}
```

We process all devices in the database (the first `for` loop). This is to handle the case where there are multiple PCL-711 cards installed; this iterates through all of them. The next `for` loop processes all of the channels on each card, and then the next `for` loop processes all of the ports on each channel. Finally, the two innermost `for` loops process the analog and digital points.

Notice that we call *install_card()* within the first `for` loop. This function registers the card name with the process manager via *resmgr_attach()*.

Processing mainly involves range checking. For the PCL-711, the channels are organized like this:

Channel(s)	Meaning
0 - 7	Analog Inputs
8	Analog Output
9	Digital Input (16 bits)
10	Digital Output (16 bits)

There is no concept of a “port” on the PCL-711 card (nor on the ISO-813 card, for that matter), so we don't do any checking for ports in that logic.

An easy way to remember the ordering is with the name of the server, ADIOS. *A*nalog channels are grouped together first, followed by *d*igital channels. Within each group, the *i*nputs are first, followed by the *o*utputs.

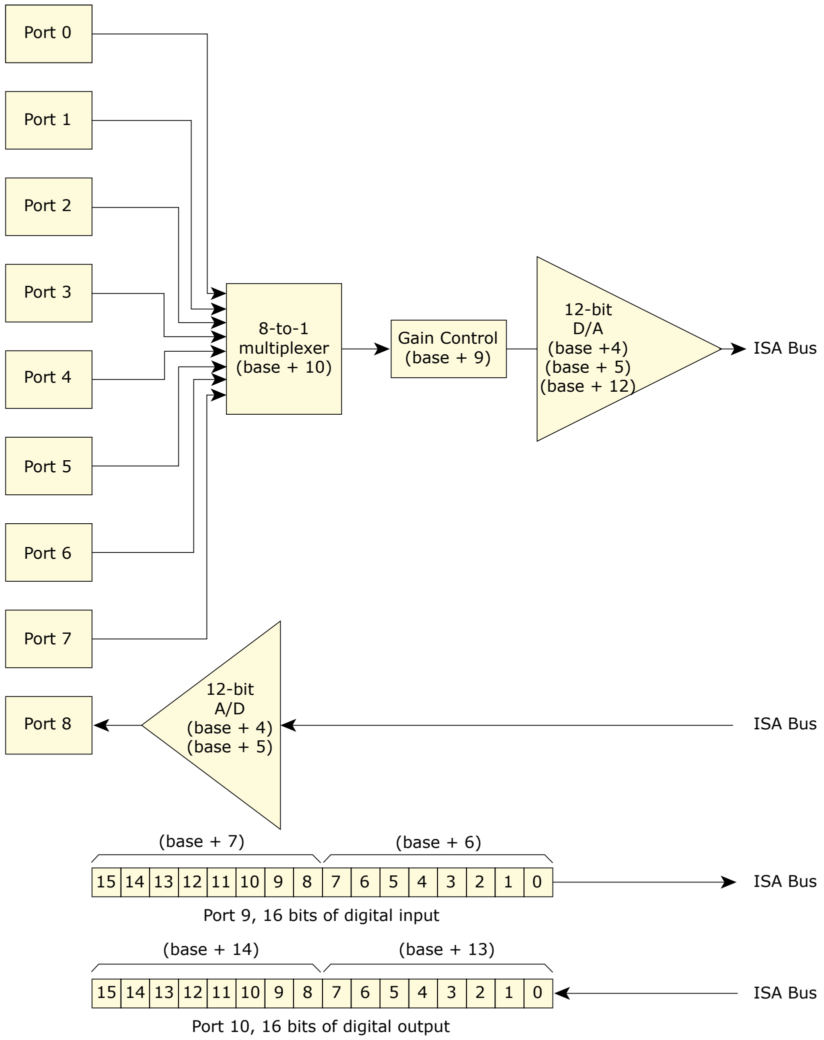


Figure 19: The PCL-711 card, with register offsets.

For reference, here are the channel assignments for the other two cards (DIO-144 and ISO-813), starting with the DIO-144:

Channel(s)	Port(s)	Meaning
0 - 5	A, B, or C	Digital I/O

On the DIO-144, ports A and B can be configured as input or output for the entire 8-bit port, whereas port C can be configured on a nybble (4-bit) basis (the upper or lower 4 bits have the same direction within each nybble, but the nybbles are configured independently). To obtain the base address for any given channel and port, multiply the channel number by 4 and add it to the base address plus the port offset.

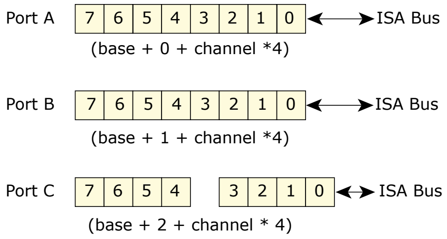


Figure 20: The DIO-144 card with register offsets.

On the ISO-813, the assignments are as follows:

Channel(s)	Meaning
0–31	Analog input

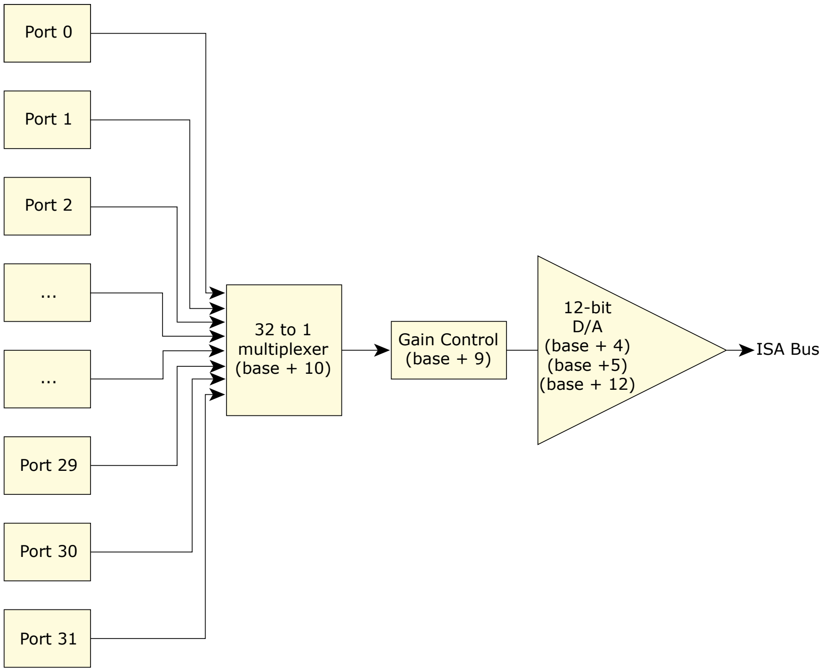


Figure 21: The ISO-813 card high-level view with register offsets.

When *install_cards_from_database()* returns, we have already registered the pathnames with the process manager (via *install_card()*) and configured it (again via *install_card()*). All that's left to do is to enter the resource manager main loop, *execute_resmgr()*.

Copyright © 2014, QNX Software Systems Limited

There's nothing special or different in *execute_resmgr()* that we haven't seen before in the other examples (like the web counter presented in the Web Counter Resource Manager chapter) so we won't discuss it.

What I will quickly discuss is the extended attributes structure:

```
typedef struct pcl711_s
{
    iofunc_attr_t   attr;
    int             port;
    char            *name;
    int             dout;
    int             gains [NAI];
}   pcl711_t;
```

As usual, the normal attributes structure is the first member of the extended attributes structure. Here are the other members:

port

I/O port that this device is based at.

name

ASCII name of the device (e.g. the string "`/dev/pcl711-0200`").

dout

Shadow copy of the digital output for this card — see "Digital output," below.

gains

This array stores the gain values (in card-native format) for each channel.

We'll see how most of these fields are used when we discuss the code for the analog and digital I/O functions.

The resource manager modules

Once the option processor has finished, all of the cards are initialized, and we've entered the *execute_resmgr()* main processing loop. From that point on, we are a resource manager, and we're waiting for requests. The driver doesn't do anything on its own; it's entirely client-driven.

In `iofuncs.c` you see the one and only callout that we're providing, namely the *io_devctl()* handler.

The *io_devctl()* handler is responsible for the following commands:

`DCMD_GET_CONFIG`

Returns the number of analog and digital I/O points, the analog I/O resolution (number of bits), and the number of bytes per digital channel. The data for the configuration comes from the constants in the include file `pcl711.h`.

DCMD_GET_ADIS

Read the analog and digital inputs and return them to the client. This is the main command that's used to get data out of the resource manager. (ADIS stands for *A*nalog/*D*igital *I*nput*S*.) The data originates from the card interface functions *pcl711_read_analog()* and *pcl711_read_digital()* in `pcl711.c`.

DCMD_SET_CPAO

Writes one or more analog outputs. This is the command that clients use to write analog data. (CPAO is *C*hannel and *P*ort *A*nalog *O*utput.) The data is handled by the card interface function *pcl711_write_analog()* in `pcl711.c`.

DCMD_SET_CPBDO

Writes one or more digital output *bits* (not nybbles nor bytes). This is the command that clients use to write digital data. (CPBDO means *C*hannel and *P*ort *B*it *D*igital *O*utput). The data is handled by the card interface function *pcl711_write_digital_bit()* in `pcl711.c`

The other drivers (DIO-144 and ISO-813) are responsible for the same commands (and use similarly named card interface functions), and return `EINVAL` for any commands that aren't appropriate.

So as far as the resource manager interface goes, it's very simple. The real work gets done in the individual interface functions in `pcl711.c` (and `dio144.c` and `iso813.c` for the other cards).

The card interface modules

Finally, we'll look at the hardware interface.

Card configuration

As mentioned above, card configuration happens with the *install_card()* function. You'll note that there's a special debug flag, -d, that puts the driver into diagnostic mode (for the first detected card) and causes it to never become a resource manager.

Analog input

The *pcl711_read_analog()* function is used to handle the analog input:

```
#define PCL711_DELAY    1000  // 1 us

int pcl711_read_analog (pcl711_t *pcl, int channel)
{
  int   data, base, timeout;
  static int calibrated = 0;
  static struct timespec when;

  // 1) calibrate nanospin if required
  if (!calibrated) {
    nanospin_calibrate (1);      // with interrupts off
    nsec2timespec (&when, PCL711_DELAY);
    calibrated = 1;
  }

  // 2) ensure we are in range
  channel &= 7;
  base = pcl -> port;

  // 3) select the channel
  out8 (base + PCL711_MUX_SCAN_CONTROL, channel);

  // 4) select the gain
  out8 (base + PCL711_GAIN_CONTROL, pcl -> gains [channel]);

  // 5) trigger the conversion
  out8 (base + PCL711_SOFTWARE_AD_TRIGGER, 0 /* any data */);

  // 6) wait for the conversion
  timeout = PCL711_TIMEOUT;
  do {
    data = in8 (base + PCL711_ANALOG_HIGH);
    nanospin (&when);     // spin
  } while ((data & PCL711_ANALOG_HIGH_DRDY) && (timeout-- >= 0));

  // 7) indicate timeout if any
  if (timeout < 0) {
    return (-1);
  }

  // 8) return data
  data = ((data & 0x0f) << 8) + in8 (base + PCL711_ANALOG_LOW);
  return (data);
}
```

The code performs the following steps:

1. If we haven't already done so (in a previous invocation) calibrate the *nanospin()* values, and set up a time delay (`PCL711_DELAY` is 1000, or 1 microsecond).
2. The channel number is clamped to the range of 0 through 7. This is a sanity enforcement within the code. We get the I/O port base address from the `pcl711_t` structure (see the section after step 8, below).
3. We write the desired channel number into the multiplexer control register. This doesn't start the conversion yet, but selects the input channel only.
4. We write the desired gain to the gain control register. Notice how the gains are stored in the extended attributes structure, just as the base register was stored there and used in step 3 above.
5. When we write any data to the software A/D trigger register, the PCL-711 card begins the data conversion. We've already gated the source into the gain amplifier, and the converter takes the output of the gain amplifier and converts it. The manual says that this operation takes on the order of microseconds.
6. Here we *poll* for the conversion to be complete (see the section after step 8, below). When the data is ready, the `PCL711_ANALOG_HIGH_DRDY` bit will go low, and we'll exit the loop (we also exit on timeout).
7. If there was a timeout, we return the special value of -1 to whoever called us.
8. Finally, our data is ready. We use the lowest four bits of the data register we used for polling (these four bits end up being bits 8 through 11 of the result) and we add that to the least-significant part of the value (from the `PCL711_ANALOG_LOW` register).

Notice that the first parameter, *pcl*, is of type pointer to `pcl711_t`. The `pcl711_t` is the extended attributes structure used in the resource manager. It's a convenient place to store additional information, such as the base port address.

Notice that in step 6 we are polling. While polling is generally frowned upon in realtime control systems, we have no choice. The manual states that the conversion will take place within microseconds, so the overhead of giving up the CPU and letting another thread run is going to be the same as, or greater than, the time it takes to poll and get the data. So we might as well poll. Note also that we don't poll forever; we poll only for `PCL711_TIMEOUT` number of iterations.

The polling is in place to handle hardware that's not present — if the hardware is missing (or defective), we will time out. Also, notice that we use *nanospin()* to give up the ISA bus between polling. The *nanospin()* delay value is selected to be 1 microsecond; this ensures that we poll “often enough” to minimize the number of times we poll.

In conjunction with the *pcl711_read_analog()* function, there's a function that sets the gain value. While we could have written the gain value directly into the extended attributes structure's *gain* member, it's much nicer to have a function to do it, so that we can isolate accesses to that parameter (and change things around if we need to, without changing a bunch of code).

This function is *pcl711_set_gain()*:

```
void
pcl711_set_gain (pcl711_t *pcl, int channel, int gaincode)
{
  if (gaincode < 0 || gaincode > 4) {
    return;
  }
  channel &= 7;
  pcl -> gains [channel] = gaincode;
}
```

Notice the sanity checking up front to ensure that no bad values are used before we write the value into the *gains* array member.

Analog output

For analog output, the function *pcl711_write_analog()* is used:

```
void
pcl711_write_analog (pcl711_t *pcl, int value)
{
  out8 (pcl -> port + PCL711_ANALOG_LOW, value & 0xff);
  out8 (pcl -> port + PCL711_ANALOG_HIGH, (value & 0xf00) >> 8);
}
```

This function simply writes the low byte of the value to the register `PCL711_ANALOG_LOW` and then writes the high byte (actually, bits 8 through 11) of the value to `PCL711_ANALOG_HIGH`.

Order is important here! The PCL-711 is an 8-bit ISA card, which means that the I/O ports are only eight bits, so they must be written individually. The D/A conversion is triggered *after* the `HIGH` portion of the value is written, so if we

Copyright © 2014, QNX Software Systems Limited

wrote the data in the opposite order, we'd be triggering a conversion with the correct HIGH value, but the previous LOW value.

Just something to watch out for.

Digital input

Digital input is accomplished by:

```
static int bits [16] = {
  0x0001, 0x0002, 0x0004, 0x0008,
  0x0010, 0x0020, 0x0040, 0x0080,
  0x0100, 0x0200, 0x0400, 0x0800,
  0x1000, 0x2000, 0x4000, 0x8000
};

int
pcl711_read_digital_bit (pcl711_t *pcl, int bitnum)
{
  bitnum &= 15;       // guarantee range

  if (bitnum < 8) {
    return (!!(in8 (pcl -> port + PCL711_DI_LOW) & bits [bitnum]));
  } else {
    return (!!(in8 (pcl -> port + PCL711_DI_HIGH) & bits [bitnum - 8]));
  }
}
```

This function determines if the bit that's to be read is in the LOW or HIGH register, and then reads it. The read bit is then logically ANDed against the *bits* array to isolate the bit, and then the special Boolean typecast operator (“!!”) that I invented a few years back is used to convert a zero or nonzero value to a zero or a one.

The !! technique is something I discovered a few years back. It's 100% legal ANSI C and, more importantly, guaranteed to work. And it made my editor think there was a feature of C that he'd missed! :-)

Digital output

Finally, digital output is accomplished by:

```
void
pcl711_write_digital_bit (pcl711_t *pcl, int bitnum, int bitval)
{
  bitnum &= 15;       // guarantee range

  if (bitval) {
    pcl -> dout |= bits [bitnum];
  } else {
    pcl -> dout &= ~bits [bitnum];
  }

  if (bitnum < 8) {     // 0 .. 7 are in the first byte
    out8 (pcl -> port + PCL711_DO_LOW, pcl -> dout & 0xff);
  } else {
    out8 (pcl -> port + PCL711_DO_HIGH, pcl -> dout >> 8);
  }
}
```

Digital output is a little bit trickier, because the hardware wants to be written to one byte at a time, rather than one bit at a time. We manage this by maintaining a shadow, called *dout*, in the extended attributes structure. This shadow contains the currently written value of the digital output port. When we wish to set or clear a particular bit, the first `if` in the function updates the shadow register to reflect the about-to-be-written value. Next, we determine whether the bit is in the `HIGH` or `LOW` portion of the word, then write out the entire 8-bit byte to the port.

Copyright © 2014, QNX Software Systems Limited

The ADIOS server code

Now that we've seen all of the work that goes into the low-level driver, it's time to look at the ADIOS server that uses these drivers.

At the highest layer of abstraction, ADIOS creates a shared memory segment and periodically polls all of the installed drivers for their analog and digital data before sticking it into shared memory.

ADIOS has two threads: one thread handles the resource manager side of things, and another thread handles polling the individual drivers. This approach was taken because I didn't want the polling thread to bog down the responsiveness of the resource manager. However, this is a moot point, because the resource manager currently doesn't actually *do* anything. I provided a resource manager interface for future compatibility in case we needed a way of changing the sampling rate (for example), but there haven't been any requirements to do that.

The usual stuff

There's nothing to see in *main()*, *optproc()*, and *execute_resmgr()* that you haven't seen before, with the possible exception of the *pthread_create()* in *main()* to create the worker thread, *daq_thread()*:

```
pthread_create (NULL, NULL, daq_thread, NULL);
```

Even that is a plain vanilla call to *pthread_create()*.

For every card that's specified in the configuration file, *optproc()* calls the worker function *install_cards_from_database()* to create a database (stored in the global variable *adios*). Part of the work of installing the card is to send its driver a *devctl()* asking it about its capabilities. You'll recall from above that this is the DCMD_GET_CONFIG message.

The shared memory region

Finally, the main thread sets up a shared memory region.

If we find that there's already a shared memory region present, we invalidate its signature block and unlink it. The rationale here is that a previous ADIOS manager created one, and was killed (or died). By invalidating the signature, we're telling any users of that shared memory region that it's no longer valid.

Creating the shared memory region (and setting up various pointers and data areas) is done by the function *create_shmem()* in daq.c. Since it's a fairly large function (about 100 lines), I'll present the steps it performs first, and then we'll look at the details in smaller chunks:

Copyright © 2014, QNX Software Systems Limited

1. Calculate sizes of various data structures.
2. Open the shared memory via *shm_open()*.
3. If it already exists, invalidate its signature, close it, unlink it, and attempt to reopen it.
4. If that second open fails, we abort.
5. Truncate the shared memory region to the correct (calculated) size.
6. Using *mmap()*, map the shared memory region into our address space.
7. Set up our utility pointers.
8. Clear the signature, and initialize the other fields in the signature structure.
9. Set the *head* member to point to the last entry in the buffer, so that when we add our first entry, *head* will point to 0.
10. Fill the *card information structure* (CIS) for all the cards that we are polling.

Calculating the sizes of data structures

As mentioned above, the shared memory region is divided into two sections, each beginning on a 4k page boundary. The first section contains the `adios_signature_t`, followed by the `adios_daq_status_t`, followed by one `adios_cis_t` for each installed card.

Suppose that we have one PCL-711 card and one DIO-144 installed. This is what the first part of the memory layout will look like:

Offset (bytes)	Name	Size (bytes)	Description	Value
	`adios_signature_t`			
0000	*signature*	4	Signature	`"ADIO"`
0004	*datablock*	4	Datablock; which 4 KB page the data section starts on	1
0008	*num_cis*	4	Number of entries in the CIS	2
000C	*num_elems*	4	Size of the ring buffer	2000
	`adios_daq_status_t`			
0010	*head*	4	Index to the newest valid and stable data element	0
0014	*tail*	4	Index to the oldest valid and stable data element	99
0018	*element_size*	4	Size of each element, including any padding	52
	`adios_cis_t`			

Copyright © 2014, QNX Software Systems Limited

Offset (bytes)	Name	Size (bytes)	Description	Value
001C	*name*	128	Name of the device	`/dev/pcl711-02d0`
009C	*nai*	4	Number of analog inputs	8
00A0	*nao*	4	Number of analog outputs	1
00A4	*ndi*	4	Number of digital inputs	16
00A8	*ndo*	4	Number of digital outputs	16
00AC	*nbpc*	4	Number of bytes per channel	2
00B0	*maxresai*	4	Maximum bit resolution of analog input	12
	`adios_cis_t`			
00B4	*name*	128	Name of the device	`/dev/dio144-0220`
0134	*nai*	4	Number of analog inputs	0
0138	*nao*	4	Number of analog outputs	0
013C	*ndi*	4	Number of digital inputs	144
0140	*ndo*	4	Number of digital outputs	144
0144	*nbpc*	4	Number of bytes per channel	3
0148	*maxresai*	4	Maximum bit resolution of analog input	0
014C–0FFF			Filler	

The second part of the shared memory contains the data. Each data set consists of a tiny `adios_data_header_t` followed by the samples from the various cards. There are as many data sets as specified with the command line -S option (or the default of 1000).

Continuing with our example of the two cards from above, here's what the first and part of the second data set look like:

Offset (bytes)	Name	Size (bytes)	Description
	`adios_data_header_t`		
1000	*t0ns*	8	Beginning of snapshot 0 time
1008	*t1ns*	8	End of snapshot 0 time
	(Data)		
1010	*ai0*	2	PCL-711 analog input channel 0 sample

Copyright © 2014, QNX Software Systems Limited

Offset (bytes)	Name	Size (bytes)	Description
1012	*ai1*	2	PCL-711 analog input channel 1 sample
1014	*ai2*	2	PCL-711 analog input channel 2 sample
1016	*ai3*	2	PCL-711 analog input channel 3 sample
1018	*ai4*	2	PCL-711 analog input channel 4 sample
101A	*ai5*	2	PCL-711 analog input channel 5 sample
101C	*ai6*	2	PCL-711 analog input channel 6 sample
101E	*ai7*	2	PCL-711 analog input channel 7 sample
1020	*di*	2	PCL-711 digital input channel 8 (16 bits)
1022	*di*	18	DIO-144 digital input channel 0–5 samples
	`adios_data_header_t`		
1034	*t0ns*	8	Beginning of snapshot 1 time
103C	*t1ns*	8	End of snapshot 1 time
	(Data)		
1044	*ai0*	2	PCL-711 analog input channel 0 sample
1046	*ai1*	2	PCL-711 analog input channel 1 sample
...	...		

Therefore, the first job of *create_shmem()* is to figure out the sizes of the various data structures.

```
void
create_shmem (void)
{
  int   size;
  int   size_c, size_d;
  int   size_d_ai, size_d_di;
  int   size_element;
  int   i;
  int   sts;

  // size_c is the control section size
  size_c = sizeof (adios_signature_t) + sizeof (adios_cis_t) * nadios;

  // size_d is the data section size
  size_d = sizeof (adios_data_header_t);
  for (i = 0; i < nadios; i++) {
    size_d_ai = adios [i].nai * ((adios [i].maxresai + 15) / 16) * 2;
    size_d_di = (adios [i].ndi + 31) / 32 * 4;
    size_d += size_d_ai + FILLER_ALIGN_32bits (size_d_ai) + size_d_di;
  }
  size_element = size_d;
  size_d *= optS;

  // compute the total size of shared memory
  size = size_c + FILLER_ALIGN_4kbytes (size_c)
```

Copyright © 2014, QNX Software Systems Limited

```
       + size_d + FILLER_ALIGN_4kbytes (size_d);
...
```

As you can see, the code runs through the *adios* global database that it filled in as the final phase of option processing, and accumulates the total data sizes that all of the samples will need. The *optS* variable is the number of samples; once we know each sample's size we multiply by that number to compute the total size. The total size of the shared memory is *size_c* (rounded up to be a multiple of 4096 bytes) plus *size_d* (also rounded up).

Open and check the shared memory

The next step in processing is to open the shared memory region. If one already exists, we invalidate it and remove it, and then try to open it again (error messages in code not shown):

```
...

  sts = shm_open (opts, O_RDWR | O_CREAT | O_EXCL, 0644);
  if (sts == -1) {
    if (errno == EEXIST) {

      // in the case that it already exists, we'll wipe the
      // signature of the existing one and unlink it.
      sts = shm_open (opts, O_RDWR, 0);
      if (sts == -1) {
        exit (EXIT_FAILURE);
      }

      // map it in so we can wipe it
      shmem_ptr = mmap (0, 4096, PROT_READ | PROT_WRITE,
                        MAP_SHARED, sts, 0);
      if (shmem_ptr == MAP_FAILED) {
        exit (EXIT_FAILURE);
      }

      // wipe it
      memset (shmem_ptr, 0, 4096);
      close (sts);
      munmap (shmem_ptr, 4096);
      shm_unlink (opts);

      // now try again to open it!
      sts = shm_open (opts, O_RDWR | O_CREAT | O_EXCL, 0644);
      if (sts == -1) {
        exit (EXIT_FAILURE);
      }
    } else {
      // if the initial open failed for any reason
      // *other than* EEXIST, die.
      exit (EXIT_FAILURE);
    }
  }
  shmem_fd = sts;
...
```

The first call to *shm_open()* uses the `O_CREAT` and `O_EXCL` flags. These indicate that the shared memory region is being created, and must not already exist. Notice that the work involved in wiping the signature is the same work that we must do later to map the shared memory region into our address space.

At this point, we now have *shmem_fd* as the shared memory file descriptor.

Truncate and map shared memory

Finally, we need to set the size of the shared memory segment:

```
...

  sts = ftruncate (shmem_fd, size);
  // error check code omitted
...
```

And map it into our address space via *mmap()*:

```
...

  shmem_ptr = mmap (0, size, PROT_READ | PROT_WRITE,
                    MAP_SHARED, shmem_fd, 0);
  // error checking code omitted
...
```

The flags to *mmap()* are:

PROT_READ

Let us read from the shared memory region.

PROT_WRITE

Let us write into the shared memory region.

MAP_SHARED

We are mapping an existing object (the one given by *shmem_fd*), or we anticipate sharing the object.

Now that our shared memory region exists and is the correct size, we assign some utility pointers into the areas within the shared memory:

```
...

  // set up our utility pointers
  sig = (adios_signature_t *) shmem_ptr;
  daq = (adios_daq_status_t *) (shmem_ptr + sizeof (*sig));
  cis = (adios_cis_t *) (shmem_ptr + sizeof (*sig)
        + sizeof (*daq));
...
```

Note that inside the shared memory region we *never* store any pointer values. After all, the shared memory region could be mapped into a different address space within each process that needs to access it. In the code snippet above, we create pointers into the areas of interest *after* we know where the shared memory region is mapped in our address space.

Then we fill the shared memory structure:

```
...

  // clear the signature (just for safety, a
  // new shmem region is zeroed anyway)
  memset (sig -> signature, 0, sizeof (sig -> signature));
  sig -> datablock = size_c + FILLER_ALIGN_4kbytes (size_c);
```

Copyright © 2014, QNX Software Systems Limited

```
  sig -> datablock /= 4096;  // convert to blocks
  sig -> num_cis = nadios;
  sig -> num_elems = optS;
  database = shmem_ptr + sig -> datablock * 4096;

  daq -> element_size = size_element;

  // head points to the last entry in the buffer, so that
  // when we add our first entry head will point to 0
  daq -> tail = daq -> head = optS - 1;

  for (i = 0; i < nadios; i++) {
    strncpy (cis [i].name, adios [i].name, MAXNAME);
    cis [i].nai = adios [i].nai;
    cis [i].nao = adios [i].nao;
    cis [i].ndi = adios [i].ndi;
    cis [i].ndo = adios [i].ndo;
    cis [i].nbpc = adios [i].nbpc;
    cis [i].maxresai = adios [i].maxresai;
  }
}
```

Filling the shared memory structure means that we set:

datablock

Indicates which 4k block the data block starts on.

num_cis

The number of card information structures (CISs) that are present.

num_elems

The number of data elements (sets) that are in the shared memory ring buffer.

element_size

The size of each element within the data set.

head* and *tail

Indexes into the data elements to indicate head and tail of the ring buffer.

nai*, *nao*, *ndi*, *ndo

The number of AI, AO, DI, and DO points (note that analog points, AI and AO, are given as the number of channels, and digital points, DI and DO, are given as the number of bits).

nbpc

The number of bytes per channel. This comes in handy when retrieving data from the ring buffer; you'll see it again in the `tag` utility description.

maxresai

Number of bits of analog input resolution (12 for all of the card types mentioned in this chapter).

Notice that we did *not* fill in the signature field (we only zeroed it). That's because we validate the signature only when we have at least one valid sample in the shared memory ring buffer.

That was a lot of work. Compared to this, managing the shared memory ring buffer data is much simpler!

Acquiring data

Now that the shared memory ring buffer is set up, we can enter a polling loop and acquire data from the various devices.

This involves the following steps:

- Allocate a buffer for the transfer from the individual drivers.
- Loop forever:
  - Adjust the *tail* index to make room.
  - Take a snapshot of the clock before and after the sample set.
  - Using DCMD_GET_ADIS, issue a *devctl()* to get the data from each driver.
  - Copy the data into shared memory.
  - Adjust the *head* pointer to reflect the new valid sample set.
  - If we haven't signed the signature area, do so now.
  - Delay until the next polling interval.

Notice that we sign the signature if we haven't done so yet. This ensures that there is at least one valid sample before we declare the shared memory area okay to use.

Here are the pieces of the DAQ thread that do the above steps (error messages in code not shown):

```
daq_thread (void *not_used)
{
  ...
  // calculate the *maximum* transfer size
  ai = di = 0;
  for (i = 0; i < nadios; i++) {
    if (adios [i].ndi > di) {
      di = adios [i].ndi;
    }
    if (adios [i].nai > ai) {
      ai = adios [i].nai;
    }
  }

  // allocate a buffer which we never free
  xfersize = ai * 2 + di / 8;

  c = malloc (sizeof (*c) + xfersize);
  if (c == NULL) {
    // trash the process; no real use in continuing
    // without this thread
    exit (EXIT_FAILURE);
  }
...
```

We have the *adios* array in memory, and that tells us the number of analog and digital input points. We use this in our calculation to come up with a transfer size (*xfersize*)

Copyright © 2014, QNX Software Systems Limited

that represents the maximum transfer size. The transfer size calculated may be much bigger than actually required, because we've summed *all* of the data I/O points, rather than trying to figure out the biggest transfer size required per card. In smaller systems, this won't be a problem because we're talking about only a few hundred bytes. In a large system, you may wish to revisit this piece of code and calculate a more appropriate transfer size. The *xfersize* variable gets overwritten later, so it's safe to modify it in the above code.

Within the loop, we perform some *head* and *tail* manipulations in a local copy. That's because we don't want to move the *head* and *tail* pointers in shared memory until the data set is in place. In case the shared memory region is full (as it will be once it initially fills up — it never actually “drains”), we do need to adjust the *tail* pointer in the shared memory region as soon as possible, so we do so immediately.

```
...

  // now loop forever, acquiring samples
  while (1) {
    // do one sample

    // adjust the head in a local working copy
    // while we futz with the data
    head = daq -> head + 1;
    if (head >= optS) {
      head = 0;
    }

    // adjust the tail in an atomic manner
    // so that it's always valid
    if (daq -> tail == head) {
      tail = daq -> tail + 1;
      if (tail >= optS) {
        tail = 0;
      }
      daq -> tail = tail;
    }
...
```

Notice how the *daq -> tail* member is adjusted after we calculate the correct version of *tail* in a local copy. This is done in order to present an atomic update of the *tail* index. Otherwise, we'd have a potentially out-of-range value of *tail* in shared memory after we incremented it, and before we looped it back around to zero.

There's another window of failure here. Theoretically, the client of the shared memory interface and ADIOS should maintain a mutex (or semaphore) to control access to the shared memory. That's because it's possible that, if the client requires the full number of sample sets (i.e. the 1000 samples or whatever it's been changed to via -S on the command line), ADIOS could be in the middle of writing out the new data sample set over top of the oldest data set.

I thought about this, and decided *not* to incur the additional complexity of a synchronization object, and instead informed the end-users of ADIOS that they should make their number of samples bigger than the number of samples they actually require from the shared memory region. While this may appear to be somewhat *tacky*, in reality it's not that bad owing to the speed at which things happen. In normal operation, the customer needs something like a few hundred samples at most, and these samples are updated at a rate of ten per second. So by extending the number of samples to be much bigger than a few hundred, it would take a significant amount of time (tens of seconds) before the oldest data set reached this about-to-be-overwritten state.

```
...

    // get the data
    ptr = (void *) (database + head * daq -> element_size);
    dhdr = ptr;
    ClockTime (CLOCK_REALTIME, NULL, &dhdr -> t0ns);
    ptr = dhdr + 1;

...

    /*
     * Here we get the data; I've moved this code into the next
     * para so we can see just the clock manipulation here.
    */

...

    ClockTime (CLOCK_REALTIME, NULL, &dhdr -> t1ns);

    // finally, set the daq -> head to our "working" head now
    // that the data is stable
    daq -> head = head;
...
```

The code above illustrates the outer time snapshot and update of the *head* index. Between the two time snapshots, we acquire the data (see the code below). The point of doing the two *ClockTime()* snapshots was for performance measuring, statistics, and sanity.

The *ClockTime()* function gives us the number of nanoseconds since the beginning of time (well, QNX Neutrino's concept of the “beginning of time” anyway). The difference in the value of the members *t0ns* and *t1ns* is the amount of time it took to acquire the samples, and *t0ns* can also be used to determine when the sample acquisition started. This data is stored with each sample set. The performance measurement aspect of this should be obvious — we just determine how long it takes to acquire the samples. The statistics and freshness aspects of this are based on the customer's requirement. They need to know exactly when each sample was taken, so that they can plug these numbers into their proprietary process control formula.

```
...

    // code removed from section above:
    for (i = 0; i < nadios; i++) {
      c -> i.nais = adios [i].nai;
      c -> i.ndis = adios [i].ndi;
      xfersize = c -> i.nais * 2 + c -> i.ndis / 8;
      sts = devctl (adios [i].fd, DCMD_GET_ADIS,
                    c, xfersize, NULL);
      if (sts != EOK) {
        // code prints an error here...
        exit (EXIT_FAILURE);
      }
      // just memcpy the data from there to shmem
      memcpy (ptr, c -> o.buf, xfersize);
      ptr = (void *) ((char *) ptr + xfersize);
    }
...
```

Copyright © 2014, QNX Software Systems Limited

Above is the code that went between the two time snapshots. As you can see, we run through our *adios* array database, calculate an appropriate transfer size for each particular transfer (but still using the bigger transfer buffer we allocated above). The transfer is accomplished by a *devctl()* to each driver with the DCMD_GET_ADIS command. This command returns a packed analog input and digital input array, which we simply *memcpy()* into shared memory at the correct place. (We do pointer math with *ptr* to get it to walk along the data set.)

```
...

    /*
     *  See if we need to sign the data area.  We do this only
     *  after at least one sample has been put into shmem.
     */

    if (!memory_signed) {
      memcpy (sig -> signature, ADIOS_SIGNATURE,
              sizeof (sig -> signature));
      memory_signed = 1;
    }

    // wait for the next sample time to occur
    delay (optp);
  }
}
```

Finally, we sign the shared memory region if we haven't already (to ensure that there's at least one sample before the signature is valid), and we delay until the next sample time.

You may have noticed that we aren't going to be acquiring samples at *exactly* 10 Hz (or whatever rate the user specifies on the command line). That's because the amount of time spent in accumulating the samples *adds to* the total delay time. This was not a concern for the customer, and code with a fixed delay (and with slippage) is much easier to implement than code that runs at a fixed period.

If we did want the code to run at a fixed period, then there are a couple of ways to do that:

- We could replace the *delay()* with a semaphore wait (*sem_wait()*), and then have a separate thread that hits the semaphore at a fixed interval, or
- We could replace the *delay()* with a *MsgReceive()* and wait for a pulse from a periodic timer that we would set up earlier.

Both approaches are almost identical as far as timing is concerned. The semaphore approach may also suffer from “lag” if the thread that's hitting the semaphore gets preempted. Both approaches may suffer from the inability to keep up on a large system, or if a higher sample rate were used. Because enabling the next sample (i.e. the *sem_wait()* or the *MsgReceive()*) is asynchronously updated with respect to the data acquisition, it's possible that if the data acquisition takes longer than the period, things will start to “back up.”

This too can be solved, by draining all events (semaphore or pulses) before continuing to the next sample set. If you count the number of events you've drained, you can get a good idea of how far behind you are lagging, and that can be output to an operator or a log for diagnostics.

Copyright © 2014, QNX Software Systems Limited

The showsamp and tag utilities

Now that we have a good understanding of where the data comes from (the pcl711, iso813, and the dio144 drivers and their implementation) and how the data gets put into shared memory (the adios server), let's take a quick look at two additional utilities, namely showsamp and tag.

The showsamp utility was written first, and was mainly a debugging aid to determine if the samples being stored by ADIOS were correct and accessible. Then tag was written to exercise the configuration files and the tag accessibility. So, as with most projects, two of the useful utilities weren't in the initial requirements, and were actually written as simple debug programs.

The showsamp utility

Before we dive into the code for showsamp, here is an example of what the output looks like. I'm running this with a simulated device, called /dev/sim000-0220, which generates patterns of data.

```
# showsamp

SHOWSAMP
Shared memory region:  /adios
Size:          40960 bytes
Devices: (1)
    [0] "/dev/sim000-0220" 8 x AI, 16 x DI
HEAD 206 TAIL 999 ELEMENT_SIZE 36 NUM_ELEMS 1000
Data Set 202, acquired 2003 10 09 14:25:50.73824624 (delta 0 ns)
CARD /dev/sim000-0220, 8 AI, (1 AO), 16 DI, (16 DO)
  AI: 0651 0652 0653 0654 0655 0656 0657 0658
  DI: 00CB

Data Set 203, acquired 2003 10 09 14:25:50.83914604 (delta 0 ns)
CARD /dev/sim000-0220, 8 AI, (1 AO), 16 DI, (16 DO)
  AI: 0659 065A 065B 065C 065D 065E 065F 0660
  DI: 00CC

Data Set 204, acquired 2003 10 09 14:25:50.94004585 (delta 0 ns)
CARD /dev/sim000-0220, 8 AI, (1 AO), 16 DI, (16 DO)
  AI: 0661 0662 0663 0664 0665 0666 0667 0668
  DI: 00CD

Data Set 205, acquired 2003 10 09 14:25:51.04096166 (delta 0 ns)
CARD /dev/sim000-0220, 8 AI, (1 AO), 16 DI, (16 DO)
  AI: 0669 066A 066B 066C 066D 066E 066F 0670
  DI: 00CE

Data Set 206, acquired 2003 10 09 14:25:51.14186147 (delta 0 ns)
CARD /dev/sim000-0220, 8 AI, (1 AO), 16 DI, (16 DO)
  AI: 0671 0672 0673 0674 0675 0676 0677 0678
  DI: 00CF
```

The showsamp utility starts up and tells me that it's using a shared memory region identified as /adios, that it's 40k bytes in size, and that there is only one device present, called /dev/sim000-0220. It then tells me that this device has 8 analog inputs and 16 digital inputs and dumps the default 5 samples, showing the analog and the digital values.

The showsamp utility has a rather large *main()* function. That's because, as stated above, it was written initially to be a quick hack to test ADIOS's shared memory. Basically, it opens the shared memory segment, calls *mmap()* to map the shared memory segment to a pointer, and then verifies the signature.

If this all succeeds, then *main()* creates a series of pointers into the various data structures of interest, and dumps out the last N samples (where N defaults to 5 and can be overridden on the command line with -n):

```
for (i = head - optn + 1; i <= head; i++) {
  if (i < 0) {
    do_data_set (i + sig -> num_elems);
  } else {
    do_data_set (i);
  }
}
```

The code above starts at N positions before the current *head* and goes until the *head*. Notice that the result of the subtraction may be negative, and that's fine — we handle it by adding in the number of elements from the *num_elems* member in shared memory.

To actually get one data set's worth of samples:

```
static void
do_data_set (int ds)
{
  int                   i, j;
  adios_data_header_t   *dhead;
  uint16_t              *u16;
  struct tm             *tm;
  char                  buf [BUFSIZ];

  dhead = (adios_data_header_t *) (database
        + ds * daq -> element_size);
  i = dhead -> t0ns / 1000000000LL;
  tm = localtime (&i);
  strftime (buf, sizeof (buf), "%Y %m %d %H:%M:%S", tm);

  printf ("Data Set %d, acquired %s.%09lld (delta %lld ns)\n",
          ds, buf, dhead -> t1ns % 1000000000LL,
          dhead -> t1ns - dhead -> t0ns);
  u16 = (uint16_t *) (dhead + 1);
  for (i = 0; i < sig -> num_cis; i++) {
    printf ("CARD %s, %d AI, (%d AO), %d DI, (%d DO)\n",
            cis [i].name,
            cis [i].nai, cis [i].nao,
            cis [i].ndi, cis [i].ndo);
    if (cis [i].nai) {
      printf ("  AI: ");
      for (j = 0; j < cis [i].nai; j++) {
        printf ("%04X ", *u16++);
      }
      printf ("\n");
    }
    if (cis [i].ndi) {
      printf ("  DI: ");
      for (j = 0; j < (cis [i].ndi + 15) / 16; j++) {
        printf ("%04X ", *u16++);
      }
      printf ("\n");
    }
  }
  printf ("\n");
}
```

The *ds* parameter is the data set number, and the first line of code does pointer arithmetic to get the pointer *dhead* to point to the appropriate data set. We adjust and format the time that the sample was acquired, and then dump out all of the data

Copyright © 2014, QNX Software Systems Limited

elements. The CIS tells us how many samples we are expecting. Note that the CIS does *not* tell us the formatting of the digital I/O — whether they are "naturally" presented as 8-bit or 16-bit samples, or how they are grouped (e.g. on the DIO-144 they are grouped as three sets of 8-bit samples). There are other assumptions in this code, like that all analog samples will be 16 bits or less, and that all digital samples will be printed as 16-bit quantities. You can easily change this by adding more information into the CIS and then examining that information here.

The tag utility

Here's an example of a command-line session using the tag utility with the simulator device, sim000:

```
[wintermute@ttyp0] tag -m1000 s_pc10b9=1 s_pc8=259 \
s_pc9b0 s_pc9b1 s_pc9b2
Data Set 217, acquired 2003 10 09 14:32:36.16915388 (delta 0 ns)
Tag "s_pc9b0" 0
Tag "s_pc9b1" 1
Tag "s_pc9b2" 0

Data Set 227, acquired 2003 10 09 14:32:37.17816996 (delta 0 ns)
Tag "s_pc9b0" 0
Tag "s_pc9b1" 0
Tag "s_pc9b2" 1

Data Set 237, acquired 2003 10 09 14:32:38.18866655 (delta 0 ns)
Tag "s_pc9b0" 0
Tag "s_pc9b1" 1
Tag "s_pc9b2" 1

Data Set 247, acquired 2003 10 09 14:32:39.19768263 (delta 0 ns)
Tag "s_pc9b0" 0
Tag "s_pc9b1" 0
Tag "s_pc9b2" 0

Data Set 257, acquired 2003 10 09 14:32:40.20668071 (delta 0 ns)
Tag "s_pc9b0" 0
Tag "s_pc9b1" 1
Tag "s_pc9b2" 0
```

In this example, I've instructed tag to set the tag *s_pc10b9* to the value 1, and the tag *s_pc8* to the value 259 (decimal). Then, via the -m option, I told tag to dump out the values for the three tags *s_pc9b0*, *s_pc9b1*, and *s_pc9b2* every 1000 milliseconds. I killed the tag program via **Ctrl–C** after five samples were printed.

Part of the functionality present in the tag utility is identical to that from showsamp, namely the setup of shared memory, mapping the shared memory to a pointer, and setting up the various utility pointers to point into items of interest in the shared memory.

Basically, tag can do three things:

1. Set a tag to a particular raw or converted value.
2. Display the raw and converted value of a tag once.
3. Display the raw and converted value of a tag repeatedly.

Obviously, the last two are mutually exclusive. Setting a tag is performed within the option processor *optproc()*, while all display activities are deferred until after the shared memory has been set up. This is because in order to set a tag, we don't need the shared memory — we can talk directly to the device driver and get it to perform the work. However, in order to read a tag, we need to have access to the shared memory, because that's where we will be reading the tag from.

To set a tag, the library function *adios_set_tag_raw()* or *adios_set_tag_span()* is called. What I mean by "raw" is that the value that you are writing is not interpreted by the configuration file's optional span keyword — on a card with a 12-bit D/A converter, the raw value would be between 0 and 4095. Contrast that with the span version, where we look at the configuration file and determine, for example, that the analog output point has a range of -5 to +5, so only values in that range would be valid. (They're converted from floating-point to raw values by the library anyway — this is a nicety for the applications programmer.)

To display a tag, the option processor adds the tag into a list of tags. After option processing has finished, the list of tags is finalized (by looking into shared memory and finding the actual offsets into the data set), and displayed. The function *fixup_display_list()* that does this is long and boring. Basically, what it does is match the channel, port, and, in the case of digital I/O points, the bit number, against the data contained in the CIS, and then it determines the offset.

You can see the result of this by looking at the function that actually displays the data, *display_list()*:

```
static int bits [16] = {
  0x0001, 0x0002, 0x0004, 0x0008,
  0x0010, 0x0020, 0x0040, 0x0080,
  0x0100, 0x0200, 0x0400, 0x0800,
  0x1000, 0x2000, 0x4000, 0x8000};

static void
display_list (void)
{
  adios_data_header_t  *dhead;
  int                  ds;
  int                  i;
  char                 buf [BUFSIZ];
  uint16_t             *u16;
  uint8_t              *u8;
  struct tm            *tm;
  double               v;

  ds = daq -> head;
  dhead = (adios_data_header_t *) (database
        + ds * daq -> element_size);

  i = dhead -> t0ns / 1000000000LL;
  tm = localtime (&i);
  strftime (buf, sizeof (buf), "%Y %m %d %H:%M:%S", tm);
  u16 = (uint16_t *) (dhead + 1);   // get ptr to sample set
  u8 = (uint8_t *) u16;             // get 8-bit version too

  printf ("Data Set %d, acquired %s.%09lld (delta %lld ns)\n",
          ds, buf, dhead -> t1ns % 1000000000LL,
          dhead -> t1ns - dhead -> t0ns);

  for (i = 0; i < ntags; i++) {
    if (tags [i].type == 'a') {
      printf ("Tag \"%s\" raw value 0x%04X", tags [i].name,
              u16 [tags [i].offset / 2]);
      if (tags [i].span_low != tags [i].span_high) {
        v = u16 [tags [i].offset / 2];
        v = v / 4095. * (tags [i].span_high - tags [i].span_low)
```

Copyright © 2014, QNX Software Systems Limited

```
          + tags [i].span_low;
        printf (" span-compensated value %s%g",
                v > 0.0 ? "+" : "", v);
      }
      printf ("\n");
    }
    if (tags [i].type == 'd') {
      printf ("Tag \"%s\" %d\n", tags [i].name,
        !!(u8 [tags [i].offset] & bits [tags [i].bitnum]));
    }
  }
  printf ("\n");
}
```

It has similar date and time printing and calculation logic as the showsamp utility. The for loop runs through all of the tags, and determines if they are analog or digital. If the tag is an analog tag, we display the raw value. If the tag has a span range that's not zero, we calculate and display the converted value as well. In the case of a digital value, we simply fetch the appropriate byte, and mask it off with the appropriate bit number.

The fine distinction between displaying the values once and displaying the values repeatedly is handled by *main()*:

```
do {
  display_list ();
  if (optm != -1) {
    delay (optm);
  }
} while (optm != -1);
```

The do-while ensures that the code executes at least once. The delay factor, *optm* (from the command line -m), is used as both a flag (when the value is -1, meaning “display once”) and the actual delay count in milliseconds (if not -1).

References

The following references apply to this chapter.

Header files

- `<hw/inout.h>` — contains the hardware *in*()* and *out*()* functions used to access I/O ports.

Functions

See the following functions in the QNX Neutrino *C Library Reference*:

- *ClockTime()*
- *delay()*
- *in8()*
- *mmap()*
- *mmap_device_io()*
- *munmap()*
- *nanospin()*
- *nanospin_calibrate()*
- *nsec2timespec()*
- *out8()*
- *shm_open()*
- *shm_unlink()*

Copyright © 2014, QNX Software Systems Limited

Chapter 6
RAM-disk Filesystem

This resource manager is something I've been wanting to do for a long time. Since I wrote the first book on QNX Neutrino, I've noticed that a lot of people still ask questions in the various newsgroups about resource managers, such as "How, exactly, do I support symbolic links?" or "How does the *io_rename()* callout work?"

I've been following the newsgroups, and asking questions of my own, and the result of this is the following RAM-disk filesystem manager.

The code isn't necessarily the best in terms of data organization for a RAM-disk filesystem — I'm sure there are various optimizations that can be done to improve speed, cut down on memory, and so on. My goal with this chapter is to answer the detailed implementation questions that have been asked over the last few years. So, consider this a "reference design" for a filesystem resource manager, but don't consider this the best possible design for a RAM disk.

In the *next chapter* (p. 177), I'll present a variation on this theme — a TAR filesystem manager. This lets you `cd` into a `.tar` (or, through the magic of the `zlib` compression library, a `.tar.gz`) file, and perform `ls`, `cp`, and other commands, as if you had gone through the trouble of (optionally uncompressing and) unpacking the `.tar` file into a temporary directory.

In the *Filesystems* appendix, I present background information about filesystem implementation within the resource manager framework. Feel free to read that before, during, or after you read this chapter.

Copyright © 2014, QNX Software Systems Limited

Requirements

The requirements for this project are fairly simple: "Handle all of the messages that a filesystem would handle, and store the data in RAM." That said, let me clarify the functions that we will be looking at here.

Connect functions

The RAM disk supports the following connect functions:

c_link()

Handles symbolic and hard links.

c_mknod()

Makes a directory.

c_mount()

Mounts a RAM disk at a specified mount point.

c_open()

Opens a file (possibly creating it), resolves all symbolic links, and performs permission checks.

c_readlink()

Returns the value of a symbolic link.

c_rename()

Changes the name of a file or directory, or moves a file or directory to a different location within the RAM disk.

c_unlink()

Removes a file or directory.

I/O functions

The RAM disk supports the following I/O functions:

io_read()

Reads a file's contents or returns directory entries.

io_close_ocb()

Copyright © 2014, QNX Software Systems Limited

Closes a file descriptor, and releases the file if it was open but unlinked.

io_devctl()

Handles a few of the standard *devctl()* commands for filesystems.

io_write()

Writes data to a file.

Missing functions

We won't be looking at functions like *io_lseek()*, for example, because the QNX Neutrino default function *iofunc_lseek_default()* does everything that we need.

Other functions are not generally used, or are understood only by a (very) few people (e.g. *io_mmap()*). :-)

Copyright © 2014, QNX Software Systems Limited

Design

Some aspects of the design are apparent from the Filesystems appendix; I'll just note the ones that are different.

The design of the RAM-disk filesystem was done in conjunction with the development, so I'll describe the design in terms of the development path, and then summarize the major architectural features.

The development of the RAM disk started out innocently enough. I implemented the *io_read()* and *io_write()* functions to read from a fixed (internal) file, and the writes went to the bit bucket. The nice thing about the resource manager library is that the worker functions (like *io_read()* and *io_write()*) can be written independently of things like the connect functions (especially *c_open()*). That's because the worker functions base all of their operations on the OCB and the attributes structure, regardless of where these actually come from.

The next functionality I implemented was the internal in-memory directory structure. This let me create files with different names, and test those against the already-existing *io_read()* and *io_write()* functions. Of course, once I had an in-memory directory structure, it wasn't too long before I added the ability to read the directory structure (as a directory) from within *io_read()*. Afterward, I added functionality like a block allocator and filled in the code for the *io_write()* function.

Once that was done, I worked on functions like the *c_open()* in order to get it to search properly through multiple levels of directories, handle things like the `O_EXCL` and `O_TRUNC` flags, and so on. Finally, the rest of the functions fell into place.

The main architectural features are:

- extended attributes structures
- block allocator and memory pool subsystem.

Notice that we didn't need to extend the OCB.

Copyright © 2014, QNX Software Systems Limited

The code

Before we dive into the code, let's look at the major data structures.

The extended attributes structure

The first is the extended attributes structure:

```
typedef struct cfs_attr_s
{
  iofunc_attr_t     attr;

  int               nels;
  int               nalloc;
  union {
    struct des_s      *dirblocks;
    iov_t             *fileblocks;
    char              *symlinkdata;
  } type;
} cfs_attr_t;
```

As normal, the regular attributes structure, *attr*, is the first member. After this, the three fields are:

nels

The number of elements actually in use. These elements are the *type* union described below.

nalloc

The number of elements allocated. This number may be bigger than *nels* to make more efficient use of allocated memory. Instead of growing the memory each time we need to add one more element, the memory is grown by a multiple (currently, 64). The *nels* member indicates how many are actually in use. This also helps with deallocation, because we don't have to shrink the memory; we simply decrement *nels*.

type

This is the actual type of the entry. As you can see, it's a union of three possible types, corresponding to the three possible data elements that we can store in a filesystem: directories (type `struct des_s`), files (an array of `iov_t`'s), and symbolic links (a string).

For reference, here is the `struct des_s` directory entry type:

```
typedef struct des_s
{
  char        *name;          // name of entry
  cfs_attr_t  *attr;          // attributes structure
}   des_t;
```

It's the name of the directory element (i.e. if you had a file called `spud.txt`, that would be the name of the directory element) and a pointer to the attributes structure corresponding to that element.

From this we can describe the organization of the data stored in the RAM disk.

The root directory of the RAM disk contains one `cfs_attr_t`, which is of type `struct des_s` and holds all of the entries within the root directory. Entries can be files, other directories, or symlinks. If there are 10 entries in the RAM disk's root directory, then *nels* would be equal to 10 (*nalloc* would be 64 because that's the "allocate-at-once" size), and the `struct des_s` member *dirblocks* would be an array with 64 elements in it (with 10 valid), one for each entry in the root directory.

Each of the 10 `struct des_s` entries describes its respective element, starting with the name of the element (the *name* member), and a pointer to the attributes structure for that element.

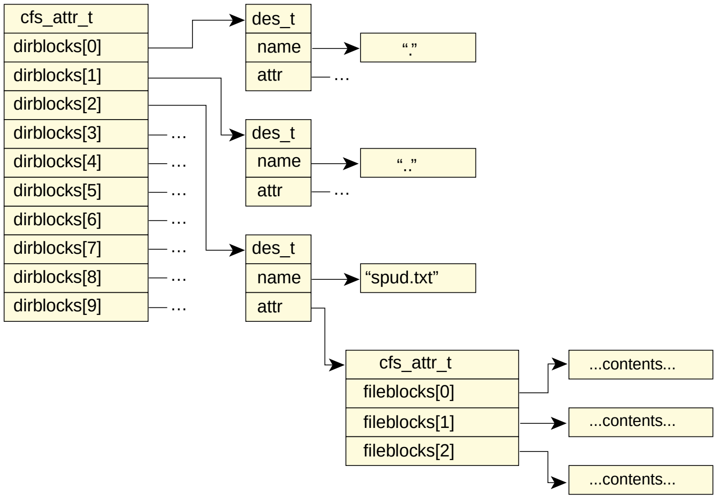


Figure 22: A directory, with subdirectories and a file, represented by the internal data types.

If the element is a text file (our `spud.txt` for example), then its attributes structure would use the *fileblocks* member of the *type* union, and the content of the *fileblocks* would be a list of `iov_ts`, each pointing to the data content of the file.

A direct consequence of this is that we do not support sparse files. A sparse file is one with "gaps" in the allocated space. Some filesystems support this notion. For example, you may write 100 bytes of data at the beginning of the file, *lseek()* forward 1000000 bytes and write another 100 bytes of data. The file will occupy only a few kilobytes on disk, rather than the expected megabyte, because the filesystem didn't store the "unused" data. If, however, you write one megabyte worth of zeros instead of using *lseek()*, then the file would actually consume a megabyte of disk storage.

We don't support that, because all of our `iov_ts` are implicitly contiguous. As an exercise, you could modify the filesystem to have variable-sized `iov_ts`, with the constant `NULL` instead of the address member to indicate a "gap."

If the element was a symbolic link, then the *symlinkdata* union member is used instead; the *symlinkdata* member contains a *strdup()*'d copy of the contents of the symbolic link. Note that in the case of symbolic links, the *nels* and *nalloc* members are not used, because a symbolic link can have only one value associated with it.

The *mode* member of the base attributes structure is used to determine whether we should look at the *dirblocks*, *fileblocks*, or *symlinkdata* union member. (That's why there appears to be no "demultiplexing" variable in the structure itself; we rely on the base one provided by the resource manager framework.)

A question that may occur at this point is, "Why isn't the name stored in the attributes structure?" The short answer is: hard links. A file may be known by multiple names, all hard-linked together. So, the actual "thing" that represents the file is an unnamed object, with zero or more named objects pointing to it. (I said "zero" because the file could be open, but unlinked. It still exists, but doesn't have any named object pointing to it.)

The *io_read()* function

Probably the easiest function to understand is the *io_read()* function. As with all resource managers that implement directories, *io_read()* has both a file personality and a directory personality.

The decision as to which personality to use is made very early on, and then branches out into the two handlers:

```
int
cfs_io_read (resmgr_context_t *ctp, io_read_t *msg,
             RESMGR_OCB_T *ocb)
{
  int   sts;

  // use the helper function to decide if valid
  if ((sts = iofunc_read_verify (ctp, msg, ocb, NULL)) != EOK) {
    return (sts);
  }

  // decide if we should perform the "file" or "dir" read
  if (S_ISDIR (ocb -> attr -> attr.mode)) {
    return (ramdisk_io_read_dir (ctp, msg, ocb));
  } else if (S_ISREG (ocb -> attr -> attr.mode)) {
    return (ramdisk_io_read_file (ctp, msg, ocb));
  } else {
    return (EBADF);
  }
}
```

The functionality above is standard, and you'll see similar code in every resource manager that has these two personalities. It would almost make sense for the resource manager framework to provide two distinct callouts, say an *io_read_file()* and an *io_read_dir()* callout.

It's interesting to note that the previous version of the operating system, QNX 4, did in fact have two separate callouts, one for “read a file” and one for “read a directory.” However, to complicate matters a bit, it also had *two* separate open functions, one to open a file, and one to open a “handle.”

Win some, lose some.

To read the directory entry, the code is almost the same as what we've seen in the Web Counter Resource Manager chapter.

I'll point out the differences:

```
int
ramdisk_io_read_dir (resmgr_context_t *ctp, io_read_t *msg,
                     iofunc_ocb_t *ocb)
{
  int   nbytes;
  int   nleft;
  struct  dirent *dp;
  char  *reply_msg;
  char  *fname;
  int   pool_flag;

  // 1) allocate a buffer for the reply
  if (msg -> i.nbytes <= 2048) {
    reply_msg = mpool_calloc (mpool_readdir);
    pool_flag = 1;
  } else {
    reply_msg = calloc (1, msg -> i.nbytes);
    pool_flag = 0;
  }

  if (reply_msg == NULL) {
    return (ENOMEM);
  }

  // assign output buffer
  dp = (struct dirent *) reply_msg;

  // we have "nleft" bytes left
  nleft = msg -> i.nbytes;
  while (ocb -> offset < ocb -> attr -> nels) {

    // 2) short-form for name
    fname = ocb -> attr -> type.dirblocks [ocb -> offset].name;

    // 3) if directory entry is unused, skip it
    if (!fname) {
      ocb -> offset++;
      continue;
    }

    // see how big the result is
    nbytes = dirent_size (fname);

    // do we have room for it?
    if (nleft - nbytes >= 0) {

      // fill the dirent, and advance the dirent pointer
      dp = dirent_fill (dp, ocb -> offset + 1,
                        ocb -> offset, fname);

      // move the OCB offset
      ocb -> offset++;

      // account for the bytes we just used up
      nleft -= nbytes;

    } else {

      // don't have any more room, stop
      break;
    }
  }

  // if we returned any entries, then update the ATIME
  if (nleft != msg -> i.nbytes) {
    ocb -> attr -> attr.flags |= IOFUNC_ATTR_ATIME
                              | IOFUNC_ATTR_DIRTY_TIME;
  }

  // return info back to the client
  MsgReply (ctp -> rcvid, (char *) dp - reply_msg, reply_msg,
            (char *) dp - reply_msg);

  // 4) release our buffer
  if (pool_flag) {
    mpool_free (mpool_readdir, reply_msg);
  } else {
    free (reply_msg);
  }

  // tell resource manager library we already did the reply
  return (_RESMGR_NOREPLY);
}
```

There are four important differences in this implementation compared to the implementations we've already seen:

1. Instead of calling *malloc()* or *calloc()* all the time, we've implemented our own memory-pool manager. This results in a speed and efficiency improvement because, when we're reading directories, the size of the allocations is always the same. If it's not, we revert to using *calloc()*. Note that we keep track of where the memory came from by using the *pool_flag*.
2. In previous examples, we generated the name ourselves via *sprintf()*. In this case, we need to return the actual, arbitrary names that are stored in the RAM-disk directory entries. While dereferencing the name may look complicated, it's only looking through the OCB to find the attributes structure, and from there it's looking at the directory structure as indicated by the offset stored in the OCB.
3. Oh yes, directory gaps. When an entry is deleted (i.e. `rm spud.txt`), the temptation is to *move* all the entries by one to cover the hole (or, at least to swap the last entry with the hole). This would let you eventually shrink the directory entry, because you know that all the elements at the end are blank. By examining *nels* versus *nalloc* in the extended attributes structure, you could make a decision to shrink the directory. But alas! That's not playing by the rules, because you cannot move directory entries around as you see fit, unless absolutely no one is using the directory entry. So, you must be able to support directory entries with holes. (As an exercise, you can add this “optimization cleanup” in the *io_close_ocb()* handler when you detect that the use-count for the directory has gone to zero.)
4. Depending on where we allocated our buffer from, we need to return it to the correct place.

Apart from the above comments, it's a plain directory-based *io_read()* function.

To an extent, the basic skeleton for the file-based *io_read()* function, *ramdisk_io_read_file()*, is also common. What's not common is the way we get the data. Recall that in the web counter resource manager (and in the `atoz` resource manager in the previous book) we manufactured our data on the fly. Here, we must dutifully return the exact same data as what the client wrote in.

Therefore, what you'll see here is a bunch of code that deals with blocks and `iov_t`s. For reference, this is what an `iov_t` looks like:

```
typedef struct iovec {
  void     *iov_base;
  uint32_t   iov_len;
} iov_t;
```

(This is a slight simplification; see `<sys/target_nto.h>` for the whole story.) The *iov_base* member points to the data area, and the *iov_len* member indicates the size of that data area. We create arrays of `iov_t`s in the RAM-disk filesystem to hold our data. The `iov_t` is also the native data type used with the message-passing functions, like *MsgReplyv()*, so it's natural to use this data type, as you'll see soon.

Before we dive into the code, let's look at some of the cases that come up during access of the data blocks. The same cases (and others) come up during the write implementation as well.

We'll assume that the block size is 4096 bytes.

When reading blocks, there are several cases to consider:

- The total transfer will originate from within one block.
- The transfer will span two blocks, perhaps not entirely using either block fully.
- The transfer will span many blocks; the intermediate blocks will be fully used, but the end blocks may not be.

It's important to understand these cases, especially since they relate to boundary transfers of:

- 1 byte
- 2 bytes within the same block
- 2 bytes and spanning two blocks
- 4096 bytes within one complete block
- more than 4096 bytes within two blocks (the first block complete and the second incomplete)
- more than 4096 bytes within two blocks (the first block incomplete and the second incomplete)
- more than 4096 bytes within two blocks (the first block incomplete and the second complete)
- more than 4096 bytes within more than two blocks (like the three cases immediately above, but with one or more full intermediate blocks).

Believe me, I had fun drawing diagrams on the white board as I was coding this. `:-)`

Copyright © 2014, QNX Software Systems Limited

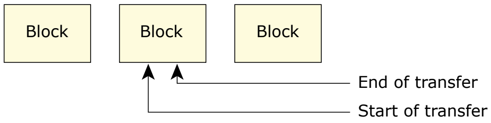


Figure 23: Total transfer originating entirely within one block.

In the above diagram, the transfer starts somewhere within one block and ends somewhere within the same block.

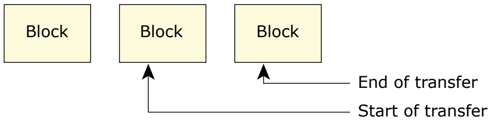


Figure 24: Total transfer spanning a block.

In the above diagram, the transfer starts somewhere within one block, and ends somewhere within the next block. There are no full blocks transferred. This case is similar to the case above it, except that two blocks are involved rather than just one block.

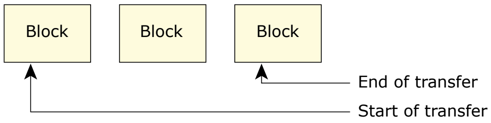


Figure 25: Total transfer spanning at least one full block.

In the above diagram, we see the case of having the first and last blocks incomplete, with one (or more) full intermediate blocks.

Keep these diagrams in mind when you look at the code.

```
int
ramdisk_io_read_file (resmgr_context_t *ctp, io_read_t *msg,
                      iofunc_ocb_t *ocb)
{
  int   nbytes;
  int   nleft;
  int   towrite;
  iov_t *iovs;
  int   niovs;
  int   so;      // start offset
  int   sb;      // start block
  int   i;
  int   pool_flag;

  // we don't do any xtypes here...
  if ((msg -> i.xtype & _IO_XTYPE_MASK) != _IO_XTYPE_NONE) {
    return (ENOSYS);
  }

  // figure out how many bytes are left
  nleft = ocb -> attr -> attr.nbytes - ocb -> offset;

  // and how many we can return to the client
  nbytes = min (nleft, msg -> i.nbytes);

  if (nbytes) {

    // 1) calculate the number of IOVs that we'll need
    niovs = nbytes / BLOCKSIZE + 2;
    if (niovs <= 8) {
      iovs = mpool_malloc (mpool_iov8);
      pool_flag = 1;
    } else {
      iovs = malloc (sizeof (iov_t) * niovs);
      pool_flag = 0;
    }
    if (iovs == NULL) {
      return (ENOMEM);
    }

    // 2) find the starting block and the offset
    so = ocb -> offset & (BLOCKSIZE - 1);
    sb  = ocb -> offset / BLOCKSIZE;
    towrite = BLOCKSIZE - so;
    if (towrite > nbytes) {
      towrite = nbytes;
    }

    // 3) set up the first block
    SETIOV (&iovs [0], (char *)
            (ocb -> attr -> type.fileblocks [sb].iov_base) + so, towrite);

    // 4) account for the bytes we just consumed
    nleft = nbytes - towrite;

    // 5) setup any additional blocks
    for (i = 1; nleft > 0; i++) {
      if (nleft > BLOCKSIZE) {
        SETIOV (&iovs [i],
                ocb -> attr -> type.fileblocks [sb + i].iov_base,
                BLOCKSIZE);
        nleft -= BLOCKSIZE;
      } else {

        // 6) handle a shorter final block
        SETIOV (&&iovs [i],
                ocb -> attr -> type.fileblocks [sb + i].iov_base, nleft);
        nleft = 0;
      }
    }

    // 7) return it to the client
    MsgReplyv (ctp -> rcvid, nbytes, iovs, i);

    // update flags and offset
    ocb -> attr -> attr.flags |= IOFUNC_ATTR_ATIME
                              | IOFUNC_ATTR_DIRTY_TIME;
    ocb -> offset += nbytes;

    if (pool_flag) {
      mpool_free (mpool_iov8, iovs);
    } else {
      free (iovs);
    }
  } else {
    // nothing to return, indicate End Of File
    MsgReply (ctp -> rcvid, EOK, NULL, 0);
  }

  // already done the reply ourselves
  return (_RESMGR_NOREPLY);
}
```

We won't discuss the standard resource manager stuff, but we'll focus on the unique functionality of this resource manager.

1. We're going to be using IOVs for our data-transfer operations, so we need to allocate an array of them. The number we need is the number of bytes that we'll be transferring plus 2 — we need an extra one in case the initial block is short, and another one in case the final block is short. Consider the case were we're transferring two bytes on a block boundary. The `nbytes / BLOCKSIZE` calculation yields zero, but we need one more block for the first byte and one more block for the second byte. Again, we allocate the IOVs from a pool of IOVs because the maximum

size of IOV allocation fits well within a pool environment. We have a *malloc()* fallback in case the size isn't within the capabilities of the pool.

2. Since we could be at an arbitrary offset within the file when we're asked for bytes, we need to calculate where the first block is, and where in that first block our offset for transfers should be.
3. The first block is special, because it may be shorter than the block size.
4. We need to account for the bytes we just consumed, so that we can figure out how many remaining blocks we can transfer.
5. All intermediate blocks (i.e. not the first and not the last) will be `BLOCKSIZE` bytes in length.
6. The last block may or may not be `BLOCKSIZE` bytes in length, because it may be shorter.
7. Notice how the IOVs are used with the *MsgReplyv()* to return the data from the client.

The main trick was to make sure that there were no boundary or off-by-one conditions in the logic that determines which block to start at, how many bytes to transfer, and how to handle the final block. Once that was worked out, it was smooth sailing as far as implementation.

You could optimize this further by returning the IOVs *directly* from the extended attributes structure's *fileblocks* member, but beware of the first and last block — you might need to modify the values stored in the *fileblocks* member's IOVs (the address and length of the first block, and the length of the last block), do your *MsgReplyv()*, and then restore the values. A little messy perhaps, but a tad more efficient.

The *io_write()* function

Another easy function to understand is the *io_write()* function. It gets a little more complicated because we have to handle allocating blocks when we run out (i.e. when we need to extend the file because we have written past the end of the file).

The *io_write()* functionality is presented in two parts, one is a fairly generic *io_write()* handler, the other is the actual block handler that writes the data to the blocks.

The generic *io_write()* handler looks at the current size of the file, the OCB's *offset* member, and the number of bytes being written to determine if the handler needs to extend the number of blocks stored in the *fileblocks* member of the extended attributes structure. Once that determination is made, and blocks have been added (and zeroed!), then the RAM-disk-specific write handler, *ramdisk_io_write()*, is called.

The following diagram illustrates the case where we need to extend the blocks stored in the file:

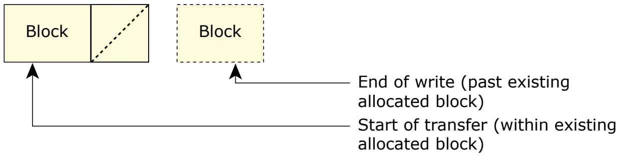

Block
Block
End of write (past existing allocated block)
Start of transfer (within existing allocated block)

Figure 26: A write that overwrites existing data in the file, adds data to the "unused" portion of the current last block, and then adds one more block of data.

The following shows what happens when the RAM disk fills up. Initially, the write would want to perform something like this:

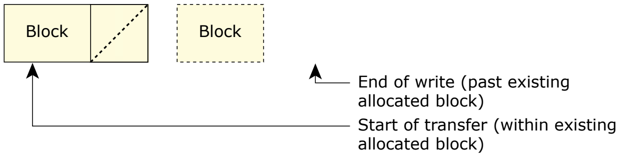

Block
Block
End of write (past existing allocated block)
Start of transfer (within existing allocated block)

Figure 27: A write that requests more space than exists on the disk.

However, since the disk is full (we could allocate only one more block), we trim the write request to match the maximum space available:

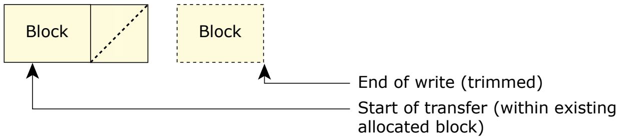


Figure 28: A write that's been trimmed due to lack of disk space.

There was only 4 KB more available, but the client requested more than that, so the request was trimmed.

```
int
cfs_io_write (resmgr_context_t *ctp, io_write_t *msg,
              RESMGR_OCB_T *ocb)
{
  cfs_attr_t    *attr;
  int       i;
  off_t       newsize;

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

  // shortcuts
  attr = ocb -> attr;
  newsize = ocb -> offset + msg -> i.nbytes;

  // 1) see if we need to grow the file
  if (newsize > attr -> attr.nbytes) {
    // 2) truncate to new size using TRUNCATE_ERASE
    cfs_a_truncate (attr, newsize, TRUNCATE_ERASE);
    // 3) if it's still not big enough
    if (newsize > attr -> attr.nbytes) {
      // 4) trim the client's size
      msg -> i.nbytes = attr -> attr.nbytes - ocb -> offset;
      if (!msg -> i.nbytes) {
        return (ENOSPC);
      }
    }
  }

  // 5) call the RAM disk version
  return (ramdisk_io_write (ctp, msg, ocb));
```

Copyright © 2014, QNX Software Systems Limited

```
}
```

The code walkthrough is as follows:

1. We compare the *newsize* (derived by adding the OCB's *offset* plus the number of bytes the client wants to write) against the current size of the resource. If the *newsize* is less than or equal to the existing size, then it means we don't have to grow the file, and we can skip to step 5.
2. We decided that we needed to grow the file. We call *cfs_a_truncate()*, a utility function, with the parameter TRUNCATE_ERASE. This will attempt to grow the file to the required size by adding zero-filled blocks. However, we could run out of space while we're doing this. There's another flag we could have used, TRUNCATE_ALL_OR_NONE, which would either grow the file to the required size or not. The TRUNCATE_ERASE flag grows the file to the desired size, but *does not* release newly added blocks in case it runs out of room. Instead, it simply adjusts the base attributes structure's *nbytes* member to indicate the size it was able to grow the file to.
3. Now we check to see if we were able to grow the file to the required size.
4. If we can't grow the file to the required size (i.e. we're out of space), then we trim the size of the client's request by storing the actual number of bytes we can write back into the message header's *nbytes* member. (We're pretending that the client asked for fewer bytes than they really asked for.) We calculate the number of bytes we can write by subtracting the total available bytes minus the OCB's *offset* member.
5. Finally, we call the RAM disk version of the *io_write()* routine, which deals with getting the data from the client and storing it in the disk blocks.

As mentioned above, the generic *io_write()* function isn't doing anything that's RAM-disk-specific; that's why it was separated out into its own function.

Now, for the RAM-disk-specific functionality. The following code implements the block-management logic (refer to the diagrams for the read logic):

```
int
ramdisk_io_write (resmgr_context_t *ctp, io_write_t *msg,
                  RESMGR_OCB_T *ocb)
{
  cfs_attr_t  *attr;
  int         sb;      // startblock
  int         so;      // startoffset
  int         lb;      // lastblock
  int         nbytes, nleft;
  int         toread;
  iov_t       *newblocks;
  int         i;
  off_t       newsize;
  int         pool_flag;

  // shortcuts
  nbytes = msg -> i.nbytes;
  attr = ocb -> attr;
  newsize = ocb -> offset + nbytes;

  // 1) precalculate the block size constants...
  sb = ocb -> offset / BLOCKSIZE;
  so = ocb -> offset & (BLOCKSIZE - 1);
  lb = newsize / BLOCKSIZE;
  // 2) allocate IOVs
  i = lb - sb + 1;
  if (i <= 8) {
    newblocks = mpool_malloc (mpool_iov8);
    pool_flag = 1;
  } else {
    newblocks = malloc (sizeof (iov_t) * i);
    pool_flag = 0;
  }

  if (newblocks == NULL) {
    return (ENOMEM);
  }

  // 3) calculate the first block size
  toread = BLOCKSIZE - so;
  if (toread > nbytes) {
    toread = nbytes;
  }
  SETIOV (&newblocks [0], (char *)
          (attr -> type.fileblocks [sb].iov_base) + so, toread);

  // 4) now calculate zero or more blocks;
  //    special logic exists for a short final block
  nleft = nbytes - toread;
  for (i = 1; nleft > 0; i++) {
    if (nleft > BLOCKSIZE) {
      SETIOV (&newblocks [i],
              attr -> type.fileblocks [sb + i].iov_base, BLOCKSIZE);
      nleft -= BLOCKSIZE;
    } else {
      SETIOV (&newblocks [i],
              attr -> type.fileblocks [sb + i].iov_base, nleft);
      nleft = 0;
    }
  }

  // 5) transfer data from client directly into the ramdisk...
  resmgr_msgreadv (ctp, newblocks, i, sizeof (msg -> i));

  // 6) clean up
  if (pool_flag) {
    mpool_free (mpool_iov8, newblocks);
  } else {
    free (newblocks);
  }

  // 7) use the original value of nbytes here...
  if (nbytes) {
    attr -> attr.flags |= IOFUNC_ATTR_MTIME | IOFUNC_ATTR_DIRTY_TIME;
    ocb -> offset += nbytes;
  }
  _IO_SET_WRITE_NBYTES (ctp, nbytes);
  return (EOK);
}
```

1. We precalculate some constants to make life easier later on. The *sb* variable contains the starting block number where our writing begins. The *so* variable ("start offset") contains the offset into the start block where writing begins (we may be writing somewhere other than the first byte of the block). Finally, *lb* contains the last block number affected by the write. The *sb* and *lb* variables define the range of blocks affected by the write.
2. We're going to allocate a number of IOVs (into the *newblocks* array) to point into the blocks, so that we can issue the *MsgRead()* (via *resmgr_msgreadv()* in step 5, below). The `+ 1` is in place in case the *sb* and *lb* are the same — we still need to read *at least* one block.

3. The first block that we read may be short, because we don't necessarily start at the beginning of the block. The *toread* variable contains the number of bytes we transfer in the first block. We then set this into the first *newblocks* array element.
4. The logic we use to get the rest of the blocks is based on the remaining number of bytes to be read, which is stored in *nleft*. The `for` loop runs until *nleft* is exhausted (we are guaranteed to have enough IOVs, because we calculated the number in step 1, above).
5. Here we use the *resmgr_msgreadv()* function to read the actual data from the client directly into our buffers through the *newblocks* IOV array. We don't read the data from the passed message, *msg*, because we may not have enough data from the client sitting in that buffer (even if we somehow determine the size a priori and specify it in the *resmgr_attr.msg_max_size*, the network case doesn't necessarily transfer all of the data). In the network case, this *resmgr_msgreadv()* may be a blocking call — just something to be aware of.
6. Clean up after ourselves. The flag *pool_flag* determines where we allocated the data from.
7. If we transferred any data, adjust the access time as per POSIX.

The *c_open()* function

Possibly the most complex function, *c_open()* performs the following:

1. Find the target.
2. Analyze the *mode* flag, and create/truncate as required.
3. Bind the OCB and attributes structure.

We'll look at the individual sub-tasks listed above, and then delve into the code walkthrough for the *c_open()* call itself at the end.

Finding the target

In order to find the target, it seems that all we need to do is simply break the pathname apart at the / characters and see if each component exists in the *dirblocks* member of the extended attributes structure. While that's basically true at the highest level, as the saying goes, “The devil is in the details.”

Permission-checks complicate this matter slightly. Symbolic links complicate this matter significantly (a symbolic link can point to a file, a directory, or another symbolic link). And, to make things even more complicated, under certain conditions the target may not even exist, so we may need to operate on the directory entry above the target instead of the target itself.

So, the connect function (*c_open()*) calls *connect_msg_to_attr()*, which in turn calls *pathwalk()*.

Copyright © 2014, QNX Software Systems Limited

The *pathwalk()* function

The *pathwalk()* function is called only by *connect_msg_to_attr()* and by the rename function (*c_rename()*, which we'll see later). Let's look at this lowest-level function first, and then we'll proceed up the call hierarchy.

```
int
pathwalk (resmgr_context_t *ctp, char *pathname,
          cfs_attr_t *mountpoint, int flags, des_t *output,
          int *nrets, struct _client_info *cinfo)
{
  int           nels;
  int           sts;
  char          *p;

  // 1) first, we break apart the slash-separated pathname
  memset (output, 0, sizeof (output [0]) * *nrets);
  output [0].attr = mountpoint;
  output [0].name = "";

  nels = 1;
  for (p = strtok (pathname, "/"); p; p = strtok (NULL, "/")) {
    if (nels >= *nrets) {
      return (E2BIG);
    }
    output [nels].name = p;
    output [nels].attr = NULL;
    nels++
  }

  // 2) next, we analyze each pathname
  for (*nrets = 1; *nrets < nels; ++*nrets) {

    // 3) only directories can have children.
    if (!S_ISDIR (output [*nrets - 1].attr -> attr.mode)) {
      return (ENOTDIR);
    }

    // 4) check access permissions
    sts = iofunc_check_access (ctp,
                               &output [*nrets-1].attr -> attr,
                               S_IEXEC, cinfo);
    if (sts != EOK) {
      return (sts);
    }

    // 5) search for the entry
    output [*nrets].attr = search_dir (output [*nrets].name,
                                       output [*nrets-1].attr);
    if (!output [*nrets].attr) {
      ++*nrets;
      return (ENOENT);
    }

    // 6) process the entry
    if (S_ISLNK (output [*nrets].attr -> attr.mode)) {
      ++*nrets;
      return (EOK);
    }
  }

  // 7) everything was okay
  return (EOK);
}
```

The *pathwalk()* function fills the *output* parameter with the pathnames and attributes structures of each pathname component. The **nrets* parameter is used as both an input and an output. In the input case it tells *pathwalk()* how big the *output* array is, and when *pathwalk()* returns, **nrets* is used to indicate how many elements were successfully processed (see the walkthrough below). Note that the way that we've broken the string into pieces first, and *then* processed the individual components one

Copyright © 2014, QNX Software Systems Limited

at a time means that when we abort the function (for any of a number of reasons as described in the walkthrough), the *output* array may have elements that are valid past where the **nrets* variable indicates. This is actually useful; for example, it lets us get the pathname of a file or directory that we're creating (and hence doesn't exist). It also lets us check if there are *additional* components past the one that we're creating, which would be an error.

Detailed walkthrough:

1. The first element of the output string is, by definition, the attributes structure corresponding to the mount point and to the empty string. The `for` loop breaks the pathname string apart at each and every / character, and checks to see that there aren't too many of them (the caller tells us how many they have room for in the passed parameter **nrets*).

   Note that we use *strtok()* which isn't thread-safe; in this resource manager we are single-threaded. We would have used *strtok_r()* if thread-safety were a concern.

2. Next, we enter a `for` loop that analyzes each pathname component. It's within this loop that we do all of the checking. Note that the variable **nrets* points to the "current" pathname element.
3. In order for the current pathname element to even be valid, its parent (i.e. **nrets* minus 1) *must* be a directory, since only directories can have children. If that isn't the case, we return `ENOTDIR` and abort. Note that when we abort, the **nrets* return value includes the nondirectory that failed.
4. We use the helper function *iofunc_check_access()* to verify accessibility for the component. Note that if we abort, **nrets* includes the inaccessible directory.
5. At this point, we have verified that everything is okay up to the entry, and all we need to do is find the entry within the directory. The helper function *search_dir()* looks through the *dirblocks* array member of the extended attributes structure and tries to find our entry. If the entry isn't found, **nrets* includes the entry. (This is important to make note of when creating files or directories that don't yet exist!)
6. We check if the entry itself is a symbolic link. If it is, we give up, and let higher levels of software deal with it. We return `EOK` because there's nothing actually *wrong* with being a symbolic link, it's just that we can't do anything about it at this level. (Why? The symbolic link could be a link to a completely different filesystem that we have no knowledge of.) The higher levels of software will eventually tell the client that the entry is a symlink, and the client's library then tries the path again — that's why we don't worry about infinite symlink loops and other stuff in our resource manager. The **nrets* return value includes the entry.
7. Finally, if everything works, (we've gone through all of the entries and found and verified each and every one of them) we return `EOK` and **nrets* contains all pathname elements.

The job of **nrets* is to give the higher-level routines an indication of where the processing stopped. The return value from *pathwalk()* will tell them why it stopped.

The *connect_msg_to_attr()* function

The next-higher function in the call hierarchy is *connect_msg_to_attr()*. It calls *pathwalk()* to break apart the pathname, and then looks at the return code, the type of request, and other parameters to make a decision.

You'll see this function used in most of the resource manager connect functions in the RAM disk.

After *pathwalk()*, several scenarios are possible:

- All components within the pathname were accessible, of the correct type, and present. In this case, pathname processing is done, and we can continue on to the next step (a zero value, indicating "all OK," is returned).
- As above, except that the final component doesn't exist. In this case, we *may* be done; it depends on whether we're creating the final component or not (a zero value is returned, but *rval* is set to ENOENT). We leave it to a higher level to determine if the final component was required.
- A component in the pathname was not a directory, does not exist, or the client doesn't have permission to access it. In this case, we're done as well, but we abort with an error return (a nonzero is returned, and *rval* is set to the error number).
- A component in the pathname is a symbolic link. In this case, we're done as well, and we perform a symlink redirect. A nonzero is returned, which should be passed up to the resource-manager framework of the caller.

This function accepts two parameters, *parent* and *target*, which are used extensively in the upper levels to describe the directory that contains the target, as well as the target itself (if it exists).

```
int
connect_msg_to_attr (resmgr_context_t *ctp,
                     struct _io_connect *cmsg,
                     RESMGR_HANDLE_T *handle,
                     des_t *parent, des_t *target,
                     int *sts, struct _client_info *cinfo)
{
  des_t     components [_POSIX_PATH_MAX];
  int       ncomponents;

  // 1) Find target, validate accessibility of components
  ncomponents = _POSIX_PATH_MAX;
  *sts = pathwalk (ctp, cmsg -> path, handle, 0, components,
                   &ncomponents, cinfo);

  // 2) Assign parent and target
  *target = components [ncomponents - 1];
  *parent = ncomponents == 1 ? *target
                             : components [ncomponents - 2];

  // 3) See if we have an error, abort.
  if (*sts == ENOTDIR || *sts == EACCES) {
    return (1);
  }

  // 4) missing non-final component
  if (components [ncomponents].name != NULL && *sts == ENOENT) {
    return (1);
```

Copyright © 2014, QNX Software Systems Limited

```
  }

  if (*sts == EOK) {
    // 5) if they wanted a directory, and we aren't one, honk.
    if (S_ISDIR (cmsg -> mode)
    && !S_ISDIR (components [ncomponents-1].attr->attr.mode)) {
      *sts = ENOTDIR;
      return (1);
    }

    // 6) yes, symbolic links are complicated!
    //    (See walkthrough and notes)
    if (S_ISLNK (components [ncomponents - 1].attr -> attr.mode)
    && (components [ncomponents].name
        || (cmsg -> eflag & _IO_CONNECT_EFLAG_DIR)
        || !S_ISLNK (cmsg -> mode))) {
      redirect_symlink (ctp, cmsg, target -> attr,
                        components, ncomponents);
      *sts = _RESMGR_NOREPLY;
      return (1);
    }
  }
  // 7) all OK
  return (0);
}
```

1. Call *pathwalk()* to validate the accessibility of all components. Notice that we use the `des_t` directory entry structure that we used in the extended attributes structure for the call to *pathwalk()* — it's best if you don't need to reinvent many similar but slightly different data types.
2. The last two entries in the broken-up *components* array are the last two pathname components. However, there may be only one entry. (Imagine creating a file in the root directory of the filesystem — the file that you're creating doesn't exist, and the root directory of the filesystem is the first and only entry in the broken-up pathname components.) If there is only one entry, then assign the last entry to both the *parent* and *target*.
3. Now take a look and see if there were any problems. The two problems that we're interested in at this point are missing directory components and the inability to access some path component along the way. If it's either of these two problems, we can give up right away.
4. We're missing an intermediate component (i.e. `/valid/missing/extra/extra` where `missing` is not present).
5. The caller of *connect_msg_to_attr()* passes its connect message, which includes a *mode* field. This indicates what kind of thing it's expecting the target to be. If the caller wanted a directory, but the final component isn't a directory, we return an error as well.
6. Symbolic links. Remember that *pathwalk()* aborted at the symbolic link (if it found one) and didn't process any of the entries below the symlink (see below).
7. Everything passed.

Fun with symlinks

Symbolic links complicate the processing greatly.

Let's spend a little more time with the line:

```
if (
  S_ISLNK (components [ncomponents - 1].attr -> attr.mode)
  &&
    (
      components [ncomponents].name
      || (cmsg -> eflag & _IO_CONNECT_EFLAG_DIR)
      || !S_ISLNK (cmsg -> mode)
    )
   )
{
```

I've broken it out over a few more lines to clarify the logical relationships. The very first condition (the one that uses the macro *S_ISLNK()*) gates the entire `if` clause. If the entry we are looking at is *not* a symlink, we can give up right away, and continue to the next statement.

Next, we examine a three-part OR condition. We perform the redirection if *any* of the following conditions is true:

- We have an intermediate component. This would be the case in the example `/valid/symlink/extra` where we are currently positioned at the `symlink` part of the pathname. In this case, we *must* look through the symlink.
- The `_IO_CONNECT_EFLAG_DIR` flag of the connect message's *eflag* member is set. This indicates that we wish to proceed as if the entity is a directory, and that means that we need to look through the symbolic link.
- The connect message's *mode* member indicates symlink operation. This is a flag that's set to indicate that the connect message refers to the contents of the symlink, so we need to redirect.

In case we need to follow the symlink, we don't do it ourselves! It's not the job of this resource manager's connect functions to follow the symlink. All we need to do is call *redirect_symlink()* and it will reply with a redirect message back to the client's *open()* (or other connect function call). All clients' *open()* calls know how to handle the redirection, and *they* (the clients) are responsible for retrying the operation with the new information from the resource manager.

To clarify:

- We have a symlink in our RAM disk: `s -> /dev/ser1`.
- A client issues an `open ("/ramdisk/s", O_WRONLY);` call.
- The process manager directs the client to the RAM disk, where it sends an *open()* message.
- The RAM disk processes the `s` pathname component, then redirects it. This means that the client gets a message of “Redirect: Look in `/dev/ser1` instead.”
- The client asks the process manager who it should talk to about `/dev/ser1` and the client is told the serial port driver.
- The client opens the serial port.

Copyright © 2014, QNX Software Systems Limited

So, it's important to note that after the RAM disk performed the “redirect” function, it was out of the loop after that point.

Analyze the *mode* flag

We've made sure that the pathname is valid, and we've resolved any symbolic links that we needed to. Now we need to figure out the *mode* flags.

There are a few combinations that we need to take care of:

- If both the O_CREAT and O_EXCL flags are set, then the target must not exist (else we error-out with EEXIST).
- If the flag O_CREAT is set, the target may or may not exist; we might be creating it.
- If the flag O_CREAT is not set, then the target must exist (else we error-out with ENOENT).
- If the flag O_TRUNC and either O_RDWR or O_WRONLY are set, then we need to trim the target's length to zero and wipe out its data.

This may involve creating or truncating the target, or returning error indications. We'll see this in the code walkthrough below.

Bind the OCB and attributes structure

To bind the OCB and the attributes structures, we simply call the utility functions (see the walkthrough, below).

Finally, the *c_open()* code walkthrough

Now that we understand all of the steps involved in processing the *c_open()* (and, coincidentally, large chunks of all other connect functions), it's time to look at the code.

```
int
cfs_c_open (resmgr_context_t *ctp, io_open_t *msg,
            RESMGR_HANDLE_T *handle, void *extra)
{
  int       sts;
  des_t     parent, target;
  struct    _client_info cinfo;

  // 1) fetch the client information
  if (sts = iofunc_client_info (ctp, 0, &cinfo)) {
    return (sts);
  }

  // 2) call the helper connect_msg_to_attr
  if (connect_msg_to_attr (ctp, &msg -> connect, handle,
                           &parent, &target, &sts, &cinfo)) {
    return (sts);
  }

  // if the target doesn't exist
  if (!target.attr) {
    // 3) and we're not creating it, error
    if (!(msg -> connect.ioflag & O_CREAT)) {
      return (ENOENT);
    }

    // 4) else we are creating it, call the helper iofunc_open
    sts = iofunc_open (ctp, msg, NULL, &parent.attr -> attr,
                       NULL);
    if (sts != EOK) {
      return (sts);
    }

    // 5) create an attributes structure for the new entry
    target.attr = cfs_a_mkfile (parent.attr,
                                target.name, &cinfo);
    if (!target.attr) {
      return (errno);
    }

  // else the target exists
  } else {
    // 6) call the helper function iofunc_open
    sts = iofunc_open (ctp, msg, &target.attr -> attr,
                       NULL, NULL);
    if (sts != EOK) {
      return (sts);
    }
  }

  // 7) Target existed or just created, truncate if required.
  if (msg -> connect.ioflag & O_TRUNC) {
    // truncate at offset zero because we're opening it:
    cfs_a_truncate (target.attr, 0, TRUNCATE_ERASE);
  }

  // 8) bind the OCB and attributes structures
  sts = iofunc_ocb_attach (ctp, msg, NULL,
                           &target.attr -> attr, NULL);

  return (sts);
}
```

Walkthrough

The walkthrough is as follows:

1. The “client info” is used by a lot of the called functions, so it's best to fetch it in one place. It tells us about the client, such as the client's node ID, process ID, group, etc.
2. We discussed the *connect_msg_to_attr()* earlier.
3. If the target doesn't exist, and we don't have the O_CREAT flag set, then we return an error of ENOENT.
4. Otherwise, we do have the O_CREAT flag set, so we need to call the helper function *iofunc_open()*. The helper function performs a lot of checks for us (including, for example, the check against O_EXCL).
5. We need to create a new attributes structure for the new entry. In *c_open()* we are only ever creating new *files* (directories are created in *c_mknod()* and symlinks are created in *c_link()*). The helper routine *cfs_a_mkfile()* initializes the attributes structure for us (the extended part, not the base part; that was done earlier by *iofunc_open()*).
6. If the target exists, then we just call *iofunc_open()* (like step 4).
7. Finally, we check the truncation flag, and truncate the file to zero bytes if required. We've come across the *cfs_a_truncate()* call before, when we used it to grow the file in *ramdisk_io_write()*, above. Here, however, it shrinks the size.

8. The OCB and attributes structures are bound via the helper function *iofunc_ocb_attach()*.

The *redirect_symlink()* function

How to redirect a symbolic link is an interesting topic.

First of all, there are two cases to consider: either the symlink points to an absolute pathname (one that starts with a leading / character) or it doesn't and hence is relative.

For the absolute pathname, we need to forget about the current path leading up to the symbolic link, and replace the entire path up to and including the symbolic link with the contents of the symbolic link:

```
ln -s /tmp /ramdisk/tempfiles
```

In that case, when we resolve `/ramdisk/tempfiles`, we will redirect the symlink to `/tmp`. However, in the relative case:

```
ln -s ../resume.html resume.htm
```

When we resolve the relative symlink, we need to preserve the existing pathname up to the symlink, and replace only the symlink with its contents. So, in our example above, if the path was `/ramdisk/old/resume.htm`, we would replace the symlink, `resume.htm`, with its contents, `../resume.html`, to get the pathname `/ramdisk/old/../resume.html` as the redirection result. Someone else is responsible for resolving `/ramdisk/old/../resume.html` into `/ramdisk/resume.html`.

In both cases, we preserve the contents (if any) after the symlink, and simply append that to the substituted value.

Here is the *redirect_symlink()* function presented with comments so that you can see what's going on:

```
static void
redirect_symlink (resmgr_context_t *ctp,
                  struct _io_connect *msg, cfs_attr_t *attr,
                  des_t *components, int ncomponents)
{
  int   eflag;
  int   ftype;
  char  newpath [PATH_MAX];
  int   i;
  char  *p;
  struct _io_connect_link_reply     link_reply;

  // 1) set up variables
  i = 1;
  p = newpath;
  *p = 0;

  // 2) a relative path, do up to the symlink itself
  if (*attr -> type.symlinkdata != '/') {
    // 3) relative -- copy up to and including
    for (; i < (ncomponents - 1); i++) {
      strcat (p, components [i].name);
      p += strlen (p);
      strcat (p, "/");
      p++;
```

```
        }
    } else {
        // 4) absolute, discard up to and including
        i = ncomponents - 1;
    }

    // 5) now substitute the content of the symlink
    strcat (p, attr -> type.symlinkdata);
    p += strlen (p);

    // skip the symlink itself now that we've substituted it
    i++;

    // 6) copy the rest of the pathname components, if any
    for (; components [i].name && i < PATH_MAX; i++) {
        strcat (p, "/");
        strcat (p, components [i].name);
        p += strlen (p);
    }

    // 7) preserve these, wipe rest
    eflag = msg -> eflag;
    ftype = msg -> file_type;
    memset (&link_reply, 0, sizeof (link_reply));

    // 8) set up the reply
    _IO_SET_CONNECT_RET (ctp, _IO_CONNECT_RET_LINK);
    link_reply.file_type = ftype;
    link_reply.eflag = eflag;
    link_reply.path_len = strlen (newpath) + 1;
    SETIOV (&ctp -> iov [0], &link_reply, sizeof (link_reply));
    SETIOV (&ctp -> iov [1], newpath, link_reply.path_len);

    MsgReplyv (ctp -> rcvid, ctp -> status, ctp -> iov, 2);
}
```

1. The architecture of the RAM-disk resource manager is such that by the time we're called to fix up the path for the symlink, we have the path already broken up into components. Therefore, we use the variable *newpath* (and the pointer *p*) during the reconstruction phase.
2. The variable *ncomponents* tells us how many components were processed before *connect_msg_to_attr()* stopped processing components (in this case, because it hit a symlink). Therefore, `ncomponents - 1` is the index of the symlink entry. We see if the symlink is absolute (begins with `/`) or relative.
3. In the relative case, we need to copy (because we are reconstructing components into *newpath*) all of the components up to but not including the symbolic link.
4. In the absolute case, we discard all components up to and including the symbolic link.
5. We then copy the *contents* of the symlink in place of the symlink, and increment *i* (our index into the original pathname component array).
6. Then we copy the rest of the pathname components, if any, to the end of the new path string that we're constructing.
7. While preparing the reply buffer, we need to preserve the *eflag* and *file_type* members, so we stick them into local variables. Then we clear out the reply buffer via *memset()*.
8. The reply consists of setting a flag via the macro *_IO_SET_CONNECT_RET()* (to indicate that this is a redirection, rather than a pass/fail indication for the client's *open()*), restoring the two flags we needed to preserve, setting the *path_len* parameter to the length of the string that we are returning, and setting up a two

Copyright © 2014, QNX Software Systems Limited

part IOV for the return. The first part of the IOV points to the `struct _io_connect_link_reply` (the header), the second part of the reply points to the string (in our case, *newpath*). Finally, we reply via *MsgReplyv()*.

So basically, the main trick was in performing the symlink substitution, and setting the flag to indicate redirection.

The *c_readlink()* function

This is a simple one. You've already seen how symlinks are stored internally in the RAM-disk resource manager. The job of *c_readlink()* is to return the value of the symbolic link. It's called when you do a full `ls`, for example:

```
# ls -lF /my_temp
lrwxrwxrwx  1 root    root    4 Aug 16 14:06 /my_temp@ -> /tmp
```

Since this code shares a lot in common with the processing for *c_open()*, I'll just point out the major differences.

```
int
cfs_c_readlink (resmgr_context_t *ctp, io_readlink_t *msg,
                RESMGR_HANDLE_T *handle, void *reserved)
{
  des_t   parent, target;
  int     sts;
  int     eflag;
  struct  _client_info cinfo;
  int     tmp;

  // get client info
  if (sts = iofunc_client_info (ctp, 0, &cinfo)) {
    return (sts);
  }

  // get parent and target
  if (connect_msg_to_attr (ctp, &msg -> connect, handle,
                           &parent, &target, &sts, &cinfo)) {
    return (sts);
  }

  // there has to be a target!
  if (!target.attr) {
    return (sts);
  }

  // 1) call the helper function
  sts = iofunc_readlink (ctp, msg, &target.attr -> attr, NULL);
  if (sts != EOK) {
    return (sts);
  }

  // 2) preserve eflag...
  eflag = msg -> connect.eflag;
  memset (&msg -> link_reply, 0, sizeof (msg -> link_reply));
  msg -> link_reply.eflag = eflag;

  // 3) return data
  tmp = strlen (target.attr -> type.symlinkdata);
  SETIOV (&ctp -> iov [0], &msg -> link_reply,
          sizeof (msg -> link_reply));
  SETIOV (&ctp -> iov[1], target.attr -> type.symlinkdata, tmp);
  msg -> link_reply.path_len = tmp;
  MsgReplyv (ctp -> rcvid, EOK, ctp -> iov, 2);
  return (_RESMGR_NOREPLY);
}
```

The detailed code walkthrough is as follows:

1. We use the helper function *iofunc_readlink()* to do basic sanity checking for us. If it's not happy with the parameters, then we return whatever it returned.
2. Just like in symlink redirection, we need to preserve flags; in this case it's just the *eflag* — we zero-out everything else.
3. And, just as in the symlink redirection, we return a two-part IOV; the first part points to the header, the second part points to the string. Note that in this case, unlike symlink redirection, we didn't need to construct the pathname. That's because the goal of this function is to return just the contents of the symlink, and we know that they're sitting in the *symlinkdata* member of the extended attributes structure.

The *c_link()* function

The *c_link()* function is responsible for soft and hard links. A hard link is the “original” link from the dawn of history. It's a method that allows one resource (be it a directory or a file, depending on the support) to have multiple names. In the example in the symlink redirection, we created a symlink from `resume.htm` to `../resume.html`; we could just as easily have created a hard link:

```
# ln ../resume.html resume.htm
```

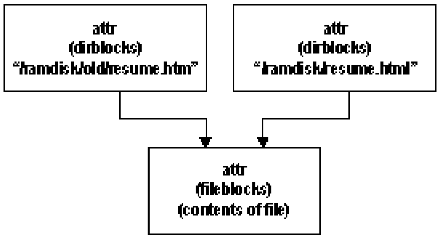


Figure 29: A hard link implemented as two different attributes structures pointing to the same file.

In this case, both `../resume.html` and `resume.htm` would be considered identical; there's no concept of “original” and “link” as there is with symlinks.

When the client calls *link()* or *symlink()* (or uses the command-line command `ln`), our RAM-disk resource manager's *c_link()* function will be called.

The *c_link()* function follows a similar code path as all of the other connect functions we've discussed so far (*c_open()* and *c_readlink()*), so once again we'll just focus on the differences:

```
int
cfs_c_link (resmgr_context_t *ctp, io_link_t *msg,
            RESMGR_HANDLE_T *handle, io_link_extra_t *extra)
{
  RESMGR_OCB_T  *ocb;
  des_t         parent, target;
  int           sts;
  char          *p, *s;
  struct        _client_info cinfo;

  if (sts = iofunc_client_info (ctp, 0, &cinfo)) {
    return (sts);
  }
```

Copyright © 2014, QNX Software Systems Limited

```
  if (connect_msg_to_attr (ctp, &msg -> connect, handle,
                           &parent, &target, &sts, &cinfo)) {
    return (sts);
  }
  if (target.attr) {
    return (EEXIST);
  }

  // 1) find out what type of link we are creating
  switch (msg -> connect.extra_type) {
  // process a hard link
  case  _IO_CONNECT_EXTRA_LINK:
    ocb = extra -> ocb;
    p = strdup (target.name);
    if (p == NULL) {
      return (ENOMEM);
    }
    // 2) add a new directory entry
    if (sts = add_new_dirent (parent.attr, ocb -> attr, p)) {
      free (p);
      return (sts);
    }
    // 3) bump the link count
    ocb -> attr -> attr.nlink++;
    return (EOK);

  // process a symbolic link
  case  _IO_CONNECT_EXTRA_SYMLINK:
    p = target.name;
    s = strdup (extra -> path);
    if (s == NULL) {
      return (ENOMEM);
    }
    // 4) create a symlink entry
    target.attr = cfs_a_mksymlink (parent.attr, p, NULL);
    if (!target.attr) {
      free (s);
      return (errno);
    }
    // 5) write data
    target.attr -> type.symlinkdata = s;
    target.attr -> attr.nbytes = strlen (s);
    return (EOK);

  default:
    return (ENOSYS);
  }

  return (_RESMGR_DEFAULT);
}
```

The following is the code walkthrough for creating hard or symbolic links:

1. The *extra_type* member of the connect message tells us what kind of link we're creating.
2. For a hard link, we create a new directory entry. The utility function *add_new_dirent()* is responsible for adjusting the *dirblocks* member of the attributes structure to hold another entry, performing whatever allocation is needed (except for allocating the name, which is done with the *strdup()*). Notice that we get an OCB as part of the *extra* parameter passed to us. This OCB's extended attributes structure is the resource that we're creating the hard link to (yes, this means that our *c_open()* would have been called before this — that's done automatically).
3. Since this is a hard link, we need to increment the link count of the object itself. Recall that we talked about named objects (the *dirblocks* array) and unnamed objects (the *fileblocks* member). The unnamed object is the actual entity that we bump the link count of, not the individual named objects.

4. If we're creating a symlink, call the utility function *cfs_a_mksymlink()*, which allocates a directory entry within the parent. Notice that in the symlink case, we don't get an OCB, but rather a pathname as part of the *extra* parameter.
5. Write the data into the extended attribute's *symlinkdata* member, and set the size to the length of the symlink content.

The *c_rename()* function

The functionality to perform a rename can be done in one of two ways. You can simply return ENOSYS, which tells the client's *rename()* that you don't support renaming, or you can handle it. If you do return ENOSYS, an end user might not notice it right away, because the command-line utility mv deals with that and copies the file to the new location and then deletes the original. For a RAM disk, with small files, the time it takes to do the copy and unlink is imperceptible. However, simply changing the name of a directory that has lots of large files will take a long time, even though all you're doing is changing the name of the directory!

In order to properly implement rename functionality, there are two interesting issues:

- Never rename such that the destination is a child of the source. An example of this is mv x x/a — before I fixed this bug you could do the above mv command and have the directory x simply vanish! That's because the internal logic effectively creates a hard link from the original x to the new x/a, and then unlinks the original x. Well, with x gone, you'd have a hard time going into the directory!
- All of our rename targets are on our filesystem, and are free of symlinks.

The rename logic is further complicated by the fact that we are dealing with two paths instead of just one. In the *c_link()* case, one of the pathnames was implied by either an OCB (hard link) or actually given (symlink) — for the symlink we viewed the second “pathname” as a text string, without doing any particular checking on it.

You'll notice this “two path” impact when we look at the code:

```
int
cfs_c_rename (resmgr_context_t *ctp, io_rename_t *msg,
              RESMGR_HANDLE_T *handle, io_rename_extra_t *extra)
{
  // source and destination parents and targets
  des_t   sparent, starget, dparent, dtarget;
  des_t   components [_POSIX_PATH_MAX];
  int     ncomponents;
  int     sts;
  char    *p;
  int     i;
  struct  _client_info cinfo;

  // 1) check for "initial subset" (mv x x/a) case
  i = strlen (extra -> path);
  if (!strncmp (extra -> path, msg -> connect.path, i)) {
    // source could be a subset, check character after
    // end of subset in destination
    if (msg -> connect.path [i] == 0
    || msg -> connect.path [i] == '/') {
      // source is identical to destination, or is a subset
      return (EINVAL);
    }
```

Copyright © 2014, QNX Software Systems Limited

```
  }

  // get client info
  if (sts = iofunc_client_info (ctp, 0, &cinfo)) {
    return (sts);
  }

  // 2) do destination resolution first in case we need to
  //    do a redirect or otherwise fail the request.
  if (connect_msg_to_attr (ctp, &msg -> connect, handle,
                           &dparent, &dtarget, &sts, &cinfo)) {
    return (sts);
  }

  // 3) if the destination exists, kill it and continue.
  if (sts != ENOENT) {
    if (sts == EOK) {
      if ((sts = cfs_rmnod (&dparent, dtarget.name,
                            dtarget.attr)) != EOK) {
        return (sts);
      }
    } else {
      return (sts);
    }
  }

  // 4) use our friend pathwalk() for source resolution.
  ncomponents = _POSIX_PATH_MAX;
  sts = pathwalk (ctp, extra -> path, handle, 0, components,
                  &ncomponents, &cinfo);

  // 5) missing directory component
  if (sts == ENOTDIR) {
    return (sts);
  }

  // 6) missing non-final component
  if (components [ncomponents].name != NULL && sts == ENOENT) {
    return (sts);
  }

  // 7) an annoying bug
  if (ncomponents < 2) {
    // can't move the root directory of the filesystem
    return (EBUSY);
  }

  starget = components [ncomponents - 1];
  sparent = components [ncomponents - 2];

  p = strdup (dtarget.name);
  if (p == NULL) {
    return (ENOMEM);
  }

  // 8) create new...
  if (sts = add_new_dirent (dparent.attr, starget.attr, p)) {
    free (p);
    return (sts);
  }
  starget.attr -> attr.nlink++;

  // 9) delete old
  return (cfs_rmnod (&sparent, starget.name, starget.attr));
}
```

The walkthrough is as follows:

1. The first thing we check for is that the destination is not a child of the source as described in the comments above. This is accomplished primarily with a *strncmp()*. Then we need to check that there's something other than nothing or a / after the string (that's because `mv x xa` is perfectly legal, even though it would be picked up by the *strncmp()*).

2. We do the “usual” destination resolution by calling *connect_msg_to_attr()*. Note that we use the *dparent* and *dtarget* (“d” for “destination”) variables.
3. The destination better not exist. If it does, we attempt to remove it, and if that fails, we return whatever error *cfs_rmnod()* returned. If it doesn't exist, or we were able to remove it, we continue on. If there was any problem (other than the file originally existing or not existing, e.g. a permission problem), we return the status we got from *connect_msg_to_attr()*.
4. This is the only time you see *pathwalk()* called apart from the call in *c_open()*. That's because this is the only connect function that takes two pathnames as arguments.
5. Catch missing intermediate directory components in the source.
6. Catch missing nonfinal components.
7. This was a nice bug, triggered by trying to rename `.` or the mount point. By simply ensuring that we're not trying to move the root directory of the filesystem, we fixed it. Next, we set up our “source” parent/target (*sparent* and *starget*).
8. This is where we perform the “link to new, unlink old” logic. We call *add_new_dirent()* to create a new directory entry in the destination parent, then bump its link count (there are now two links to the object we're moving).
9. Finally, we call *cfs_rmnod()* (see code below in discussion of *c_unlink()*) to remove the old. The removal logic decrements the link count.

The *c_mknod()* function

The functionality of *c_mknod()* is straightforward. It calls *iofunc_client_info_ext()* to get information about the client, then resolves the pathname using *connect_msg_to_attr()*, does some error checking (among other things, calls the helper function *iofunc_mknod()*), and finally creates the directory by calling the utility function *cfs_a_mkdir()*.

The *c_unlink()* function

To unlink an entry, the following code is used:

```
int
c_unlink (resmgr_context_t *ctp, io_unlink_t *msg,
          RESMGR_HANDLE_T *handle, void *reserved)
{
  des_t   parent, target;
  int     sts;
  struct  _client_info cinfo;

  if (sts = iofunc_client_info (ctp, 0, &cinfo)) {
    return (sts);
  }

  if (connect_msg_to_attr (ctp, &msg -> connect, handle,
                           &parent, &target, &sts, &cinfo)) {
    return (sts);
  }

  if (sts != EOK) {
    return (sts);
```

Copyright © 2014, QNX Software Systems Limited

```
  }

  // see below
  if (target.attr == handle) {
    return (EBUSY);
  }

  return (cfs_rmnod (&parent, target.name, target.attr));
}
```

The code implementing *c_unlink()* is straightforward as well — we get the client information and resolve the pathname. The destination had better exist, so if we don't get an EOK we return the error to the client. Also, it's a really bad idea (read: bug) to unlink the mount point, so we make a special check against the target attribute's being equal to the mount point attribute, and return EBUSY if that's the case. Note that QNX 4 returns the constant EBUSY, QNX Neutrino returns EPERM, and OpenBSD returns EISDIR. So, there are plenty of constants to choose from in the real world! I like EBUSY.

Other than that, the actual work is done in *cfs_rmnod()*, below.

```
int
cfs_rmnod (des_t *parent, char *name, cfs_attr_t *attr)
{
  int   sts;
  int   i;

  // 1) remove target
  attr -> attr.nlink--;
  if ((sts = release_attr (attr)) != EOK) {
    return (sts);
  }

  // 2) remove the directory entry out of the parent
  for (i = 0; i < parent -> attr -> nels; i++) {
    // 3) skip empty directory entries
    if (parent -> attr -> type.dirblocks [i].name == NULL) {
      continue;
    }
    if (!strcmp (parent -> attr -> type.dirblocks [i].name,
                 name)) {
      break;
    }
  }
  if (i == parent -> attr -> nels) {
    // huh.  gone.  This is either some kind of internal error,
    // or a race condition.
    return (ENOENT);
  }

  // 4) reclaim the space, and zero out the entry
  free (parent -> attr -> type.dirblocks [i].name);
  parent -> attr -> type.dirblocks [i].name = NULL;

  // 5) catch shrinkage at the tail end of the dirblocks[]
  while (parent -> attr -> type.dirblocks
         [parent -> attr -> nels - 1].name == NULL) {
    parent -> attr -> nels--;
  }

  // 6) could check the open count and do other reclamation
  //    magic here, but we don't *have to* for now...

  return (EOK);
}
```

Notice that we may not necessarily reclaim the space occupied by the resource! That's because the file could be in use by someone else. So the only time that it's appropriate to actually remove it is when the link count goes to zero, and that's checked for in the *release_attr()* routine as well as in the *io_close_ocb()* handler (below).

Here's the walkthrough:

1. This is the place where we decrement the link count. The function *release_attr()* will try to remove the file, but will abort if the link count isn't zero, instead deferring the removal until *io_close_ocb()* decides it's safe to do so.
2. The `for` loop scans the parent, attempting to find this directory entry by name.
3. Notice that here we must skip removed entries, as mentioned earlier.
4. Once we've found it (or errored-out), we free the space occupied by the *strdup()*'d name, and zero-out the *dirblocks* entry.
5. We attempt to do a little bit of optimization by compressing empty entries at the end of the *dirblocks* array. This `while` loop will be stopped by `..` which always exists.
6. At this point, you could do further optimizations *only* if the directory entry isn't in use.

The *io_close_ocb()* function

This naturally brings us to the *io_close_ocb()* function. In most resource managers, you'd let the default library function, *iofunc_close_ocb_default()*, do the work. However, in our case, we may need to free a resource. Consider the case where a client performs the following perfectly legal (and useful for things like temporary files) code:

```
fp = fopen ("/ramdisk/tmpfile", "r+");
unlink ("/ramdisk/tmpfile");
// do some processing with the file
fclose (fp);
```

We cannot release the resources for the `/ramdisk/tmpfile` until after the link count (the number of open file descriptors to the file) goes to zero.

The *fclose()* will eventually translate within the C library into a *close()*, which will then trigger our RAM disk's *io_close_ocb()* handler. Only when the count goes to zero can we free the data.

Here's the code for the *io_close_ocb()*:

```
int
cfs_io_close_ocb (resmgr_context_t *ctp, void *reserved,
                  RESMGR_OCB_T *ocb)
{
  cfs_attr_t    *attr;
  int           sts;

  attr = ocb -> attr;
  sts = iofunc_close_ocb (ctp, ocb, &attr -> attr);
  if (sts == EOK) {
    // release_attr makes sure that no-one is using it...
    sts = release_attr (attr);
  }
  return (sts);
}
```

Note the `attr -> attr` — the helper function *iofunc_close_ocb()* expects the normal, nonextended attributes structure.

Once again, we rely on the services of *release_attr()* to ensure that the link count is zero.

Here's the source for *release_attr()* (from `attr.c`):

```
int
release_attr (cfs_attr_t *attr)
{
  int   i;

  // 1) check the count
  if (!attr -> attr.nlink  && !attr -> attr.count) {
    // decide what kind (file or dir) this entry is...

    if (S_ISDIR (attr -> attr.mode)) {
      // 2) it's a directory, see if it's empty
      if (attr -> nels > 2) {
        return (ENOTEMPTY);
      }
      // 3) need to free "." and ".."
      free (attr -> type.dirblocks [0].name);
      free (attr -> type.dirblocks [0].attr);
      free (attr -> type.dirblocks [1].name);
      free (attr -> type.dirblocks [1].attr);

      // 4) release the dirblocks[]
      if (attr -> type.dirblocks) {
        free (attr -> type.dirblocks);
        free (attr);
      }
    } else if (S_ISREG (attr -> attr.mode)) {
      // 5) a regular file
      for (i = 0; i < attr -> nels; i++) {
        cfs_block_free (attr,
                        attr -> type.fileblocks [i].iov_base);
        attr -> type.fileblocks [i].iov_base = NULL;
      }
      // 6) release the fileblocks[]
      if (attr -> type.fileblocks) {
        free (attr -> type.fileblocks);
        free (attr);
      }
    } else if (S_ISLNK (attr -> attr.mode)) {
      // 7) a symlink, delete the contents
      free (attr -> type.symlinkdata);
      free (attr);
    }
  }
  // 8) return EOK if everything went well
  return (EOK);
}
```

Note that the definition of “empty” is slightly different for a directory. A directory is considered empty if it has just the two entries `.` and `..` within it.

You'll also note that we call *free()* to release all the objects. It's important that all the objects be allocated (whether via *malloc()*/*calloc()* for the *dirblocks* and *fileblocks*, or via *stdrup()* for the *symlinkdata*).

The code walkthrough is as follows:

1. We verify the *nlink* count in the attributes structure, as well as the *count* maintained by the resource manager library. Only if both of these are zero do we go ahead and process the deletion.
2. A directory is empty if it has exactly two entries (`.` and `..`).

3. We therefore free those two entries.
4. Finally, we free the *dirblocks* array as well as the attributes structure (*attr*) itself.
5. In the case of a file, we need to run through all of the *fileblocks* blocks and delete each one.
6. Finally, we free the *fileblocks* array as well as the attributes structure itself.
7. In the case of a symbolic link, we delete the content (the *symlinkdata*) and the attributes structure.
8. Only if everything went well do we return `EOK`. It's important to examine the return code and discontinue further operations; for example, if we're trying to release a non-empty directory, you can't continue the higher-level function (in *io_unlink()*, for example) of releasing the parent's entry.

The *io_devctl()* function

In normal (i.e. nonfilesystem) resource managers, the *io_devctl()* function is used to implement device control functions. We used this in the ADIOS data acquisition driver to, for example, get the configuration of the device.

In a filesystem resource manager, *io_devctl()* is used to get various information about the filesystem.

A large number of the commands aren't used for anything other than block I/O filesystems; a few are reserved for internal use only.

Here's a summary of the commands:

DCMD_BLK_PARTENTRY

Used by x86 disk partitions with harddisk-based filesystems.

DCMD_BLK_FORCE_RELEARN

Triggers a media reversioning and cache invalidation (for removable media). This command is also used to sync-up the filesystem if `chkfsys` (and other utilities) play with it “behind its back.”

DCMD_FSYS_STATISTICS and DCMD_FSYS_STATISTICS_CLR

Returns `struct fs_stats` (see `<sys/fs_stats.h>`). The `_CLR` version resets the counters to zero after returning their values. And, to quote QNX's filesystem guru John Garvey, “There is no supported utility which obtains this, the `fs_stats` is completely undocumented, and may/will change in the future, and certainly has changed in the past.”

DCMD_FSYS_STATVFS

Returns `struct statvfs` (see below for more details).

Copyright © 2014, QNX Software Systems Limited

DCMD_FSYS_MOUNTED_ON, DCMD_FSYS_MOUNTED_AT and DCMD_FSYS_MOUNTED_BY

Each returns 256 bytes of character data, giving information about their relationship to other filesystems. See the discussion below.

DCMD_FSYS_OPTIONS

Returns 256 bytes of character data. This can be used to return the command-line options that the filesystem was mounted with.

Mounting options

The DCMD_FSYS_MOUNTED_ON, DCMD_FSYS_MOUNTED_AT, and DCMD_FSYS_MOUNTED_BY commands allow traversal of the filesystem hierarchy by utilities (like df, dinit, and chkfsys) that need to move between the filesystem and the host/image of that filesystem.

For example, consider a disk with /dev/hd0t79 as a partition of /dev/hd0, mounted at the root (/), with a directory /tmp. The table below gives a summary of the responses for each command (shortened to just the two last letters of the command) for each entity:

Command	/dev/hd0t79	/	/tmp
ON	/dev/hd0	/dev/hd0t79	/dev/hd0t79
AT	/dev/hd0t79	/	/
BY	/		

ENODEV is returned when there is no such entity (for example, an ON query of /dev/hd0, or a BY query of /).

Basically:

- ON means “Who am I on top of?”
- BY means “Who is on top of me?”
- AT means “Where am I? Who is my owner?”

Filesystem statistics

The most important command that your filesystem should implement is the DCMD_FSYS_STATVFS. In our *io_devctl()* handler, this ends up calling the utility function *cfs_block_fill_statvfs()* (in lib/block.c):

```
void
cfs_block_fill_statvfs (cfs_attr_t *attr, struct statvfs *r)
{
  uint32_t      nalloc, nfree;
  size_t        nbytes;
  mpool_info (mpool_block, &nbytes, &r -> f_blocks, &nalloc,
              &nfree, NULL, NULL);

  // INVARIANT SECTION

  // file system block size
  r -> f_bsize = nbytes;

  // fundamental filesystem block size
  r -> f_frsize = nbytes;

  // total number of file serial numbers
  r -> f_files = INT_MAX;

  // file system id
  r -> f_fsid = 0x12345678;

  // bit mask of f_flag values
  r -> f_flag = 0;

  // maximum filename length
  r -> f_namemax = NAME_MAX;

  // null terminated name of target file system
  strcpy (r -> f_basetype, "cfs");

  // CALCULATED SECTION

  if (optm) {        // for system-allocated mem with a max

    // tot number of blocks on file system in units of f_frsize
    r -> f_blocks = optm / nbytes;

    // total number of free blocks
    r -> f_bfree = r -> f_blocks - nalloc;

    // total number of free file serial numbers (approximation)
    r -> f_ffree = r -> f_files - nalloc;

  } else if (optM) { // for statically-allocated mem with a max

    // total #blocks on file system in units of f_frsize
    r -> f_blocks = optM / nbytes;

    // total number of free blocks
    r -> f_bfree = nfree;

    // total number of free file serial numbers (approximation)
    r -> f_ffree = nfree;

  } else {           // for unbounded system-allocated memory

    // total #blocks on file system in units of f_frsize
    r -> f_blocks = nalloc + 1;

    // total number of free blocks
    r -> f_bfree = r -> f_blocks - nalloc;

    // total #free file serial numbers (an approximation)
    r -> f_ffree = r -> f_files - nalloc;

  }

  // MIRROR

  // number of free blocks available to non-priv. proc
  r -> f_bavail = r -> f_bfree;

  // number of file serial numbers available to non-priv. proc
  r -> f_favail = r -> f_ffree;
}
```

The reason for the additional complexity (as opposed to just stuffing the fields directly) is due to the command-line options for the RAM disk. The -m option lets the RAM disk slowly allocate memory for itself as it requires it from the operating system, up

Copyright © 2014, QNX Software Systems Limited

to a maximum limit. If you use the -M option instead, the RAM disk allocates the specified memory right up front. Using neither option causes the RAM disk to allocate memory as required, with no limit.

Some of the numbers are outright lies — for example, the *f_files* value, which is supposed to indicate the total number of file serial numbers, is simply set to `INT_MAX`. There is no possible way that we would ever use that many file serial numbers (`INT_MAX` is 9×10^{18})!

So, the job of *cfs_block_fill_statvfs()* is to gather the information from the block allocator, and stuff the numbers (perhaps calculating some of them) into the `struct statvfs` structure.

The *c_mount()* function

The last function we'll look at is the one that handles mount requests. Handling a mount request can be fairly tricky (there are lots of options), so we've just stuck with a simple version that does everything we need for the RAM disk.

When the RAM-disk resource manager starts up, there is no mounted RAM disk, so you must use the command-line `mount` command to mount one:

```
mount -Tramdisk /dev/ramdisk /ramdisk
```

The above command creates a RAM disk at the mount point `/ramdisk`.

The code is:

```
int
cfs_c_mount (resmgr_context_t *ctp, io_mount_t *msg,
             RESMGR_HANDLE_T *handle, io_mount_extra_t *extra)
{
  char        *mnt_point;
  char        *mnt_type;
  int         ret;
  cfs_attr_t  *cfs_attr;

  // 1) shortcuts
  mnt_point = msg -> connect.path;
  mnt_type = extra -> extra.srv.type;

  // 2) Verify that it is a mount request, not something else
  if (extra -> flags &
     (_MOUNT_ENUMERATE | _MOUNT_UNMOUNT | _MOUNT_REMOUNT)) {
    return (ENOTSUP);
  }

  // 3) decide if we should handle this request or not
  if (!mnt_type || strcmp (mnt_type, "ramdisk")) {
    return (ENOSYS);
  }

  // 4) create a new attributes structure and fill it
  if (!(cfs_attr = malloc (sizeof (*cfs_attr)))) {
    return (ENOMEM);
  }
  iofunc_attr_init (&cfs_attr -> attr, S_IFDIR | 0777,
                    NULL, NULL);

  // 5) initializes extended attribute structure
  cfs_attr_init (cfs_attr);

  // set up the inode
  cfs_attr -> attr.inode = (int) cfs_attr;
  // create "." and ".."
  cfs_a_mknod (cfs_attr, ".", S_IFDIR | 0755, NULL);
  cfs_a_mknod (cfs_attr, "..", S_IFDIR | 0755, NULL);

  // 6) attach the new pathname with the new value
  ret = resmgr_attach (dpp, &resmgr_attr, mnt_point,
                       _FTYPE_ANY, _RESMGR_FLAG_DIR,
                       &connect_func, &io_func,
                       &cfs_attr -> attr);
  if (ret == -1) {
    free (cfs_attr);
    return (errno);
  }

  return (EOK);
}
```

The code walkthrough is:

1. We create some shortcuts into the *msg* and *extra* fields. The *mnt_point* indicates where we would like to mount the RAM disk.. *mnt_type* indicates what kind of resource we are mounting, in this case we expect the string “ramdisk.”
2. We don't support any of the other mounting methods, like enumeration, unmounting, or remounting, so we just fail if we detect them.
3. We ensure that the type of mount request matches the type of our device (`ramdisk`).
4. We create a new attributes structure that represents the root directory of the new RAM disk, and we initialize it.
5. We also initialize the extended portion of the attributes structure, set up the *inode* member (see below), and create the `.` and `..` directories.
6. Finally, we call *resmgr_attach()* to create the new mount point in the pathname space.

The *inode* needs to be unique on a per-device basis, so the easiest way of doing that is to give it the address of the attributes structure.

Copyright © 2014, QNX Software Systems Limited

References

The following references apply to this chapter.

Header files

- `<dirent.h>` — contains the directory structure type used by *readdir()*.
- `<devctl.h>` — contains the definition for *devctl()*; also defines the component flags used to create a command.
- `<sys/dcmd_blk.h>` — contains the `FSYS` *devctl()* block commands.
- `<sys/disk.h>` — defines `partition_entry_t`.
- `<sys/dispatch.h>`, `<sys/iofunc.h>` — used by resource managers.
- `<sys/fs_stats.h>` — defines the `fs_stats` structure returned by the filesystem block command `DCMD_FSYS_STATISTICS`.

Functions

See the following functions in the QNX Neutrino *C Library Reference*:

- *_IO_SET_WRITE_NBYTES()* in the entry for *iofunc_write_verify()*
- *iofunc_check_access()*
- *iofunc_client_info_ext()*
- *iofunc_ocb_attach()*
- *iofunc_open()*
- *iofunc_read_verify()*
- *iofunc_write_verify()*
- *MsgReply()*
- *MsgReplyv()*
- *resmgr_msgreadv()*
- *S_ISDIR()* and *S_ISREG()*in the entry for *stat()*
- *SETIOV()*

Chapter 7
TAR Filesystem

The `.tar` file resource manager is similar to the previous chapter's RAM-disk filesystem manager, and they share quite a bit of code. However, it illustrates a different way of managing files — as a virtual filesystem map of a physical file.

This resource manager lets you `cd` into a `.tar` (or, through the magic of the `zlib` compression library, into a `.tar.gz`) file, and perform `ls`, `cp`, and other commands, as if you had gone through the trouble of (optionally uncompressing and) unpacking the `.tar` file into a temporary directory. (That's a small lie; a directory entry can't be both a file and a directory at the same time, so you can't `cd` into the `.tar` file, but the resource manager creates a `.tar.dir` file that you *can* `cd` into.)

In the *Filesystems* appendix, I present background information about filesystem implementation within the resource manager framework. You might want to refer to that before, during, or after you read this chapter.

I assume that you've read the *RAM-disk Filesystem* (p. 135) chapter, because anything that's common between these two resource managers isn't repeated here.

Requirements

The requirements for this project stemmed from my desire to "just `cd` into a `.tar` file." I presented a similar, but much more basic, version of the `.tar` filesystem when I wrote the *Writing a Resource Manager* course for QNX Software Systems. In that course, students were asked to parse a `.tar` file (with lots of helper code already in place), and write a tiny virtual filesystem that would work like the resource manager presented in this chapter. Unfortunately, due to the limitations on class time, the resource manager presented in that class was very basic — it handled only one type of `.tar` file (the POSIX standard one, not the GNU one, which has a slightly different format), it mounted only one `.tar` at a time, and — the most severe limitation — the `.tar` file could have files only in the "current" directory (i.e. it didn't support subdirectories).

This resource manager remedies all that.

Copyright © 2014, QNX Software Systems Limited

Design

To understand the design of this resource manager, it's necessary to understand the basics of a `.tar` file, and the two common formats.

Way back in the dim dark early days of UNIX (I think it must have been in the roaring twenties or something), huge 30cm (1 foot) diameter 9-track magnetic tapes were all the rage. In order to transport data from one machine to another, or to make backups, many files would be archived onto these tapes. The name "TAR" stood for *T*ape *AR*chive, and the format of the `.tar` file was an attempt to accommodate tape media. The tapes worked best in block mode; rather than write individual bytes of data, the system would read or write 512-byte (or other sized) blocks of data. When you consider the technology of the day, it made a lot of sense — here was this moving magnetic tape that passed over a head at a certain speed. While the tape was moving, it was good to pump data to (or from) the tape. When the tape stopped, it would be awkward to position it exactly into the middle of a block of data, so most tape operations were block-oriented (with the notable exception of incremental tape drives, but we won't go into that here). A typical magnetic tape could hold tens to hundreds of megabytes of data, depending on the density (number of bits per inch).

The `.tar` format, therefore, was block-oriented, and completely sequential (as opposed to random access). For every file in the archive, there was a block-aligned header describing the file (permissions, name, owner, date and time, etc.), followed by one or more blocks consisting of the file's data.

Creating a `.tar` file

So following along in this command-line session, I'll show you the resulting `.tar` file:

```
# ls -la
total 73
drwxrwxr-x  2 root      root   4096 Aug 17 17:31 ./
drwxrwxrwt  4 root      root   4096 Aug 17 17:29 ../
-rw-rw-r--  1 root      root   1076 Jan 14  2003 io_read.c
-rw-rw-r--  1 root      root    814 Jan 12  2003 io_write.c
-rw-rw-r--  1 root      root   6807 Feb 03  2003 main.c
-rw-rw-r--  1 root      root  11883 Feb 03  2003 tarfs.c
-rw-rw-r--  1 root      root    683 Jan 12  2003 tarfs.h
-rw-rw-r--  1 root      root   6008 Jan 15  2003 tarfs_io_read.c

# tar cvf x.tar *
io_read.c
io_write.c
main.c
tarfs.c
tarfs.h
tarfs_io_read.c

# ls -l x.tar
-rw-rw-r--  1 root      root  40960 Aug 17 17:31 x.tar
```

Here I've taken some of the source files in a directory and created a `.tar` file (called `x.tar`) that ends up being `40960` bytes — a nice multiple of `512` bytes, as we'd expect.

Each of the files is prefixed by a header in the `.tar` file, followed by the file content, aligned to a `512`-byte boundary.

This is what each header looks like:

Offset	Length	Field Name
0	100	*name*
100	8	*mode*
108	8	*uid*
116	8	*gid*
124	12	*size*
136	12	*mtime*
148	8	*chksum*
156	1	*typeflag*
157	100	*linkname*
257	6	*magic*
263	2	*version*
265	32	*uname*
297	32	*gname*
329	8	*devmajor*
337	8	*devminor*
345	155	*prefix*
500	11	*filler*

Here's a description of the fields that we're interested in for the filesystem (all fields are ASCII octal unless noted otherwise):

name

The name of the stored entity (plain ASCII).

mode

The mode: read, write, execute permissions, as well as what the entity is (a file, symlink, etc.).

Copyright © 2014, QNX Software Systems Limited

uid

The user ID.

gid

The group ID.

size

The size of the resource (symlinks and links get a `0` size).

typeflag

POSIX says one thing, GNU says another. Under POSIX, this is supposed to be one of the single characters "g," "x," or "0." Under GNU, this is one of the single ASCII digits zero through seven, or an ASCII `NUL` character, indicating different types of entities. Sigh — "The nice thing about standards is there are so many to choose from."

mtime

The modification time.

linkname

The name of the file that this file is linked to (or blank if not linked), in plain ASCII.

We've skipped a bunch of fields, such as the *checksum*, because we don't need them for our filesystem. (For the checksum, for example, we're simply *assuming* that the file has been stored properly — in the vast majority of cases, it's not actually on an antique 9-track tape — so data integrity shouldn't be a problem!)

What I meant above by "ASCII octal" fields is that the value of the number is encoded as a sequence of ASCII digits in base 8. Really.

For example, here's the very first header in the sample `x.tar` that we created above (addresses on the left-hand side, as well as the dump contents, are in hexadecimal, with printable ASCII characters on the right-hand side):

```
0000000 69 6f 5f 72 65 61 64 2e 63 00 00 00 00 00 00 00 io_read.c.......
0000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0000030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0000040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0000050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0000060 00 00 00 00 30 31 30 30 36 36 34 00 30 30 30 30 ....0100664.0000
0000070 30 30 30 00 30 30 30 30 30 30 30 00 30 30 30 30 000.0000000.0000
0000080 30 30 30 32 30 36 34 00 30 37 36 31 31 31 34 31 0002064.07611141
0000090 34 36 35 00 30 31 31 33 33 34 00 20 30 00 00 00 465.011334..0...
00000A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

```
0000100 00 75 73 74 61 72 20 20 00 72 6f 6f 74 00 00 00 .ustar...root...
0000110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0000120 00 00 00 00 00 00 00 00 00 72 6f 6f 74 00 00 00 .........root...
0000130 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0000140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0000150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0000160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0000170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0000180 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0000190 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00001A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00001B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00001C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00001D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00001E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00001F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

Here are the fields that we're interested in for our `.tar` filesystem:

Offset 0

The name of the entity, in this case `io_read.c`.

Offset 100 (`0x64` hex)

The ASCII octal digits `0100664` representing `S_IFREG` (indicating this is a regular file), with a permission mode of `0664` (indicating it's readable by everyone, but writable by the owner and group only).

Offset 108 (`0x6C`)

The ASCII octal digits `0000000` representing the user ID (in this case, `0`, or `root`).

Offset 116 (`0x74`)

The ASCII octal digits `0000000` representing the group ID (`0000000`, or group `0`).

Offset 124 (`0x7C`)

The ASCII octal digits `00000002064` (or decimal `1076`) representing the size of the entity. (This does present a limit to the file size of `77777777777` octal, or 8 gigabytes — not bad for something invented in the days of hard disks the size of washing machines with capacities of tens of megabytes!).

Offset 136 (`0x88`)

The ASCII octal digits `07611141465` (or decimal `1042596661` seconds after January 1st, 1970, which really converts to “Tue Jan 14 21:11:01 EST 2003.”)

The one interesting wrinkle has to do with items that are in subdirectories.

Depending on how you invoked `tar` when the archive was created, you may or may not have the directories listed individually within the `.tar` file. What I mean is that if you add the file `dir/spud.txt` to the archive, the question is, is there a `tar` header corresponding to `dir`? In some cases there will be, in others there won't, so

Copyright © 2014, QNX Software Systems Limited

our `.tar` filesystem will need to be smart enough to create any intermediate directories that aren't explicitly mentioned in the headers.

Note that in *all* cases the *full* pathname is given; that is, we will always have `dir/spud.txt`. We never have a header for the directory for `dir` followed by a header for the file `spud.txt`; we'll always get the full path to each and every component listed in a `.tar` file.

Let's stop and think about how this resource manager compares to the RAM-disk resource manager in the previous chapter. If you squint your eyes a little bit, and ignore a few minor details, you can say they are almost identical. We need to:

- set up data structures to hold directories, files, and symlinks
- set up all the administration that goes along with managing these entities
- search directories
- satisfy the client's *readdir()* calls by returning directory entries
- mount the root filesystem at arbitrary points.

The only thing that's really different is that instead of storing the file contents in RAM, we're storing them on disk! (The fact that this is a read-only filesystem isn't really a difference, it's a *subset*.)

The code

Let's now turn our attention to the code in the `.tar` filesystem. We'll begin by looking at the data structures that are different from the ones in the RAM disk, and then we'll look at the C modules that are different.

The structures

We still use an extended attributes structure in the `.tar` filesystem, because we still need to store additional information (like the contents of directories and symlinks). The main modification is the way we store files — we don't. We store only a *reference* to the file, indicating where in the `.tar` file the actual file begins, and its name. (The base attributes structure still stores the usual stuff: permissions, file times, etc., and most importantly, the size.)

```
typedef struct fes_s
{
    char                *name;
    off_t               off;
}   fes_t;

typedef struct cfs_attr_s
{
    iofunc_attr_t       attr;

    int                 nels;
    int                 nalloc;
    union {
        struct des_s    *dirblocks;
//      iov_t           *fileblocks;
        fes_t           vfile;
        char            *symlinkdata;
    } type;
} cfs_attr_t;
```

As you can see, the `cfs_attr_t` is almost identical to the RAM-disk structure. In fact, I left the *fileblocks* member commented-out to show the evolution from one source base to the next.

So our data for the file is now stored in a `fes_t` data type. All that the `fes_t` data type contains is the name of the `.tar` file (where the data is actually stored), and the offset into the `.tar` — we don't need to store the size of the file, because that's already stored normally in the plain (not extended) attributes structure.

The other difference from the RAM-disk filesystem is that we store the `fes_t` directly, rather than a pointer to it (as we did with the *fileblocks* IOV array). That's because the `fes_t` doesn't ever grow, and storing a 4-byte pointer to a tiny, fixed-size *malloc()*'d data region will take more space than just storing the 8-byte item directly.

The functions

Any functions that we don't mention here are either exactly the same as the RAM-disk version, or have such minor changes that they're not worth mentioning.

Copyright © 2014, QNX Software Systems Limited

Overall operation is the same as the RAM-disk resource manager — we parse the command-line options, set up some data structures, and enter the resource manager main loop.

The *c_mount()* function

The first function called is the *c_mount()* function, which is responsible for accepting the name of a `.tar` file to operate on, and where to manifest its contents. There's a “mount helper” utility (the file `m_main.c`) that we'll look at later.

The beginning part of *c_mount()* is the same as the RAM disk, so we'll just point out the tail-end section that's different:

```
...

    // 1) allocate an attributes structure
    if (!(cfs_attr = malloc (sizeof (*cfs_attr)))) {
        return ENOMEM;
    }

    // 2) initialize it
    iofunc_attr_init (&cfs_attr -> attr,
                      S_IFDIR | 0555, NULL, NULL);
    cfs_attr_init (cfs_attr);
    cfs_attr -> attr.inode = (int) cfs_attr;
    cfs_a_mknod (cfs_attr, ".", S_IFDIR | 0555, NULL);
    cfs_a_mknod (cfs_attr, "..", S_IFDIR | 0555, NULL);

    // 3) load the tar file
    if (ret = analyze_tar_file (cfs_attr, spec_name)) {
        return (ret);
    }

    // 4) Attach the new pathname with the new value
    ret = resmgr_attach (dpp, &resmgr_attr, mnt_point,
                         _FTYPE_ANY, _RESMGR_FLAG_DIR,
                         &connect_func, &io_func,
                         &cfs_attr -> attr);
    if (ret == -1) {
        free (cfs_attr);
        return (errno);
    }
    return (EOK);
}
```

The code walkthrough is as follows:

1. Just like in the RAM disk, we initialize our attributes structure. (This is for synchronization with the RAM-disk source; after this point we diverge.)
2. The initialization is almost the same as the RAM disk, except we use mode `0555` because we are a read-only filesystem. We *could* lie and show `0777` if we wanted to, but we'd just run into grief later on (see below).
3. All of the tar-specific work is done in *analyze_tar_file()*, which we'll look at next. The function can return a non-zero value if it detects something it doesn't like — if that's the case, we just return it to the resource-manager framework.
4. Finally, we attach the mount point using *resmgr_attr()* as per normal (same as the RAM disk).

Part of the grief mentioned in step 2 above actually turns out to have a useful side-effect. If you were to reinstate the RAM-disk portion of the extended attributes structure — even though it's not implemented in the current filesystem manager — you could implement a somewhat “modifiable” .tar filesystem. If you ever went to write to a file, the resource manager would copy the data from the .tar version of the file, and then use the *fileblocks* member rather than the *vfile* member for subsequent operations. This might be a fairly simple way of making a few small modifications to an existing tar file, without the need to uncompress and untar the file. You'd then need to re-tar the entire directory structure to include your new changes. Try it and see!

The *analyze_tar_file()* function

At the highest level, the *analyze_tar_function()* function opens the .tar file, processes each file inside by calling *add_tar_entry()*, and then closes the .tar file. There's a wonderful library called zlib, which lets us open even compressed files and pretend that they are just normal, uncompressed files. That's what gives us the flexibility to open either a .tar or a .tar.gz file with no additional work on our part. (The limitation of the library is that seeking may be slow, because decompression may need to occur.)

```
int
analyze_tar_file (cfs_attr_t *a, char *fname)
{
    gzFile  fd;
    off_t   off;
    ustar_t t;
    int     size;
    int     sts;
    char    *f;

    // 1) the .tar (or .tar.gz) file must exist :-)
    if ((fd = gzopen (fname, "r")) == NULL) {
        return (errno);
    }

    off = 0;
    f = strdup (fname);

    // 2) read the 512-byte header into "t"
    while (gzread (fd, &t, sizeof (t)) > 0 && *t.name) {
        dump_tar_header (off, &t);

        // 3) get the size
        sscanf (t.size, "%o", &size);
        off += sizeof (t);

        // 4) add this entry to the database
        if (sts = add_tar_entry (a, off, &t, f)) {
            gzclose (fd);
            return (sts);
        }

        // 5) skip the data for the entry
        off += ((size + 511) / 512) * 512;
        gzseek (fd, off, SEEK_SET);
    }
    gzclose (fd);

    return (EOK);
}
```

The code walkthrough is:

1. The zlib library makes things look just like an *fopen()* call.

Copyright © 2014, QNX Software Systems Limited

2. We read each header into the variable *t*, and optionally dump the header if case debug is enabled.
3. We read the ASCII octal size of the file following the header, then store it.
4. The real work is done in *add_tar_entry()*.
5. The best part is that we skip the file content, which makes loading fast.

In step 5 we skip the file content. I'm surprised that not all of today's `tar` utilities do this when they're dealing with files — doing a `tar tvf` to get a listing of the tar file takes *forever* for huge files!

The *add_tar_entry()* function

Header analysis is done by the *add_tar_entry()* function. If this looks a little bit familiar it's because the code is based on the *pathwalk()* function from the RAM-disk resource manager, with logic added to process the `tar` file header.

```
static int
add_tar_entry (cfs_attr_t *a, off_t off, ustar_t *t, char *tarfile)
{
  des_t   output [_POSIX_PATH_MAX];
  int     nels;
  char    *p;
  int     i;
  int     mode;

  // 1) first entry is the mount point
  output [0].attr = a;
  output [0].name = NULL;

  // 2) break apart the pathname at the slashes
  nels = 1;
  for (p = strtok (t -> name, "/"); p; p = strtok (NULL, "/"), nels++) {
    if (nels >= _POSIX_PATH_MAX) {
      return (E2BIG);
    }
    output [nels].name = p;
    output [nels].attr = NULL;
  }

  // 3) analyze each pathname component
  for (i = 1; i < nels; i++) {

    // 4) sanity check
    if (!S_ISDIR (output [i - 1].attr -> attr.mode)) {
      return (ENOTDIR);       // effectively an internal error
    }

    // 5) check to see if the element exists...
    if (!(output [i].attr = search_dir (output [i].name,
                                        output [i-1].attr))) {
      mode = parse_mode (t);

      // 6) intermediate directory needs to be created...
      if (S_ISDIR (mode) || (i + 1 < nels)) {
        output [i].attr = cfs_a_mkdir (output [i - 1].attr,
                                       output [i].name, NULL);
        tar_to_attr (t, &output [i].attr -> attr);
        // kludge for implied "directories"
        if (S_ISREG (output [i].attr -> attr.mode)) {
          output [i].attr -> attr.mode =
            (output [i].attr -> attr.mode & ~S_IFREG) | S_IFDIR;
        }

      // 7) add a regular file
      } else if (S_ISREG (mode)) {
        output [i].attr = cfs_a_mkfile (output [i - 1].attr,
                                        output [i].name, NULL);
        tar_to_attr (t, &output [i].attr -> attr);
        output [i].attr -> nels = output [i].attr -> nalloc = 1;
        output [i].attr -> type.vfile.name = tarfile;
        output [i].attr -> type.vfile.off = off;

      // 8) add a symlink
      } else if (S_ISLNK (mode)) {
        output [i].attr = cfs_a_mksymlink (output [i - 1].attr,
                                           output [i].name, NULL);
        tar_to_attr (t, &output [i].attr -> attr);
        output [i].attr -> type.symlinkdata = strdup (t -> linkname);
        output [i].attr -> attr.nbytes = strlen (t -> linkname);

      } else {
        // code prints an error message here...
        return (EBADF);         // effectively an internal error
      }
    }
  }

  return (EOK);
}
```

The code walkthrough is:

1. Just as in the RAM disk's *pathwalk()*, the first entry in the *output* array is the mount point.
2. And, just as in the RAM disk's *pathwalk()*, we break apart the pathname at the / delimiters.
3. Then we use the `for` loop to analyze each and every pathname component.
4. This step is a basic sanity check — we're assuming that the `.tar` file came from a normal, sane, and self-consistent filesystem. If that's the case, then the parent of an entry will always be a directory; this check ensures that that's the case. (It's a small leftover from *pathwalk()*.)
5. Next we need to see if the component exists. Just as in *pathwalk()*, we need to be able to verify that each component in the pathname exists, not only for sanity reasons, but also for implied intermediate directories.
6. In this case, the component does not exist, and yet there are further pathname components after it! This is the case of the implied intermediate directory; we simply create an intermediate directory and “pretend” that everything is just fine.
7. In this case, the component does not exist, and it's a regular file. We just go ahead and create the regular file. Note that we don't check to see if there are any components following this regular file, for two reasons. In a sane, consistent filesystem, this wouldn't happen anyway (so we're really not expecting this case in the `.tar` file). More importantly, that error will be caught in step 4 above, when we go to process the next entry and discover that its parent isn't a directory.
8. In this case, we create a symbolic link.

The *io_read()* function and related utilities

The `.tar` filesystem's *io_read()* is the standard one that we've seen in the RAM disk — it decides if the request is for a file or a directory, and calls the appropriate function.

The `.tar` filesystem's *tarfs_io_read_dir()* is the exact same thing as the RAM disk version — after all, the directory entry structures in the extended attributes structure are identical.

The only function that's different is the *tarfs_io_read_file()* function to read the data from the `.tar` file on disk.

```
int
tarfs_io_read_file (resmgr_context_t *ctp, io_read_t *msg,
iofunc_ocb_t *ocb)
{
  int     nbytes;
  int     nleft;
  iov_t   *iovs;
  int     niovs;
  int     i;
  int     pool_flag;
  gzFile  fd;

  // we don't do any xtypes here...
  if ((msg -> i.xtype & _IO_XTYPE_MASK) != _IO_XTYPE_NONE) {
    return (ENOSYS);
  }

  // figure out how many bytes are left
  nleft = ocb -> attr -> attr.nbytes - ocb -> offset;

  // and how many we can return to the client
  nbytes = min (nleft, msg -> i.nbytes);

  if (nbytes) {

    // 1) open the on-disk .tar file
    if ((fd = gzopen (ocb -> attr -> type.vfile.name, "r")) == NULL) {
      return (errno);
    }

    // 2) calculate number of IOVs required for transfer
    niovs = (nbytes + BLOCKSIZE - 1) / BLOCKSIZE;
    if (niovs <= 8) {
      iovs = mpool_malloc (mpool_iov8);
      pool_flag = 1;
    } else {
      iovs = malloc (sizeof (iov_t) * niovs);
      pool_flag = 0;
    }
    if (iovs == NULL) {
      gzclose (fd);
      return (ENOMEM);
    }

    // 3) allocate blocks for the transfer
    for (i = 0; i < niovs; i++) {
      SETIOV (&iovs [i], cfs_block_alloc (ocb -> attr), BLOCKSIZE);
      if (iovs [i].iov_base == NULL) {
        for (--i ; i >= 0; i--) {
          cfs_block_free (ocb -> attr, iovs [i].iov_base);
        }
        gzclose (fd);
        return (ENOMEM);
      }
    }

    // 4) trim last block to correctly read last entry in a .tar file
    if (nbytes & BLOCKSIZE) {
      iovs [niovs - 1].iov_len = nbytes & BLOCKSIZE;
    }

    // 5) get the data
    gzseek (fd, ocb -> attr -> type.vfile.off + ocb -> offset, SEEK_SET);
    for (i = 0; i < niovs; i++) {
      gzread (fd, iovs [i].iov_base, iovs [i].iov_len);
    }
    gzclose (fd);
    // return it to the client
    MsgReplyv (ctp -> rcvid, nbytes, iovs, i);

    // update flags and offset
    ocb -> attr -> attr.flags |= IOFUNC_ATTR_ATIME
                              | IOFUNC_ATTR_DIRTY_TIME;
    ocb -> offset += nbytes;
    for (i = 0; i < niovs; i++) {
      cfs_block_free (ocb -> attr, iovs [i].iov_base);
    }
    if (pool_flag) {
      mpool_free (mpool_iov8, iovs);
    } else {
      free (iovs);
    }
  } else {
    // nothing to return, indicate End Of File
    MsgReply (ctp -> rcvid, EOK, NULL, 0);
  }

  // already done the reply ourselves
  return (_RESMGR_NOREPLY);
}
```

Many of the steps here are common with the RAM disk version, so only steps 1 through 5 are documented here:

1. Notice that we keep the `.tar` on-disk file closed, and open it only as required. This is an area for improvement, in that you might find it slightly faster to have a certain cache of open `.tar` files, and maybe rotate them on an LRU-basis. We keep it closed so we don't run out of file descriptors; after all, you can mount hundreds (up to 1000 — a QNX Neutrino limit) of `.tar` files with this resource manager (see the note below).
2. We're still dealing with blocks, just as we did in the RAM-disk filesystem, because we need a place to transfer the data from the disk file. We calculate the number of IOVs we're going to need for this transfer, and then allocate the *iovs* array.
3. Next, we call *cfs_block_alloc()* to get blocks from the block allocator, then we bind them to the *iovs* array. In case of a failure, we free all the blocks and fail ungracefully. A better failure mode would have been to shrink the client's request size to what we can handle, and return that. However, when you analyze this, the typical request size is 32k (8 blocks), and if we don't have 32k lying around, then we might have bigger troubles ahead.
4. The last block is probably *not* going to be exactly 4096 bytes in length, so we need to trim it. Nothing bad would happen if we were to *gzread()* the extra data into the end of the block — the client's transfer size is limited to the size of the resource stored in the attributes structure. So I'm just being extra paranoid.
5. And in this step, I'm being completely careless; we simply *gzread()* the data with no error-checking whatsoever into the blocks! `:-)`

The rest of the code is standard; return the buffer to the client via *MsgReplyv()*, update the access flags and offset, free the blocks and IOVs, etc.

In step 1, I mentioned a limit of 1000 open file descriptors. This limit is controlled by the -F parameter to `procnto` (the kernel). In version 6.2.1 of QNX Neutrino, whatever value you pass to -F is the maximum (and default), and you cannot go higher than that value. In version 6.3 of QNX Neutrino, whatever value you pass to -F is the default, and you *can* go higher. You can change the value (to be lower in 6.2.1, or to be lower or higher in 6.3) via the *setrlimit()* function, using the RLIMIT_NOFILE resource constant.

The mount helper program

One of the things that makes the .tar filesystem easy to use is the mount helper program. For example, I snarf newsgroups and store them in compressed .tar archives. So, I might have directories on my machine like /news/comp/os/qnx. As I accumulate articles, I store them in batches, say every 1000 articles. Within that directory, I might have files like 003xxx.tar.gz and 004xxx.tar.gz which are two compressed .tar files containing articles 3000 through 3999 and 4000 through 4999.

Using the .tar filesystem resource manager, I'd have to specify the following command lines to mount these two files:

```
mount -T tarfs /news/comp/os/qnx/003xxx.tar.gz 003xxx.tar.dir
mount -T tarfs /news/comp/os/qnx/004xxx.tar.gz 004xxx.tar.dir
```

That's not too bad for a handful of files, but it gets to be a real pain when you have hundreds of files.

The find fans will of course realize that it could be done “easily” using find:

```
find /news/comp/os/qnx -type f -name "*tar.gz" \
    -exec "mount -T tarfs {} {3}.dir"
```

Which is ever-so-slightly different (the {3}.dir creates an absolute path rather than a relative path, but that's okay).

However, for casual use, it would be really nice to just say:

```
mount_tarfs -m *.tar.gz
```

to mount all of the compressed archives in the current directory.

This is the job of the mount helper program. The mount helper is actually used in two modes. In one mode (as seen above) it's invoked standalone. In the second mode, it's invoked automatically by QNX Neutrino's mount command:

```
mount -T tarfs source.tar[.gz] [dirname]
```

The code for the mount_tarfs is remarkably simple: *main()* does the usual call to the option-processing function, and then we determine if we are mounting a single file or multiple files. We optionally generate the target's extension if required, and call the library function *mount()*.

Variations on a theme

The original RAM-disk filesystem was used to develop a framework that would support enough of the resource-manager functions to be useful. This framework was then adapted to the `.tar` filesystem, with very few changes.

In this section, I'd like to give you some further ideas about what can be done with the framework and with filesystems (virtual or otherwise) in general. There are several filesystems discussed here:

- Virtual filesystem for USENET news (VFNews)
- Strange and unusual filesystem
- Secure filesystem
- Line-based filesystem

Virtual filesystem for USENET news (VFNews)

I have created a virtual filesystem for USENET news (“VFNews”) under QNX 4, but I haven't ported it to QNX Neutrino.

I'll describe what VFNews does and how it works. Most news services these days are all NNTP-based high-speed article-at-a-time services, rather than the traditional bulk systems that they used to be up until the mid 1990s, so the actual end-product is of limited interest. However, you'll see an interesting way of increasing the efficiency of a processing method by *several* orders of magnitude.

How does USENET news work?

People from around the world post messages (*articles*) to the various newsgroups. Their news system then distributes these articles to neighboring machines. Neighboring machines distribute the articles to their neighbors, and so on, until the articles have propagated all the way around the world. Machines check the incoming articles to see if they already have a copy, and quietly delete duplicates.

Historically, a program like **cnews** was responsible for the on-disk management of the news articles, and performed two major operations:

1. Get the news articles, and store them on disk
2. Delete old news articles.

Let's look at a typical system. As each article arrives (whether by UUCP, NNTP, or some other means), the article's “header” portion is scanned, and the news software determines where (i.e. into which newsgroups) that article should be stored.

A long time ago, when there wasn't all that much news traffic, it seemed like a good idea to just store one article per file. The newsgroup names got converted into pathnames, and everything was simple. For example, if I had an incoming article for comp.os.qnx, I would pick the next article number for that newsgroup (say 1143), and store the new article in a file called /var/spool/news/comp/os/qnx/1143. (The /var/spool/news part is just the name of the directory where all of the incoming news articles live — it's up to each site to determine which directory that is, but /var/spool/news is common.)

The next article that came in for that newsgroup would go into /var/spool/news/comp/os/qnx/1144, and so on.

So why is this a problem?

There are a number of reasons why this isn't an ideal solution. Articles can arrive in any order — we're not always going to get all of the articles for comp.os.qnx, then all of the articles for comp.os.rsx11, then comp.os.svr4, etc. This means that as the articles arrive, the news storage program is creating files in an ad-hoc manner, all over the disk. Not only that, it's creating from a few hundred thousand to many millions of files per day, depending on the size of the feed! (This works out to tens to hundreds or more files per second! The poor disk — and filesystem — is getting quite a workout.)

Given the volume of news these days, even terabyte-sized disks would fill up fairly quickly. So, all news systems have an expiry policy, which describes how long various articles hang around before being deleted. This is usually tuned based on the newsgroup, and can range from a few days to weeks or even months for low-traffic newsgroups. With current implementations, the expiry processing takes a significant amount of time; sometimes, the expiry processing will take so long that it appears that the machine is doing nothing *but* expiring!

The problem is that *tons* of files are being created in random places on the disk each second, and also roughly the same number of files being deleted, from different random places each second. This is suboptimal, and is exacerbated by the fact that each article even gets copied around a few times before ending up in its final location.

How can this possibly be made better?

As you are no doubt suspecting, we can do much better by knowing something about the problem domain, and putting our RAM-disk filesystem framework to good use.

One of the requirements, though, is that we still maintain the same directory structure as the old systems (i.e. the /var/spool/news structure).

Operation

As mentioned above, there are two main things that happen in the news-processing world: news comes in, and news expires.

The first trick is to realize that most news expires unread, and all news expires at some point. So by looking at the header for the article, we can determine when the article will expire, and place it in a file with all the other articles that will expire at the same time:

```
/var/spool/vfnews/20030804.news
/var/spool/vfnews/20030805.news
/var/spool/vfnews/20030806.news
```

Here we have three files, with the filenames representing the date that the batch of news articles expires, ranging from August 4, 2003 to August 6, 2003.

All we need to do is make a virtual filesystem that knows how to index into a real on-disk file, at a certain offset, for a certain length, and present those contents as if they were the real contents of the virtual file. Well, we've just done the *exact* same thing with the `.tar` filesystem! Effectively, (with a few optimizations) what we're doing is very similar to creating several different `.tar` files, and placing articles into those files. The files are named based on when they expire. When an article is added to the end of a file, we track its name (like `/var/spool/news/comp/os/qnx/1145`), the name of the bulk file we put it into, and its offset and size. (Some of the optimizations stem from the fact that we don't need to be 512-byte aligned, and that we don't need a header in the article's storage file.)

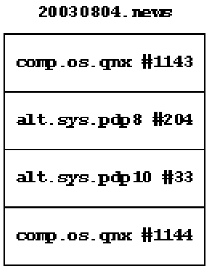


Figure 30: Articles from different newsgroups stored in a bulk file that expires on August 4, 2003.

When the time comes to expire old news, we simply remove the bulk file and any in-memory references to it.

This is the real beauty of this filesystem, and why we gain so much improvement from doing things this way than the original way. We've changed our disk access from writing tons of tiny files all over the disk to now writing sequentially to a few very large files. For expiration, we've done the same thing — we've changed from deleting tons of tiny files all over the disk to deleting one very large file when it expires. (The in-memory virtual filesystem deletion of the files is very fast; orders of magnitude faster than releasing blocks on disk.)

To give you an idea of the efficiency of this approach, consider that this system was running on a QNX 4 box in the early 1990s, with a 386 at 20 MHz, and a “full” news feed (19.2 kilobaud Trailblazer modem busy over 20 hours per

Copyright © 2014, QNX Software Systems Limited

day). When running with cnews, the hard disk was constantly thrashing. When running with VFNews, there was no disk activity, except every five seconds when a cache was flushed. In those days, system administrators would run cnews "expiry" once a day because of the overhead. I was able to run it once per hour with no noticeable impact on the CPU/disk! Also, ISPs would replace their hard disks every so often due to the amount of disk thrashing that was occurring.

Strange and unusual filesystems

Another very interesting thing that you can do with filesystems is completely abuse the assumptions about what a filesystem can do, and thus come up with strange and unusual filesystems. (Cue Twilight Zone theme music...)

The easiest entity to abuse is the symbolic link. Effectively, it's a back door into the filesystem. Since you control what happens in the *c_link()* entry point when the symlink is created, you control the interpretation of the symlink. This gives tremendous potential for abuse — as I like to quote Dr Seuss:

> Then he got an idea!
>
> An awful idea!
>
> *The Grinch got a wonderful, awful idea!*

Indexed filesystem

For example, we could implement what can generally be called an indexed filesystem. This shares the characteristics of the .tar and VFNews filesystems. In an indexed filesystem, you create a symlink that contains the information about which filename we are accessing, the start offset, and the size.

For example:

```
# ln -s @/etc/data:1024:44 spud
```

This creates a symlink with the value @/etc/data:1024:44. If a regular filesystem tried to open this, it would yield an ENOENT immediately, as there is no path that starts with an at-sign.

However, since we control the *c_link()* function call, we can look at the value of the symlink, and determine that it's something special. The @ is our escape character. When you process the *c_link()* function call, instead of creating a symlink as you would normally, you can create anything you like. In this case, we'd create a plain, ordinary-looking file, that internally had the information that we passed to the symlink.

You can now see how this worked in the .tar filesystem — the information that we stored in the extended attributes structure told us the name of the on-disk file (the

`.tar` or `.tar.gz` file) and the offset into that file where the data begins; the regular (non-extended) attributes structure gave us the length.

It's a “simple matter of programming” for you to do the same thing with the indexed filesystem. The only funny thing that happens, though, is that after we create the special symlink, the file looks like a regular file:

```
# ln -s @/etc/data:1024:44 spud
# ls -lF spud
-r--r--r--  1 root    root    44 Aug 16 17:41 spud
```

Normally, you'd expect to see:

```
# ls -lF spud
-r--r--r--  1 root  root  18 Aug 16 17:41 spud@ -> @/etc/data:1024:44
```

but since we converted the symlink to a plain file in the *c_link()* function, you'll never see that.

Executing commands

Don't despair, further abuses are possible!

We can create a symlink that has a `!` or `|` as the first character. We can take this to mean, “Execute the following command and return the standard output of the command as the file content” (in the case of `!`) or “When writing data to the file, pipe it through the following command” (in the case of `|`).

Or, with the creative use of shell escape sequences, you can have the filename in the symlink actually be the name of a command to execute; the standard output of that command is the filename that gets used as the actual value of the symlink:

```
# ln -s \"redirector\" spud
# ls -l spud
-r--r--r--  1 root  root   44 Aug 16 17:41 spud@ -> /dev/server1
# ls -l spud
-r--r--r--  1 root  root   44 Aug 16 17:41 spud@ -> /dev/server2
# ls -l spud
-r--r--r--  1 root  root   44 Aug 16 17:41 spud@ -> /dev/server3
```

In this manner, you could implement some load-sharing functionality all by using what appears to be a “normal” symlink (the double quote character in `"redirector"` is what's used to tell the *c_link()* code that this command's standard output should be used as the value of the symlink). Our little program, `redirector`, simply has a *printf()* that outputs different server names.

And the fun doesn't end there, because when you're processing your *c_link()* handler, you have access to all of the client's information — so you can make decisions based on who the client is, what node they are based on, etc.

You're limited purely by your imagination and what you can get away with at a code review.

Copyright © 2014, QNX Software Systems Limited

Secure filesystem

A more practical filesystem, however, is a secure (or encrypted) filesystem. In this filesystem, you use the underlying facilities of the disk-based filesystem for your backing store (the place where you actually store the data), and you present an encryption/decryption layer on top of the backing store's filesystem.

In essence, this is a modification of the `.tar` filesystem (in that we aren't actually storing the files in memory as in the RAM disk), with the added challenge that we are also allowing writes as well as file/directory creation.

An interesting facet of this filesystem would be to encrypt the filenames themselves. You could use something like the Rijndael/AES algorithm for all of your encryption, and then use base-64 encoding on the resulting (binary) filenames so that they can be stored by a conventional filesystem.

The reason you'd want to encrypt the filenames as well is to prevent “known plain-text” attacks. For example, if you didn't encrypt the filenames, an intruder could look at the files, and seeing a whole bunch of `.wav` files, could immediately deduce that there's a standard header present, giving them a portion of your file in plain text with which to begin their attack.

Line-based filesystem

Another potentially useful spin on filesystems is to make one that's explicitly able to manage text files. Whether you do it with a backing store (on-disk) implementation, or as a RAM disk, depends on your requirements. The characteristic of text files is that they are sequences of line-feed-separated entities, with the data on each line somewhat independent of the data on the previous or following lines. Consider a form-letter, where the only things that might change are the salutation, the account number, and perhaps an amount that's already overdue.

By creating a line-based filesystem, you could change the dynamic components of the form-letter without having to readjust all of the other bytes in the file. If you combine this with something like HTML or PostScript, you can start to see the potential.

The basis of line-based filesystems lies in the same technology used in text editors. An array of pointers-to-characters forms the basis of the “lines.” What the pointer points to is a complete line of text, and that's where you'd make your dynamic content plug in. The rest of the lines wouldn't need to change. By implementing it as an array of IOVs, you can pass this directly to the *MsgReplyv()* function that you'd use to return the data to the client in your *io_read()* handler.

References

Dr. Dobb's Journal, May 1996, “Improving USENET News Performance” describes the original design of the Virtual Filesystem for USENET News.

Header files

The following header files contain useful things for filesystem handlers:

- `<devctl.h>` — contains the definition for *devctl()*, and also defines the component flags used to create a command.
- `<dirent.h>` — contains the directory structure type used by *readdir()*.
- `<sys/dcmd_blk.h>` — contains the `FSYS` *devctl()* block commands.
- `<sys/disk.h>` — defines `partition_entry_t`.
- `<sys/dispatch.h>`, `<sys/iofunc.h>` — Used by resource managers.
- `<sys/fs_stats.h>` — defines the `fs_stats` structure returned by the FSYS block command `DCMD_FSYS_STATISTICS`.
- `<sys/mount.h>` — defines the mount flags.
- `<tar.h>` — defines the layout of the `.tar` header.
- `<zlib.h>` — contains the definitions for the transparent compression library.

Functions

See the following functions:

- *gzclose()*
- *gzopen()*
- *gzread()*
- *gzseek()*

as well as the following in the QNX Neutrino *C Library Reference*:

- *_IO_SET_WRITE_NBYTES()* in the entry for *iofunc_write_verify()*
- *iofunc_attr_init()*
- *iofunc_check_access()*
- *iofunc_client_info_ext()*
- *iofunc_ocb_attach()*
- *iofunc_open()*
- *iofunc_read_verify()*
- *iofunc_write_verify()*
- *MsgReply()*
- *MsgReplyv()*

Copyright © 2014, QNX Software Systems Limited

- *resmgr_attach()*
- *resmgr_msgreadv()*
- *S_ISDIR()*, *S_ISREG()*, and *S_ISLNK()* in the entry for *stat()*
- *SETIOV()*
- *setrlimit()*

Books

An excellent introduction to cryptography is *Applied Cryptography — Protocols, Algorithms, and Source Code in C* by Bruce Schneier (John Wiley & Sons, Inc., 1996, ISBN 0-471-11709-9)

Appendix A
Filesystems

In this appendix, we'll discuss what filesystems are, and how they're implemented within the resource-manager framework. This chapter is written as a tutorial, starting from the basic operations required, and proceeding into detailed implementation. We'll cover the basic concepts used by the RAM disk and the `.tar` filesystem. The RAM disk and `.tar` filesystems themselves are covered in their own chapters, presented earlier in this book.

What is a filesystem?

Before delving deeply into such topics as inodes and attributes structures and symbolic links, let's step back a little and discuss what a filesystem is. At the most basic level, a filesystem is usually a hierarchical arrangement of data. I say "usually" because several (historical) filesystems were flat filesystems — there was no hierarchical ordering of the data. An example of this is the old floppy-based MS-DOS 1.00 and 1.10 filesystems. A nonhierarchical filesystem is a subset of a hierarchical one.

Hierarchical arrangement

The concept of a hierarchical arrangement for filesystems dates back to the dawn of modern computing history. And no, it wasn't invented by Microsoft in MS-DOS 2.00. `:-)`

The hierarchy, or layering, of the filesystem is accomplished by directories. By definition, a directory can contain zero or more subdirectories, and zero or more data elements. (We're not counting the two directories `.` and `..` here.) On UNIX (and derived) systems, pathnames are constructed by having each component separated by a forward slash. (On other systems, like VAX/VMS for example, pathname components were placed in square brackets and separated by periods.) Each slash-separated component represents a directory, with the last component representing a directory or a data element. Only directories can contain further subdirectories and data elements.

Since a directory can contain subdirectories, and the subdirectories can contain further subsubdirectories, this "chain" can theoretically go on forever. Practically, however, there are limits to the length of this chain, because utilities need to be able to allocate buffers big enough to deal with the longest possible pathname. It's for this reason that the names of the individual components (be they directories or data elements) have a maximum length as well.

Data elements

So far, I've been talking about "data elements" rather than the more familiar term "file." That's because a directory can contain not just files, but also symbolic links, named special devices, block devices, and so on. I wanted to distinguish between directories and everything else — that's because only directories can have subdirectories and data elements (although symlinks can "appear" to have sub-directories, but that's a topic for later).

Copyright © 2014, QNX Software Systems Limited

The mount point and the root

A filesystem is often represented as an inverted tree — the root of the tree is shown on the top, and the trunk and branches grow downwards.

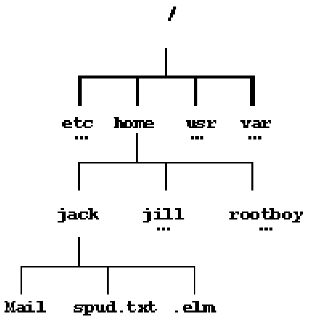


Figure 31: A small portion of a filesystem tree.

Note that I've used an ellipsis (...) to indicate that there are other entries, but they're not shown. Do not confuse this with the two standard directories . and .., which are also not shown!

Here we see the root directory contains etc, home, usr, and var, all of which are directories. The home directory contains users' home directories, and one of these directories is shown, jack, as containing two other directories (Mail and .elm) and a file called spud.txt.

A QNX Neutrino system may have many filesystems within it — there's no reason that there should be only one filesystem. Examples of filesystems are things like CD-ROMs, hard disks, foreign filesystems (like an MS-DOS, EXT2, or QNX 2 partition), network filesystems, and other, more obscure things. The important thing to remember, though, is that these filesystems have their *root* at a certain location within QNX Neutrino's filesystem space. For example, the CD-ROM filesystem might begin at /fs/cd0. This means that the CD-ROM itself is *mounted* at /fs/cd0. Everything under /fs/cd0 is said to belong to the CD-ROM filesystem. Therefore, we say that the CD-ROM filesystem is mounted at /fs/cd0.

What does a filesystem do?

Now that we've looked at the components of a filesystem, we need to understand the basic functionality. A filesystem is a way of organizing data (the hierarchical nature of directories, and the fact that directories and data elements have names), as well as a way of storing data (the fact that the data elements contain useful information).

There are many types of filesystems. Some let you create files and directories within the filesystem, others (called read-only) don't. Some store their data on a hard-disk, others might manufacture the data on-the-fly when you request it. The length of the names of data elements and directories varies with each filesystem. The characters that are valid for the names of data elements and directories vary as well. Some let you create links to directories, some do not. And so on.

Copyright © 2014, QNX Software Systems Limited

Filesystems and QNX Neutrino

Filesystems are implemented as resource managers under QNX Neutrino.

At the highest level, the mount point is represented by the mount structure, `iofunc_mount_t`. Directories and data elements (files, symlinks, etc.) are represented by the attributes structure, `iofunc_attr_t`, which is almost always extended with additional information pertaining to the filesystem.

Note that this representation may look almost exactly, or nothing like, the on-disk data structures (if there even are any). For example, the disk structures corresponding to an MS-DOS filesystem have no concept of QNX Neutrino's `iofunc_attr_t`, even though the filesystem itself has concepts for files and directories. Part of the magic of a filesystem resource manager is that it arranges the attributes structures to mimic the perceived organization of the on-media (if any) data structures.

Effectively, the resource manager builds an internal representation of the underlying filesystem, by using the QNX Neutrino data structures. This is key to understanding QNX Neutrino filesystem implementations.

In our previous example, the filesystem might be represented internally as follows:

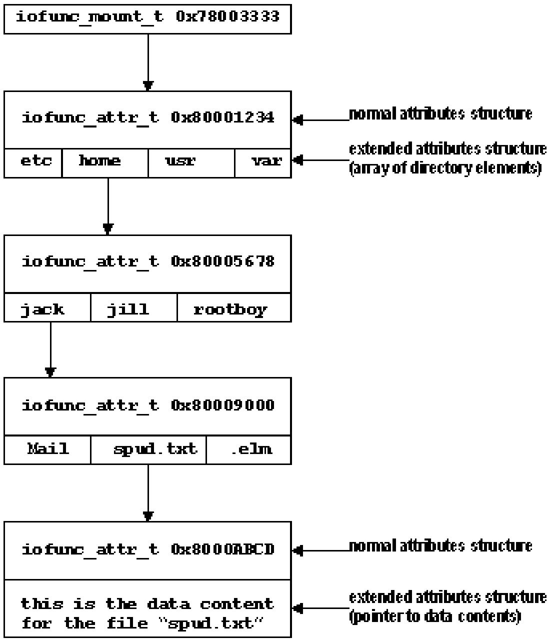

iofunc_mount_t 0x78003333
iofunc_attr_t 0x80001234
normal attributes structure
etc
home
usr
var
extended attributes structure
(array of directory elements)
iofunc_attr_t 0x80005678
jack
jill
rootboy
iofunc_attr_t 0x80009000
Mail
spud.txt
.elm
iofunc_attr_t 0x8000ABCD
normal attributes structure
this is the data content
for the file "spud.txt"
extended attributes structure
(pointer to data contents)

Figure 32: Internal resource manager view of a filesystem.

The example above shows the data structures leading up to the file `/home/jack/spud.txt`. The numbers in hexadecimal (e.g. `0x7800333`) are sample addresses of the data structures, to emphasize the point that these things live in memory (and not on a disk somewhere).

Copyright © 2014, QNX Software Systems Limited

As you can see, we've used extended versions of the regular resource manager structures. This is because the regular structures have no concepts of things like directories, so we need to add that kind of information. Our attributes structure extensions are of two types, one for directories (to contain the names of the components within that directory) and another for data elements (to contain, in this example, the data itself).

Note that the attributes structure for the data element is shown containing the data directly (this would be the case, for example, with the RAM disk — the data is stored in RAM, so the extended attributes structure would simply contain a pointer to the block or blocks that contain the data). If, on the other hand, the data was stored on media, then instead of storing a pointer to the data, we might store the starting disk block number instead. This is an implementation detail we'll discuss later.

Copyright © 2014, QNX Software Systems Limited

How does a filesystem work?

There are several distinct operations carried out by a filesystem resource manager:

- mount point management
- pathname resolution
- directory management
- data element content management

Most of these operations are either very trivial or nonexistent in a non-filesystem manager. For example, a serial port driver doesn't worry about mount point, pathname, or directory management. Its data element content management consists of managing the circular buffer that's common between the resource manager and the interrupt service routine that's interfacing to the serial port hardware. In addition, the serial port driver may worry about things like *devctl()* functionality, something that's rarely used in filesystem managers.

Regardless of whether the filesystem that we're discussing is based on a traditional disk-based medium, or if it's a *virtual* filesystem, the operations are the same. (A virtual filesystem is one in which the files or directories aren't necessarily tied directly to the underlying media, perhaps being manufactured on-demand.)

Let's look at the operations of the filesystem in turn, and then we'll take a look at the detailed implementation.

Mount point management

A filesystem needs to have a *mount point* — somewhere that we consider to be the root of the filesystem. For a traditional disk-based filesystem, this often ends up being in several places, with the root directory (i.e. `/`) being one of the mount points. On your QNX Neutrino system, other filesystems are mounted by default in `/fs`. The `.tar` filesystem, just to give another example, may be mounted in the same directory as the `.tar` file it's managing.

The exact location of the mount point isn't important for our discussion here. What is important is that there exists some mount point, and that the top level directory (the mount point itself) is called the *root of the filesystem*. In our earlier example, we had a CD-ROM filesystem mounted at `/fs/cd0`, so we said that the directory `/fs/cd0` is both the “mount point” for the CD-ROM filesystem, and is also the “root” of the CD-ROM filesystem.

One of the first things that the filesystem must do, therefore, is to register itself at a given mount point. There are actually three registrations that need to be considered:

- unnamed mount registration
- special device registration

Copyright © 2014, QNX Software Systems Limited

- mount point registration

All three registrations are performed with the resource manager function *resmgr_attach()*, with variations given in the arguments. For reference, here's the prototype of the *resmgr_attach()* function (from `<sys/dispatch.h>`):

```
int resmgr_attach (
    dispatch_t *dpp,
    resmgr_attr_t *attr,
    const char *path,
    enum _file_type file_type,
    unsigned flags,
    const resmgr_connect_funcs_t *connect_funcs,
    const resmgr_io_funcs_t *io_funcs,
    RESMGR_HANDLE_T *handle);
```

Of interest for this discussion are *path*, *file_type*, and *flags*.

Unnamed mount registration

The purpose of the unnamed mount registration is to serve as a place to "listen" for mount messages. These messages are generated from the command-line `mount` command (or the C function *mount()*, which is what the command-line version uses).

To register for the unnamed mount point, you supply a `NULL` for the *path* (because it's unnamed), a `_FTYPE_MOUNT` for the *file_type*, and the flags `_RESMGR_FLAG_DIR | _RESMGR_FLAG_FTYPEONLY` for the *flags* field.

Special device registration

The special device is used for identifying your filesystem driver so that it can serve as a target for mounts and other operations. For example, if you have two CD-ROMs, they'll show up as `/dev/cd0` and `/dev/cd1`. When you mount one of those drives, you'll need to specify which drive to mount from. The special device name is used to make that determination.

Together with the unnamed mount registration, the special device registration lets you get the mount message and correctly determine which device should be mounted, and where.

Once you've made that determination, you'd call *resmgr_attach()* to perform the actual mount point registration (see the next section below).

The special device is registered with a valid name given to *path* (for example, `/dev/ramdisk`), a flag of `_FTYPE_ANY` given to *file_type*, and a `0` passed to *flags*.

Mount point registration

The final step in mounting your filesystem is the actual mount point registration where the filesystem will manifest itself. The preceding two steps (the unnamed and the special device registration) are optional — you can write a perfectly functioning filesystem that doesn't perform those two steps, but performs only this mount point registration (e.g. the RAM-disk filesystem).

Copyright © 2014, QNX Software Systems Limited

Apart from the initial setup of the filesystem, the unnamed and special device registration mount points are almost never used.

Once you've called *resmgr_attach()* to create your filesystem's mount point, you can handle requests for the filesystem. The *resmgr_attach()* function is called with a valid path given to *path* (for example, `/ramdisk`), a flag of `_FTYPE_ANY` given to *file_type*, and a flag of `_RESMGR_FLAG_DIR` given to *flags* (because you will support directories within your mount point).

Pathname resolution

The next function that your filesystem must perform is called *pathname resolution*. This is the process of accepting an arbitrary pathname, and analyzing the components within the pathname to determine if the target is accessible, possibly creating the target, and eventually resolving the target into an attributes structure.

While this may sound fairly trivial, it's actually a fair bit of work to get it just right, especially if you want to handle symbolic links correctly along the way. `:-)`

The algorithm can be summarized as follows. Break up the pathname into components (each component is delimited by the forward slash / character). For each pathname component, perform the following:

1. If there are additional components after this one, then this component must be a directory. Return an error if this component is not a directory.
2. Check to see that we can access the current component (permission checks). If we can't, return an error indicating the access problem.
3. Search the directory of the parent to see if this component exists. If not, return an error (for new component creation, this error is handled higher up, and a new component is created.)
4. If this component is a symbolic link (symlink), and it's either a non-terminal symlink, or certain flags are set, redirect the symlink (processing ends).

If all of the above steps proceeded without problems, we've effectively performed “pathname resolution.” We now have either a pointer to an attributes structure that identifies the target (and, for convenience, the target's parent attributes structure as well), or we have a pointer to just the parent attributes structure. The reason we might not have a pointer to the target is when we're creating the target. We need to verify that we can actually *get to* where the target will be, but obviously we don't expect the target to exist.

Returning to our example of `/home/jack/spud.txt`, let's imagine a user trying to open that file for reading. Here's the diagram we used earlier:

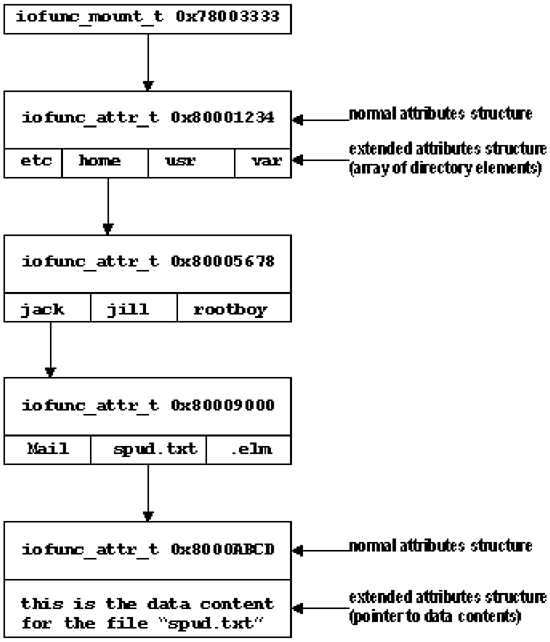

iofunc_mount_t 0x78003333
iofunc_attr_t 0x80001234
etc
home
usr
var
normal attributes structure
extended attributes structure (array of directory elements)
iofunc_attr_t 0x80005678
jack
jill
rootboy
iofunc_attr_t 0x80009000
Mail
spud.txt
.elm
iofunc_attr_t 0x8000ABCD
this is the data content for the file "spud.txt"
normal attributes structure
extended attributes structure (pointer to data contents)

Figure 33: Internal resource manager view of filesystem.

We see that the root of this filesystem is the attributes structure at address `0x80001234`. We're assuming (for simplicity) that this filesystem is mounted at the root of the filesystem space. The first component we see in `/home/jack/spud.txt` is `home`. Because we have the address of the attributes structure (`0x80001234`), we can look at the array of directory elements and see if we can find the component `home`. According to our first rule, since there are additional components after this one (in fact, they are `jack/spud.txt`), we know that `home` must be a directory. So we check the attributes structure's *mode* member, and discover that `home` is indeed a directory.

Next, we need to look at the attributes structure's *uid*, *gid*, and *mode* fields to verify that the user doing the *open()* call (to open the file for reading) is actually allowed to perform that operation. Note that we look at all the pathname components along the way, not just the final one! We assume this passes. We repeat the process for the attributes structure at `0x80005678`, and this time search for the next component, `jack`. We find it, and we have the appropriate permissions. Next we perform this process again for the attributes structure at `0x80009000`, and find `spud.txt` (and once again assume permissions are OK). This leads us to the attributes structure at `0x8000ABCD`, which contains the data that we're after. Since we're only doing an *open()*, we don't actually return any data. We bind our OCB to the attributes structure at `0x8000ABCD` and declare the *open()* a terrific success.

Directory management

The example walkthrough above hinted at a few of the directory management features; we need to have the contents of the directories stored somewhere where we can search them, and we need to have the individual directory entries point to the attributes structures that contain information for that entry.

So far, we've made it look as if all possible files and directories need to be stored in RAM at all times. This is not the case! Almost all disks these days are far, far, bigger than available memory, so storing all possible directory and file entries is just not possible. Typically, attributes structures are cached, with only a fixed number of them being present in memory at any one time. When an attributes structure is no longer needed, its space is reused (or it's *free()*'d). The notable exception to this is, of course, a RAM disk, where all the data (including the contents!) must be stored in RAM.

The extended attributes structure shown in the diagram earlier implies that all of the directory entries are stored within the attributes structure itself, in some kind of an array. The actual storage method (array, linked list, hash table, balanced tree, whatever) is entirely up to your imagination; I just showed a simple one for discussion. The two examples in this book, the RAM disk and the `.tar` filesystem, use the array method; again, purely for simplicity.

You must be able to search in a directory, and you also need to be able to add, remove, and move (rename) entries. An entry is added when a file or directory is created; an entry is removed when the file or directory is deleted; and an entry is moved (renamed) when the name of the entry is changed. Depending on the complexity of your filesystem, you may allow or disallow moving directory entries from one directory entry to another. A simple rename, like `mv spud.txt abc.txt`, is supported by almost all filesystems. A more complex rename, like `mv spud.txt tmp/` may or may not be supported. The RAM-disk filesystem supports complex renames.

Finally, the last aspect of directory management that your filesystem must support is the ability to return information about the contents of directories. This is accomplished in your *io_read()* handler (when the client calls *readdir()*), and is discussed thoroughly in the *RAM-disk Filesystem* (p. 135) chapter.

Data element content management

Finally, your filesystem might just contain some actual files! There's no requirement to do this — you can have perfectly legal and useful filesystems that have only symlinks (and no files), for example.

Data element content management consists of binding data to the attributes structure, and providing some access to the data. There's a range of complexity involved here. In the RAM disk, the simplest thing to do is allocate or deallocate space as the size of the file changes (using a block allocator for efficiency), and store the new contents in the allocated space. It's more complicated if you're dealing with an actual disk-based filesystem, since you'll need to allocate blocks on disk, make records of which blocks you allocated, perhaps link them together, and so on. You can see the details of the *io_read()* handlers in both the RAM disk and the `.tar` filesystem, and the *io_write()* handler in the RAM disk filesystem (the `.tar` filesystem is read-only and therefore doesn't support writing).

Copyright © 2014, QNX Software Systems Limited

References

The following references apply to this chapter:

- See the *RAM-disk Filesystem* (p. 135), *TAR Filesystem* (p. 177), and *Web Counter Resource Manager* (p. 67) chapters in this book for examples of filesystem resource managers.
- *Get Programming with the QNX Neutrino RTOS* by Robert Krten contains the basics of resource managers, message passing, etc.

Appendix B
The /proc Filesystem

If you need to gather information about the processes running on your machine, you can use the /proc filesystem. Although there's a section about it in the Processes chapter of the QNX Neutrino *Programmer's Guide*, this filesystem isn't understood very well. This appendix describes the main features of the /proc filesystem so you can use it in your own utilities.

First of all, the /proc filesystem is a *virtual* filesystem — it doesn't actually exist on disk; it exists only as a figment of the process manager's imagination.

The /proc filesystem contains a number of entities:

- directories, one per process, for every process running in the system
- the boot subdirectory — a "mini" filesystem that contains the files from the startup image
- the (hidden) mount subdirectory.

Our main focus in this chapter is the first item, the directories describing each process in the system. We'll describe the functions available to get information about the processes (and their threads).

For completeness, however, we'll just briefly mention the other two items.

Copyright © 2014, QNX Software Systems Limited

The /proc/boot directory

By default, the files from the startup image are placed into a read-only filesystem mounted at /, in a directory called /proc/boot. In a tiny embedded system, for example, this might be used to hold configuration files, or other data files, without the need for a full-blown filesystem manager, and also without the need for additional storage. You can get more information about this by looking at the mkifs command in the *Utilities Reference*.

Copyright © 2014, QNX Software Systems Limited

The /proc/mount directory

This one is actually pretty neat. When I say that this filesystem is hidden, I mean that when you do an `ls` of `/proc`, the `mount` directory doesn't show up. But you can certainly `cd` into it and look at its contents.

There are two main types of entities:

- directories with names that are comma-separated numbers,
- directories with names corresponding to mounted (i.e. *resmgr_pathname_attach()*'d) entities.

This is what `/proc/mount` looks like on my system:

```
# ls /proc/mount
ls: No such file or directory (/proc/mount/0,8,1,0,0)
ls: No such file or directory (/proc/mount/0,1,1,2,-1)
0,1,1,10,11/        0,344083,1,0,11/    0,6,7,10,0/         proc/
0,1,1,11,0/         0,360468,1,0,11/    0,8,1,1,0/          usr/
0,1,1,3,-1/         0,393228,4,0,11/    dev/
0,12292,1,0,6/      0,4105,1,0,4/       fs/
0,12292,1,1,8/      0,6,7,0,11/         pkgs/
```

Each “numbered” directory name (e.g. `0,344083,1,0,11`) consists of five numbers, separated by commas. The numbers are, in order:

- node ID,
- process ID,
- channel ID,
- handle, and
- file type.

The node ID is usually zero, indicating “this node.” The process ID is that of the process. The channel ID is the number of the channel created via *ChannelCreate()*. Finally, the handle is an index describing which *resmgr_pathname_attach()* this is. The last number is the file type (see `<sys/ftype.h>` for values and meanings).

Together, these five numbers describe a pathname prefix that has been registered in the pathname space.

The other, “normal” directories are the actual registered paths. If we examine a random one, say `/proc/mount/dev`, we'll see *directories* corresponding to each of the registered mount points under `/dev`. You may be wondering why they are directories, and not the actual devices. That's because you can register the same pathname multiple times. Recall that in the High Availability chapter we said that in order to achieve hot-standby mode, we'd want to register two resource managers at the same mount point — the one “in front” would be the active resource manager, the one registered “behind” would be the standby resource manager. If we did this, we'd have two sets of numbers in the subdirectory corresponding to the named device.

For example, currently we have one resource manager managing the serial ports:

```
# ls /proc/mount/dev/ser1
0,344080,1,0,0
```

If we had a hot-standby serial port driver (we don't, but play along) the directory listing might now look something like:

```
# ls /proc/mount/dev/ser1
0,344080,1,0,0    0,674453,1,0,0
```

The process ID 344080 is the active serial port driver, and the process ID 674453 is the standby serial port driver. The order of the pathname resolution is given by the order that the entries are returned by the *readdir()* function call. This means that it's not immediately apparent via the `ls` above which process is resolved first (because by default, `ls` sorts alphabetically by name), but by calling `ls` with the -S (“do not sort” option) the order can be determined.

The /proc by-process-ID directories

In the main portion of this appendix, we'll look at the format, contents, and functions you can use with the remainder of the /proc filesystem.

You may wish to have the <sys/procfs.h>, <sys/syspage.h>, and <sys/debug.h> header files handy — I'll show important pieces as we discuss them.

A casual ls of /proc yields something like this, right now on my system:

```
# ls -F /proc
1/          12292/      3670087/    6672454/    950306/
1011730/    2/          3670088/    6676553/    950309/
1011756/    3/          393228/     7/          950310/
1011757/    344077/     405521/     7434315/    950311/
1011760/    344079/     4105/       7462988/    950312/
1011761/    344080/     442378/     7467085/    950313/
1011762/    344083/     45067/      7499854/    950314/
1011764/    3551288/    466965/     770071/     950315/
1011769/    3551294/    471062/     8/          950318/
1011770/    3571775/    479246/     815133/     950319/
1011773/    360468/     4886602/    831519/     boot/
1015863/    3608627/    5/          831520/     dumper#
1036347/    3608629/    548888/     868382/     self/
1040448/    3629116/    593947/     868387/
1044547/    3629121/    6/          868388/
1044548/    3649602/    622620/     950298/
1093686/    3649605/    626713/     950305/
```

We've discussed the boot entry above. The dumper entry is a hook for dumper (the system core dump utility). Finally, self is a short-form for the current process (in this case, ls).

The individual numbered directories are more interesting. Each number is a process ID. For example, what is process ID 4105? Doing the following:

```
# pidin -p4105
     pid tid name               prio STATE       Blocked
    4105   1 sbin/pipe           10o RECEIVE     1
    4105   2 sbin/pipe           10o RECEIVE     1
    4105   4 sbin/pipe           10o RECEIVE     1
    4105   5 sbin/pipe           10o RECEIVE     1
```

shows us that process ID 4105 is the pipe process. It currently has four threads (thread IDs 1, 2, 4 and 5 — thread ID 3 ran at some point and died, that's why it's not shown).

Within the /proc directory, doing a:

```
# ls -l 4105
total 2416
-rw-r--r--  1 root      root        1236992 Aug 21 21:25 as
```

shows us a file called as (not in the sense of "as if..." but as an abbreviation for *a*ddress *s*pace). This file contains the addressable address space of the entire process. The size of the file gives us the size of the addressable address space, so we can see that pipe is using a little under one and a quarter megabytes of address space.

Copyright © 2014, QNX Software Systems Limited

To further confuse our findings, here's:

```
# pidin -p4105 mem
 pid tid name           prio STATE       code  data        stack
4105   1 sbin/pipe       10o RECEIVE       16K  148K  4096(132K)
4105   2 sbin/pipe       10o RECEIVE       16K  148K  4096(132K)
4105   4 sbin/pipe       10o RECEIVE       16K  148K  4096(132K)
4105   5 sbin/pipe       10o RECEIVE       16K  148K  4096(132K)
        ldqnx.so.2     @b0300000         312K   16K
```

If you do the math (assuming the stack is 4096 bytes, as indicated) you come up with:

16k + 148k + 4 x 132k + 312k + 16k

Or 1020k (1,044,480 bytes), which is short by 192,512 bytes.

You'll notice that the sizes don't match up! That's because the as file totals up all the segments that have the MAP_SYSRAM flag on in them and uses that as the size that it reports for a *stat()*. MAP_SYSRAM can't be turned on by the user in *mmap()*, but it indicates that the system allocated this piece of memory (as opposed to the user specifying a direct physical address with MAP_PHYS), so when the memory no longer has any references to it, the system should free the storage (this total includes memory used by any shared objects that the process has loaded). The code in pidin that calculates the pidin mem sizes is, to put it nicely, a little convoluted.

Operations on the as entry

Given that the as entry is the virtual address space of the process, what can we do with it? The as entity was made to look like a file so that you could perform file-like functions on it (*read()*, *write()*, and *lseek()*).

For example, if we call *lseek()* to seek to location 0x80001234, and then call *read()* to read 4 bytes, we have effectively read 4 bytes from the process's virtual address space at 0x80001234. If we then print out this value, it would be equivalent to doing the following code within that process:

```
...

int     *ptr;

ptr = (int *) 0x80001234;
printf ("4 bytes at location 0x80001234 are %d\n", *ptr);
```

However, the big advantage is that we can read the data in another process's address space by calling *lseek()* and *read()*.

Discontiguous address space

The address space within the entry is discontiguous, meaning that there are “gaps” in the “file offsets.” This means that you will not be able to *lseek()* and then *read()* or *write()* to arbitrary locations — only the locations that are valid within the address space of that process. This only makes sense, especially when you consider that the process itself has access to only the “valid” portions of its own address space — you can't construct a pointer to an arbitrary location and expect it to work.

Someone else's virtual address space

Which brings us to our next point. The address space that you are dealing with is that of the process under examination, *not your own*. It is impossible to map that process's address space on a one-to-one basis into your own (because of the potential for virtual address conflicts), so you must use *lseek()*, *read()* and *write()* to access this memory.

The statement about not being able to *mmap()* the process's address space to your own is true right now (as of version 6.2.1), but eventually you will be able to use that file as the file descriptor in *mmap()* (it'll be allowed as a memory-mapped file). My sources at QNX Software Systems indicate that this is not in the “short term” plan, but something that might appear in a future version.

Why is it impossible to map it on a one-to-one basis? Because the whole point of virtual addressing is that multiple processes could have their own, independent address spaces. It would defeat the purpose of virtual addressing if, once a process was assigned a certain address range, that address range then became unavailable for all other processes.

Since the reason for mapping the address space of the other process to your own would be to use the other process's pointers “natively,” and since that's not possible due to address conflicts, we'll just stick with the file operations.

Now, in order to be able to read “relevant portions” of the process's address space, we're going to need to know where these address ranges actually are. There are a number of *devctl()*'s that are used in this case (we'll see these shortly).

Finding a particular process

Generally, the first thing you need to do is select a particular process (or some set of processes) to perform further work on. Since the `/proc` filesystem contains process IDs, if you already know the process ID, then your work is done, and you can continue on to the next step (see “Iterating through the list of processes” below). However, if all you know is the name of the process, then you need to search through the list of process IDs, retrieve the names of each process, and match it against the process name you're searching for.

There may be criteria other than the name that you use to select your particular process. For example, you may be interested in processes that have more than six threads, or processes that have threads in a particular state, or whatever. Regardless, you will still need to iterate through the process ID list and select your process(es).

Copyright © 2014, QNX Software Systems Limited

Iterating through the list of processes

Since the /proc filesystem looks like a normal filesystem, it's appropriate to use the filesystem functions *opendir()* and *readdir()* to iterate through the process IDs.

The following code sample illustrates how to do this:

```
void
iterate_processes (void)
{
  struct dirent   *dirent;
  DIR             *dir;
  int             r;
  int             pid;

  // 1) find all processes
  if (!(dir = opendir ("/proc"))) {
    fprintf (stderr, "%s:  couldn't open /proc, errno %d\n",
             progname, errno);
    perror (NULL);
    exit (EXIT_FAILURE);
  }

  while (dirent = readdir (dir)) {
    // 2) we are only interested in process IDs
    if (isdigit (*dirent -> d_name)) {
      pid = atoi (dirent -> d_name);
      iterate_process (pid);
    }
  }
  closedir (dir);
}
```

At this point, we've found all valid process IDs. We use the standard *opendir()* function in step 1 to open the /proc filesystem. In step 2, we read through all entries in the /proc filesystem, using the standard *readdir()*. We skip entries that are nonnumeric — as discussed above, there are other things in the /proc filesystem besides process IDs.

Next, we need to search through the processes generated by the directory functions to see which ones match our criteria. For now, we'll just match based on the process name — by the end of this appendix, it will be apparent how to search based on other criteria (short story: ignore the name, and search for your other criteria in a later step).

```
void
iterate_process (int pid)
{
  char      paths [PATH_MAX];
  int       fd;

  // 1) set up structure
  static struct {
    procfs_debuginfo    info;
    char                buff [PATH_MAX];
  } name;

  sprintf (paths, "/proc/%d/as", pid);

  if ((fd = open (paths, O_RDONLY)) == -1) {
    return;
  }

  // 2) ask for the name
  if (devctl (fd, DCMD_PROC_MAPDEBUG_BASE, &name,
              sizeof (name), 0) != EOK) {
    if (pid == 1) {
      strcpy (name.info.path, "(procnto)");
```

Copyright © 2014, QNX Software Systems Limited

```
    } else {
      strcpy (name.info.path, "(n/a)");
    }
  }

  // 3) we can compare against name.info.path here...
  do_process (pid, fd, name.info.path);
  close (fd);
}
```

In step 1, we set up an extension to the `procfs_debuginfo` data structure. The *buff* buffer is implicitly past the end of the structure, so it's natural to set it up this way. In step 2, we ask for the name, using DCMD_PROC_MAPDEBUG_BASE.

Note that some versions of QNX Neutrino didn't provide a “name” for the process manager. This is easy to work around, because the process manager is *always* process ID 1.

Just before step 3 is a good place to compare the name against whatever it was you're looking for. By not performing any comparison, we match all names.

If the name matches (or for all processes, as shown in the code above), we can call *do_process()*, which will now work on the process. Notice that we pass *do_process()* the opened file descriptor, *fd*, to save on having to reopen the `as` entry again in *do_process()*.

Finding out information about the process

Once we've identified which process we're interested in, one of the first things we need to do is find information about the process. (We'll look at how to get information about the threads in a process shortly.)

There are six *devctl()* commands that deal with processes:

DCMD_PROC_MAPDEBUG_BASE

Returns the name of the process (we've used this one above, in *iterate_process()*).

DCMD_PROC_INFO

Returns basic information about the process (process IDs, signals, virtual addresses, CPU usage).

DCMD_PROC_MAPINFO and DCMD_PROC_PAGEDATA

Returns information about various chunks (“segments,” but not to be confused with x86 segments) of memory.

DCMD_PROC_TIMERS

Returns information about the timers owned by the process.

DCMD_PROC_IRQS

Returns information about the interrupt handlers owned by the process.

Other *devctl()* commands deal with processes as well, but they're used for control operations rather than fetching information.

DCMD_PROC_INFO

The following information is readily available about the process via the DCMD_PROC_INFO *devctl()* command:

```
typedef struct _debug_process_info {
        pid_t      pid;
        pid_t      parent;
        _Uint32t   flags;
        _Uint32t   umask;
        pid_t      child;
        pid_t      sibling;
        pid_t      pgrp;
        pid_t      sid;
        _Uint64t   base_address;
        _Uint64t   initial_stack;
        uid_t      uid;
        gid_t      gid;
        uid_t      euid;
        gid_t      egid;
        uid_t      suid;
        gid_t      sgid;
        sigset_t   sig_ignore;
        sigset_t   sig_queue;
        sigset_t   sig_pending;
        _Uint32t   num_chancons;
        _Uint32t   num_fdcons;
        _Uint32t   num_threads;
        _Uint32t   num_timers;
        _Uint64t   start_time;     /* Start time in nsec */
        _Uint64t   utime;          /* User running time in nsec */
        _Uint64t   stime;          /* System running time in nsec */
        _Uint64t   cutime;         /* terminated children user time in nsec */
        _Uint64t   cstime;         /* terminated children user time in nsec */
        _Uint8t    priority;       /* process base priority */
        _Uint8t    reserved2[7];
        _Uint8t    extsched[8];
        _Uint64t   pls;            /* Address of process local storage */
        _Uint64t   sigstub;        /* Address of process signal trampoline */
        _Uint64t   canstub;        /* Address of process thread cancellation trampoline *
        _Uint64t   private_mem;    /* Amount of MAP_PRIVATE memory */
        _Uint32t   appid;          /* Application id */
        _Uint32t   reserved3[1];
        _Uint64t   reserved[8];
} debug_process_t;
```

This information is filled into the debug_process_t structure by issuing the DCMD_PROC_INFO *devctl()*. Note that the debug_process_t is the same type as procfs_info (via a typedef in <sys/procfs.h>). To get this structure:

```
void
dump_procfs_info (int fd, int pid)
{
  procfs_info info;
  int         sts;

  sts = devctl (fd, DCMD_PROC_INFO, &info, sizeof (info), NULL);
  if (sts != EOK) {
    fprintf(stderr, "%s: DCMD_PROC_INFO pid %d errno %d (%s)\n",
            progname, pid, errno, strerror (errno));
    exit (EXIT_FAILURE);
  }

  // structure is now full, and can be printed, analyzed, etc.
  ...
}
```

As an example, we'll stick with the pipe process. Here are the contents of the procfs_info structure for the pipe process:

```
PROCESS ID 4105
Info from DCMD_PROC_INFO
  pid            4105
  parent         2
  flags          0x00000210
  umask          0x00000000
  child          0
  sibling        8
  pgrp           4105
  sid            1
  base_address   0x0000000008048000
  initial_stack  0x0000000008047F18
  uid            0
  gid            0
  euid           0
  egid           0
  suid           0
  sgid           0
  sig_ignore     0x06800000-00000000
  sig_queue      0x00000000-FF000000
  sig_pending    0x00000000-00000000
  num_chancons   4
  num_fdcons     3
  num_threads    4
  num_timers     0
  start_time     0x0EB99001F9CD1EF7
  utime          0x0000000016D3DA23
  stime          0x000000000CDF64E8
  cutime         0x0000000000000000
  cstime         0x0000000000000000
  priority       10
```

Let's look at the various fields that are present here.

Process information

The *pid*, *parent*, *child*, and *sibling* fields tell us the relationship of this process to other processes. Obviously, *pid* is the process ID of the process itself, and *parent* is the process ID of the process that created this one. Where things get interesting is the *child* and *sibling* entries. Let's take an example of a process P that created processes A, B, and C. Process P is the parent of A, B and C, thus we'd expect that the *parent* field would contains the process ID for process P (in each of the three children processes). However, you'll notice that the *child* member is a scalar, and not an array as you may have been expecting. This means that P's children are listed as a child/sibling relationship, rather than an array of children. So, it may be the case that P's *child* member is the process ID for process A, and the other children, B and C are listed as siblings (in the *sibling* member) of each other. So, instead of:

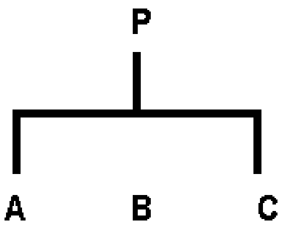


Figure 34: A parent/child relationship.

we'd see a relationship more like:

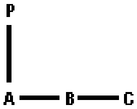


Figure 35: A parent/child/sibling relationship.

It's the same, hierarchically speaking, except that we've avoided having to keep an array of children. Instead, we have each of the children point to a sibling, thus forming a list.

Additional process information provided is the process group (*pgrp*), session ID (*sid*), and the usual extended user and group information (*uid*, *gid*, *euid*, *egid*, *suid*, and *sgid*).

The process's base priority is provided in the *priority* member. Note that, practically speaking, a process doesn't really have a priority — since threads are the actual schedulable entities, they will be the ones that “actually” have a priority. The priority given here is the default priority that's assigned to the process's first thread when the process started. New threads that are started can inherit the priority of the creating thread, have a different priority set via the POSIX thread attributes structure, or can change their priority later.

Finally, the number of threads (*num_threads*) is provided.

Memory information

Basic memory information is provided by the *base_address* and *initial_stack* members. Remember, these are the virtual addresses used by the process, and have no meaning for any other process, nor are they (easily) translatable to physical addresses.

Signal information

Three fields relating to signals are provided: *sig_ignore*, *sig_queue*, and *sig_pending*, representing, respectively, the signals that this process is ignoring, the signals that are enqueued on this process, and the signals that are pending. A signal is one of these “weird” things that has both a “process” and a “thread” facet — the fields mentioned here are for the “process” aspect.

Note also that the signals are stored in a `sigset_t`. QNX Neutrino implements the `sigset_t` as an array of two `long` integers; that's why I've shown them as a 16-digit hexadecimal number with a dash between the two 32-bit halves.

CPU usage information

Another nice thing that's stored in the structure is a set of CPU usage (and time-related) members:

Copyright © 2014, QNX Software Systems Limited

start_time

The time, in nanoseconds since January 1, 1970, when the process was started.

utime

The number of nanoseconds spent running in user space (see below).

stime

The number of nanoseconds spent running in system space (see below).

cutime* and *cstime

Accumulated time that terminated children have run, in nanoseconds, in user and system space.

The *start_time* is useful not only for its obvious “when was this process started” information, but also to detect reused process IDs. For example, if a process ID X is running and then dies, eventually the same process ID (X) will be handed out to a new (and hence completely different) process. By comparing the two process's *start_time* members, it's possible to determine that the process ID has in fact been reused.

The *utime* and *stime* values are calculated very simply — if the processor is executing in user space when the timer tick interrupt occurs, time is allocated to the *utime* variable; otherwise, it's allocated to the *stime* variable. The granularity of the time interval is equivalent to the time tick (e.g. 1 millisecond on an x86 platform with the default clock setting).

Miscellaneous

There are a few other miscellaneous members:

flags

Process flags; see `<sys/neutrino.h>`, specifically the constants beginning with `_NTO_PF`.

umask

The `umask` file mode mask used for creating files.

num_chancons

The number of connected channels.

num_fdcons

The number of connected file descriptors.

num_timers

The number of timers in use.

DCMD_PROC_MAPINFO and DCMD_PROC_PAGEDATA

The next thing that we can do with a process is look at the memory segments that it has in use. There are two *devctl()* commands to accomplish this: DCMD_PROC_MAPINFO and DCMD_PROC_PAGEDATA.

Both commands use the same data structure (edited for clarity):

```
typedef struct _procfs_map_info {
  _Uint64t   vaddr;
  _Uint64t   size;
  _Uint32t   flags;
  dev_t      dev;
  off_t      offset;
  ino_t      ino;
} procfs_mapinfo;
```

The original data structure declaration has #ifdef's for 32 versus 64 bit sizes of the *offset* and *ino* members.

The procfs_mapinfo is used in its array form, meaning that we must allocate sufficient space for all of the memory segments that we will be getting information about. Practically speaking, I've managed just fine with 512 (MAX_SEGMENTS) elements. When I use this call in code, I compare the number of elements available (returned by the *devctl()* function) and ensure that it is less than the constant MAX_SEGMENTS. In the unlikely event that 512 elements are insufficient, you can allocate the array dynamically and reissue the *devctl()* call with a bigger buffer. In practice, the 10 to 100 element range is sufficient; 512 is overkill.

Here's how the call is used:

```
#define MAX_SEGMENTS  512

void
dump_procfs_map_info (int fd, int pid)
{
  procfs_mapinfo  membufs [MAX_SEGMENTS];
  int             nmembuf;
  int             i;

  // fetch information about the memory regions for this pid
  if (devctl (fd, DCMD_PROC_PAGEDATA, membufs, sizeof (membufs),
              &nmembuf) != EOK) {
    fprintf(stderr, "%s:  PAGEDATA process %d, errno %d (%s)\n",
            progname, pid, errno, strerror (errno));
    exit (EXIT_FAILURE);
  }

  // check to see we haven't overflowed
  if (nmembuf > MAX_SEGMENTS) {
    fprintf (stderr, "%s: proc %d has > %d memsegs (%d)!!!\n",
             progname, pid, MAX_SEGMENTS, nmembuf);
    exit (EXIT_FAILURE);
  }

  for (i = 0; i < nmembuf; i++) {
    // now we can print/analyze the data
```

Copyright © 2014, QNX Software Systems Limited

```
  }
}
```

Here's the output for the `pipe` process (I've added blank lines for clarity):

```
Info from DCMD_PROC_PAGEDATA
  Buff# --vaddr--- ---size--- ---flags-- ---dev---- ---ino----

  [  0] 0x07F22000 0x00001000 0x01001083 0x00000002 0x00000001
  [  1] 0x07F23000 0x0001F000 0x01001783 0x00000002 0x00000001
  [  2] 0x07F42000 0x00001000 0x01401783 0x00000002 0x00000001

  [  3] 0x07F43000 0x00001000 0x01001083 0x00000002 0x00000001
  [  4] 0x07F44000 0x0001F000 0x01001783 0x00000002 0x00000001
  [  5] 0x07F63000 0x00001000 0x01401783 0x00000002 0x00000001

  [  6] 0x07F64000 0x00001000 0x01001083 0x00000002 0x00000001
  [  7] 0x07F65000 0x0001F000 0x01001783 0x00000002 0x00000001
  [  8] 0x07F84000 0x00001000 0x01401783 0x00000002 0x00000001

  [  9] 0x07FA6000 0x00001000 0x01001083 0x00000002 0x00000001
  [ 10] 0x07FA7000 0x0001F000 0x01001783 0x00000002 0x00000001
  [ 11] 0x07FC6000 0x00001000 0x01401783 0x00000002 0x00000001

  [ 12] 0x07FC7000 0x00001000 0x01001083 0x00000002 0x00000001
  [ 13] 0x07FC8000 0x0007E000 0x01001383 0x00000002 0x00000001
  [ 14] 0x08046000 0x00002000 0x01401383 0x00000002 0x00000001

  [ 15] 0x08048000 0x00004000 0x00400571 0x00000001 0x00000009

  [ 16] 0x0804C000 0x00001000 0x01400372 0x00000001 0x00000009
  [ 17] 0x0804D000 0x00024000 0x01400303 0x00000002 0x00000001

  [ 18] 0xB0300000 0x0004E000 0x00410561 0x00000004 0xB0300000
  [ 19] 0xB034E000 0x00004000 0x01400772 0x00000004 0xB0300000
```

This tells us that there are 20 memory regions in use, and gives us the virtual address, the size, flags, device number, and inode for each one. Let's correlate this to the `pidin` output:

```
# pidin -p4105 mem
 pid tid name         prio STATE       code  data     stack
4105   1 sbin/pipe      10o RECEIVE       16K  148K  4096(132K)
4105   2 sbin/pipe      10o RECEIVE       16K  148K  4096(132K)
4105   4 sbin/pipe      10o RECEIVE       16K  148K  4096(132K)
4105   5 sbin/pipe      10o RECEIVE       16K  148K  4096(132K)
    ldqnx.so.2     @b0300000       312K   16K
```

Regions 0, 3, 6, 9 and 12

These are the guard pages at the end of the stacks, one for each of the five threads.

Regions 1, 4, 7, 10 and 13

These are the growth areas for the stacks, one for each of the five threads. This memory is physically allocated on demand; these regions serve to reserve the virtual address ranges. This corresponds to the "`(132K)`" from the `pidin` output.

Regions 2, 5, 8, 11 and 14

These are the in-use 4k stack segments, one for each of the five threads. Only four threads are alive — we'll discuss this below. This corresponds to the 4096 from the `pidin` output.

Region 15

This is the 16 KB of code for pipe.

Regions 16 and 17

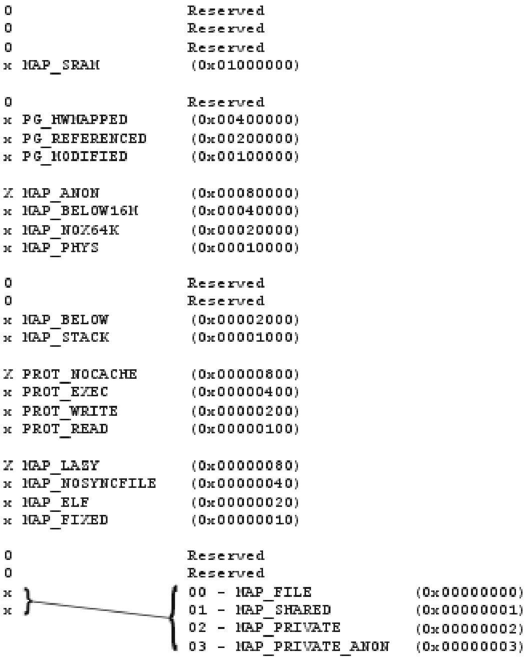
These are the data areas (4 KB and 144 KB, for a total of 148 KB).

Regions 18 and 19

These are for the shared object, ldqnx.so.2. Region 18 is the code area, region 19 is the data area. These correspond to the ldqnx.so.2 line from the pidin output.

The key to decoding the regions is to look at the *flags* member. You'll notice that there are two commands: DCMD_PROC_PAGEDATA and DCMD_PROC_MAPINFO. Both of these are used to obtain information about memory regions. However, DCMD_PROC_MAPINFO merges non-PG_* regions together, whereas DCMD_PROC_PAGEDATA lists them individually. This also implies that the three PG_* flags (PG_HWMAPPED, PG_REFERENCED, and PG_MODIFIED are valid only with DCMD_PROC_PAGEDATA).

The *flags* member is a bitmap, broken down as follows:



```
0                   Reserved
0                   Reserved
0                   Reserved
x MAP_SRAM           (0x01000000)

0                   Reserved
x PG_HWMAPPED        (0x00400000)
x PG_REFERENCED      (0x00200000)
x PG_MODIFIED        (0x00100000)

X MAP_ANON           (0x00080000)
x MAP_BELOW16M       (0x00040000)
x MAP_NOX64K         (0x00020000)
x MAP_PHYS           (0x00010000)

0                   Reserved
0                   Reserved
x MAP_BELOW          (0x00002000)
x MAP_STACK          (0x00001000)

X PROT_NOCACHE       (0x00000800)
x PROT_EXEC          (0x00000400)
x PROT_WRITE         (0x00000200)
x PROT_READ          (0x00000100)

X MAP_LAZY           (0x00000080)
x MAP_NOSYNCFILE     (0x00000040)
x MAP_ELF            (0x00000020)
x MAP_FIXED          (0x00000010)

0                   Reserved
0                   Reserved
x }                 { 00 - MAP_FILE              (0x00000000)
x }-----------------{ 01 - MAP_SHARED            (0x00000001)
                    { 02 - MAP_PRIVATE           (0x00000002)
                    { 03 - MAP_PRIVATE_ANON      (0x00000003)
```

Figure 36: Process flags.

By looking for a "tell-tale" flag, namely MAP_STACK (0x1000), I was able to find all of the stack segments (regions 0 through 14). Having eliminated those, regions 15, 18, and 19 are marked as PROT_EXEC (0x0400), so must be executable (the data area of the shared library is marked executable.) By process of elimination, regions 16 and 17 are not marked executable; therefore, they're data.

DCMD_PROC_TIMERS

We can find out about the timers that are associated with a process.

We use the DCMD_PROC_TIMERS command, and expect to get back zero or more data structures, as we did in the DCMD_PROC_PAGEDATA and DCMD_PROC_MAPINFO examples above. The structure is defined as follows:

```
typedef struct _debug_timer {
  timer_t             id;
  unsigned            spare;
  struct _timer_info  info;
} debug_timer_t;
```

This structure relies on the struct _timer_info type (defined in <sys/platform.h>, and paraphrased slightly):

```
struct _timer_info {
  struct _itimer      itime;
  struct _itimer      otime;
  _Uint32t            flags;
  _Int32t             tid;
  _Int32t             notify;
  clockid_t           clockid;
  _Uint32t            overruns;
  struct sigevent     event;
};
```

This data type, struct _timer_info is used with the *TimerInfo()* function call.

To fetch the data, we utilize code that's almost identical to that used for the memory segments (above):

```
#define MAX_TIMERS  512

static void
dump_procfs_timer (int fd, int pid)
{
  procfs_timer  timers [MAX_TIMERS];
  int           ntimers;
  int           i;

  // fetch information about the timers for this pid
  if (devctl (fd, DCMD_PROC_TIMERS, timers, sizeof (timers),
              &ntimers) != EOK) {
    fprintf (stderr, "%s: TIMERS err, proc %d, errno %d (%s)\n",
             progname, pid, errno, strerror (errno));
    exit (EXIT_FAILURE);
  }

  if (ntimers > MAX_TIMERS) {
    fprintf (stderr, "%s: proc %d has > %d timers (%d) !!!\n",
             progname, pid, MAX_TIMERS, ntimers);
    exit (EXIT_FAILURE);
  }

  printf ("Info from DCMD_PROC_TIMERS\n");
  for (i = 0; i < ntimers; i++) {
    // print information here
  }
  printf ("\n");
}
```

Since our pipe command doesn't use timers, let's look at the devb-eide driver instead. It has four timers; I've selected just one:

```
  Buffer   2 timer ID 2
    itime    1063180.652506618 s,      0.250000000 interval s
```

```
otime          0.225003825 s,      0.250000000 interval s
flags     0x00000001
tid       0
notify    196612 (0x30004)
clockid   0
overruns  0
event (sigev_notify type 4)
  SIGEV_PULSE (sigev_coid 1073741832,
               sigev_value.sival_int 0,
               sigev_priority -1, sigev_code 1)
```

The fields are as follows:

itime

This represents the time when the timer will fire, if it is active (i.e. the *flags* member has the bit _NTO_TI_ACTIVE set). If the timer is not active, but has been active in the past, then this will contain the time that it fired last, or was going to fire (in case the timer was canceled before firing).

otime

Time remaining before the timer expires.

flags

One or more of the flags _NTO_TI_ACTIVE (the timer is active), _NTO_TI_ABSOLUTE (the timer is waiting for an absolute time to occur; otherwise, the timer is considered relative), _NTO_TI_TOD_BASED (the timer is based relative to the beginning of the world (January 1, 1970, 00:00:00 GMT); otherwise, the timer is based relative to the time that QNX Neutrino was started on the machine (see the system page *qtime boot_time* member)), and _NTO_TI_EXPIRED (the timer has expired).

tid

The thread to which the timer is directed (or the value 0 if it's directed to the process).

notify

The notification type (only the bottom 16 bits are interesting; the rest are used internally).

clockid

This is the clock ID (e.g. CLOCK_REALTIME).

overruns

This is a count of the number of timer overruns.

event

This is a struct sigevent that indicates the type of event that should be delivered when the timer fires. For the example above, it's a SIGEV_PULSE, meaning that a pulse is sent. The fields listed after the SIGEV_PULSE pertain to the pulse delivery type (e.g. connection ID, etc.).

In the example above, the *flags* member has only the bit _NTO_TI_ACTIVE (the value 0x0001) set, which means that the timer is active. Since the _NTO_TI_TOD_BASED flag is not set, however, it indicates that the timer is relative to the time that the machine was booted. So the next time the timer will fire is 1063180.652506618 seconds past the time that the machine was booted (or 12 days, 7 hours, 19 minutes, and 40.652506618 seconds past the boot time). This timer might be used for flushing the write cache — at the time the snapshot was taken, the machine had already been up for 12 days, 7 hours, 19 minutes, and some number of seconds.

The *notify* type (when examined in hexadecimal) shows 0x0004 as the bottom 16 bits, which is a notification type of SIGEV_PULSE (which agrees with the data in the *event* structure).

DCMD_PROC_IRQS

Finally, we can also find out about the interrupts that are associated with a process.

We use the DCMD_PROC_IRQS command, and expect to get back zero or more data structures, as we did in the DCMD_PROC_PAGEDATA, DCMD_PROC_MAPINFO, and DCMD_PROC_TIMERS examples above. The structure procfs_irq is the same as the debug_irq_t, which is defined as follows:

```
typedef struct _debug_irq {
  pid_t             pid;
  pthread_t         tid;
  const struct sigevent *(*handler)(void *area, int id);
  void              *area;
  unsigned          flags;
  unsigned          level;
  unsigned          mask_count;
  int               id;
  unsigned          vector;
  struct sigevent   event;
} debug_irq_t;
```

To fetch the data, we use code similar to what we used with the timers and memory segments:

```
#define MAX_IRQS  512

static void
dump_procfs_irq (int fd, int pid)
{
  procfs_irq    irqs [MAX_IRQS];
  int           nirqs;
  int           i;

  // fetch information about the IRQs for this pid
  if (devctl (fd, DCMD_PROC_IRQS, irqs, sizeof (irqs),
             &nirqs) != EOK) {
    fprintf (stderr, "%s:  IRQS proc %d, errno %d (%s)\n",
             progname, pid, errno, strerror (errno));
    exit (EXIT_FAILURE);
  }

  if (nirqs > MAX_IRQS) {
    fprintf (stderr, "%s: proc %d > %d IRQs (%d) !!! ***\n",
             progname, pid, MAX_IRQS, nirqs);
    exit (EXIT_FAILURE);
  }

  printf ("Info from DCMD_PROC_IRQS\n");
  for (i = 0; i < nirqs; i++) {
    // print information here
  }
  printf ("\n");
}
```

Since our `pipe` command doesn't use interrupts either, I've once again selected `devb-eide`:

```
Info from DCMD_PROC_IRQS
  Buffer       0
    pid        6
    tid        2
    handler    0x00000000
    area       0xEFFF7168
    flags      0x0000000E
    level      14
    mask_count 0
    id         2
    vector     14
    event (sigev_notify type 4)
      SIGEV_PULSE (sigev_coid 1073741826,
                   sigev_value.sival_int 0,
                   sigev_priority 21, sigev_code 2)
  Buffer       1
    pid        6
    tid        3
    handler    0x00000000
    area       0xEFFF71E0
    flags      0x0000000E
    level      15
    mask_count 0
    id         3
    vector     15
    event (sigev_notify type 4)
      SIGEV_PULSE (sigev_coid 1073741829,
                   sigev_value.sival_int 0,
                   sigev_priority 21, sigev_code 2)
```

The members of the `debug_irq_t` shown above are as follows:

pid **and** ***tid***

The *pid* and *tid* fields give the process ID and the thread ID (obviously, it will be process ID 6, which is `devb-eide`).

handler **and** ***area***

Indicates the interrupt service routine address, and its associated parameter. The fact that the interrupt handler address is zero indicates that there is no real interrupt vector associated with the interrupts; rather, the *event* (a pulse in both cases) should be returned (i.e. the interrupt was attached with *InterruptAttachEvent()* rather than *InterruptAttach()*). In the case of the *handler* being zero, the *area* member is not important.

flags

Copyright © 2014, QNX Software Systems Limited

The *flags* value is hexadecimal `0x0E`, which is composed of the bits `_NTO_INTR_FLAGS_SMPRDY`, `_NTO_INTR_FLAGS_PROCESS`, and `_NTO_INTR_FLAGS_TRK_MSK`, meaning, respectively, that the interrupt handler is SMP-capable (this flag is deprecated in version 6.3; everything must be SMP-ready then), the interrupt belongs to the process (rather than the thread), and the number of times the interrupt is masked and unmasked should be kept track of.

level* and *vector

This is the interrupt level and vector for this particular interrupt. For an x86 architecture, they happen to be the same number. (The *level* is an internal kernel number and has no meaning for the end-user.) In our example, `devb-eide` is attached to two interrupt sources (as defined by the *vector* parameter; i.e. interrupts 14 and 15, the two EIDE controllers on my PC).

mask_count

Indicates the number of times the interrupt is masked (0 indicates the interrupt is not masked). Useful as a diagnostic aid when you are trying to determine why your interrupt fires only once. `:-)`

id

This is the interrupt identification number returned by *InterruptAttach()* or *InterruptAttachEvent()*.

event

A standard `struct sigevent` that determines what the *InterruptAttachEvent()* should do when it fires.

Finding out information about the threads

Even though we can get a lot of information about processes (above), in QNX Neutrino a process doesn't actually *do* anything on its own — it acts as a container for multiple threads. Therefore, to find out about the threads, we can call *devctl()* with the following commands:

`DCMD_PROC_TIDSTATUS`

This command gets most of the information about a thread, and also sets the “current thread” that's used for subsequent operations (except the next two in this list).

`DCMD_PROC_GETGREG`

This returns the general registers for the current thread.

DCMD_PROC_GETFPREG

This returns the floating-point registers for the current thread.

There are other commands available for manipulating the thread status (such as starting or stopping a thread, etc.), which we won't discuss here.

First we need a way of iterating through all the threads in a process. Earlier in this chapter, we called out to a function *do_process()*, which was responsible for the “per-process” processing. Let's now see what this function does and how it relates to finding all the threads in the process:

```
void
do_process (int pid, int fd, char *name)
{
  procfs_status   status;

  printf ("PROCESS ID %d\n", pid);

  // dump out per-process information
  dump_procfs_info (fd, pid);
  dump_procfs_map_info (fd, pid);
  dump_procfs_timer (fd, pid);
  dump_procfs_irq (fd, pid);

  // now iterate through all the threads
  status.tid = 1;
  while (1) {
    if (devctl (fd, DCMD_PROC_TIDSTATUS, &status,
                sizeof (status), 0) != EOK) {
      break;
    } else {
      do_thread (fd, status.tid, &status);
      status.tid++;
    }
  }
}
```

The *do_process()* function dumps out all the per-process information that we discussed above, and then iterates through the threads, calling *do_thread()* for each one. The trick here is to start with thread number 1 and call the *devctl()* with `DCMD_PROC_TIDSTATUS` until it returns something other than `EOK`. (QNX Neutrino starts numbering threads at “1.”)

The magic that happens is that the kernel will return information about the thread specified in the *tid* member of *status if it has it*; otherwise, it will return information on the *next available* thread ID (or return something other than `EOK` to indicate it's done).

The DCMD_PROC_TIDSTATUS command

The `DCMD_PROC_TIDSTATUS` command returns a structure of type `procfs_status`, which is equivalent to `debug_thread_t`:

```
typedef struct _debug_thread_info {
  pid_t       pid;
  pthread_t   tid;
  _Uint32t    flags;
  _Uint16t    why;
  _Uint16t    what;
  _Uint64t    ip;
  _Uint64t    sp;
  _Uint64t    stkbase;
  _Uint64t    tls;
  _Uint32t    stksize;
  _Uint32t    tid_flags;
  _Uint8t     priority;
  _Uint8t     real_priority;
  _Uint8t     policy;
  _Uint8t     state;
  _Int16t     syscall;
  _Uint16t    last_cpu;
  _Uint32t    timeout;
  _Int32t     last_chid;
  sigset_t    sig_blocked;
  sigset_t    sig_pending;
  siginfo_t   info;

  // blocking information deleted (see next section)

  _Uint64t    start_time;
  _Uint64t    sutime;
} debug_thread_t;
```

More information than you can shake a stick at (224 bytes)! Here are the fields and their meanings:

pid **and** ***tid***

The process ID and the thread ID.

flags

Flags indicating characteristics of the thread (see `<sys/debug.h>` and look for the constants beginning with `_DEBUG_FLAG_`).

why **and** ***what***

The *why* indicates why the thread was stopped (see `<sys/debug.h>` and look for the constants beginning with `_DEBUG_WHY_`) and the *what* provides additional information for the *why* parameter. For `_DEBUG_WHY_TERMINATED`, the *what* variable contains the exit code value, for `_DEBUG_WHY_SIGNALLED` and `_DEBUG_WHY_JOBCONTROL`, *what* contains the signal number, and for `_DEBUG_WHY_FAULTED`, *what* contains the fault number (see `<sys/fault.h>` for the values).

ip

The current instruction pointer where this thread is executing.

sp

The current stack pointer for the thread.

stkbase **and** ***stksize***

The base of the thread's stack, and the stack size.

tls

The Thread Local Storage (TLS) data area. See `<sys/storage.h>`.

tid_flags

See <sys/neutrino.h> constants beginning with _NTO_TF.

priority* and *real_priority

The *priority* indicates thread priority used for scheduling purposes (may be boosted), and the *real_priority* indicates the actual thread priority (not boosted).

policy

The scheduling policy (e.g. FIFO, Round Robin).

state

The current state of the thread (see <sys/states.h>, e.g. STATE_MUTEX if blocked waiting on a mutex).

syscall

Indicates the last system call that the thread made (see <sys/kercalls.h>).

last_cpu

The last CPU number that the thread ran on (for SMP systems).

timeout

Contains the *flags* parameter from the last *TimerTimeout()* call.

last_chid

The last channel ID that this thread *MsgReceive()*'d on. Used for priority boosting if a client does a *MsgSend()* and there are no threads in STATE_RECEIVE on the channel.

sig_blocked*, *sig_pending*, and *info

These fields all relate to signals — recall that signals have a process aspect as well as a thread aspect. The *sig_blocked* indicates which signals this thread has blocked. Similarly, *sig_pending* indicates which signals are pending on this thread. The *info* member carries the information for a *sigwaitinfo()* function.

start_time

The time, in nanoseconds since January 1, 1970, that the thread was started. Useful for detecting thread ID reuse.

Copyright © 2014, QNX Software Systems Limited

sutime

Thread's system and user running times (in nanoseconds).

Blocked thread information

When a thread is blocked, there's an additional set of fields that are important (they are within the `debug_thread_t`, above, where the comment says “blocking information deleted”). The deleted content is:

```
union {

  struct {
    pthread_t   tid;
  } join;

  struct {
    _Int32t     id;
    _Uintptrt   sync;
  } sync;

  struct {
    _Uint32t    nd;
    pid_t       pid;
    _Int32t     coid;
    _Int32t     chid;
    _Int32t     scoid;
  } connect;

  struct {
    _Int32t     chid;
  } channel;

  struct {
    pid_t       pid;
    _Uintptrt   vaddr;
    _Uint32t    flags;
  } waitpage;

  struct {
    _Uint32t    size;
  } stack;

} blocked;
```

As you can see, there are six major structures (`join`, `sync`, `connect`, `channel`, `waitpage`, and `stack`) that are unioned together (because a thread can be in only one given blocking state at a time):

join

When a thread is in `STATE_JOIN`, it's waiting to synchronize to the termination of another thread (in the same process). This thread is waiting for the termination of the thread identified by the *tid* member.

sync

When a thread is blocked on a synchronization object (such as a mutex, condition variable, or semaphore), the *id* member indicates the virtual address of the object, and the *sync* member indicates the type of object.

Copyright © 2014, QNX Software Systems Limited

connect

Indicates who the thread is blocked on (used with STATE_SEND and STATE_REPLY).

channel

Indicates the channel ID the thread is blocked in (used with STATE_RECEIVE).

waitpage

Indicates the virtual address that the thread is waiting for to be lazy-mapped in (used with STATE_WAITPAGE).

stack

Used with STATE_STACK, indicates the thread is waiting for *size* bytes of virtual address space to be made available for the stack.

The DCMD_PROC_GETGREG and DCMD_PROC_GETFPREG commands

These two commands are used to fetch the current general registers and floating-point registers for the thread.

This will, of course, be architecture-specific. For simplicity, I've shown the x86 version, and just the general registers.

The data structure is (slightly edited for clarity):

```
typedef union _debug_gregs {
  X86_CPU_REGISTERS       x86;
  ARM_CPU_REGISTERS       arm;
  _Uint64t                padding [1024];
} debug_greg_t;
```

The x86 version, (the *x86* member), is as follows (from <x86/context.h>):

```
typedef struct x86_cpu_registers {
  _Uint32t  edi, esi, ebp, exx, ebx, edx, ecx, eax;
  _Uint32t  eip, cs, efl;
  _Uint32t  esp, ss;
} X86_CPU_REGISTERS;
```

To get the information, a simple *devctl()* is issued:

```
static void
dump_procfs_greg (int fd, int tid)
{
  procfs_greg   g;
  int           sts;

  // set the current thread first!
  if ((sts = devctl (fd, DCMD_PROC_CURTHREAD, &tid,
                     sizeof (tid), NULL)) != EOK) {
    fprintf (stderr, "%s:  CURTHREAD for tid %d, sts %d (%s)\n",
             progname, tid, sts, strerror (sts));
    exit (EXIT_FAILURE);
  }
```

Copyright © 2014, QNX Software Systems Limited

```
  // fetch information about the registers for this pid/tid
  if ((sts = devctl (fd, DCMD_PROC_GETGREG, &g,
                     sizeof (g), NULL)) != EOK) {
    fprintf (stderr, "%s:  GETGREG information, sts %d (%s)\n",
             progname, sts, strerror (sts));
    exit (EXIT_FAILURE);
  }

  // print information here...
}
```

This call, unlike the other calls mentioned so far, requires you to call *devctl()* with DCMD_PROC_CURTHREAD to set the current thread. Otherwise, you'll get an EINVAL return code from the *devctl()*.

Here is some sample output:

```
Info from DCMD_PROC_GETGREG
  cs  0x0000001D   eip  0xF000EF9C
  ss  0x00000099   esp  0xEFFF7C14
  eax 0x00000002   ebx  0xEFFFEB00
  ecx 0xEFFF7C14   edx  0xF000EF9E
  edi 0x00000000   esi  0x00000000
  ebp 0xEFFF77C0   exx  0xEFFFEBEC
  efl 0x00001006
```

References

The following references apply to this chapter:

Header files

- `<sys/procfs.h>` — contains the *devctl()* command constants (e.g., `DCMD_PROC_GETGREG`) used to fetch information.
- `<sys/states.h>` — defines the states that a thread can be in (see also “Thread life cycle” in the *System Architecture* guide).
- `<sys/syspage.h>` — defines the structure of the system page, which contains system-wide items of interest (e.g. the *boot_time* member of the *qtime* structure that tells you when the machine was booted).
- `<sys/debug.h>` — defines the layout of the various structures in which information is returned.

Functions

See the following functions in the QNX Neutrino *C Library Reference*:

- *opendir()* and *readdir()* (to access the `/proc` directory)
- *devctl()* to issue the commands

Utilities

See `pidin` in the *Utilities Reference*.

Copyright © 2014, QNX Software Systems Limited

Appendix C
Sample Programs

We've put copies of the code discussed in this book in an archive that you can get from the Download area of *http://www.qnx.com/*. This archive includes all of the samples from the book.

The top-level directories in the archive include:

web/

The web counter source code, in three phases (one directory for each version).

adios/

The ADIOS (Analog/Digital I/O Server) utility, including showsamp, tag, and the required libraries.

fsys/

The RAM Disk filesystem, the .tar filesystem, and a library used by both.

procfs/

The procfs utility from the /proc filesystem appendix.

Most directories include the following:

- version.c, a string that states the version number of the program
- main.use, the usage message
- Makefile

Web-Counter resource manager

The Web Counter is presented in three design phases. The `web` directory includes a `Makefile` for the entire project, as well as a directory for each phase.

`phase-1/`

Phase I consists of the following files:

- `7seg.h`—manifest constants and declarations for the 7-segment render module
- `7seg.c`—a 7-segment LED simulator that draws images into a memory buffer
- `gif.h`—manifest constants and declarations for the GIF encoder
- `main.c`—main module

You'll also need a GIF encoder.

`phase-2/`

Phase 2 adds the following:

- `8x8.h`—manifest constants and declarations for the 8 by 8 dot render module
- `8x8.c`—an 8 by 8 “dot” font generator

`phase-3/`

Phase 3 includes updated versions of the files from phases 1 and 2.

Copyright © 2014, QNX Software Systems Limited

ADIOS — Analog / Digital I/O Server

The ADIOS project consists of the ADIOS server, drivers for the various hardware cards, a library, API headers, and some utilities.

adios.cfg

A test configuration file.

adios/

The server consists of the following:

- adios.h—manifest constants and declarations for the Analog / Digital I/O Server (ADIOS)
- daq.h—manifest constants and declarations for the data acquisition component
- iofuncs.h—manifest constants and declarations for the I/O and connect functions for the resource manager
- adios.c—the Analog / Digital I/O Server (ADIOS) module
- daq.c—shared memory region handler and the data acquisition thread
- iofuncs.c—I/O and connect functions for the resource manager
- main.c—main module for the ADIOS data acquisition server

api/

API header files:

- api.h—manifest constants and declarations for clients of ADIOS
- driver.h—manifest constants and declarations for ADIOS drivers

lib/

The ADIOS library consists of the following files:

- client.h—manifest constants and declarations for the client data acquisition library
- parser.h—manifest constants and declarations for the parser module used for ADIOS and its drivers
- client.c—library functions for the client
- parser.c—module used to parse the configuration file for ADIOS and its drivers

Drivers

The source code for the drivers is in the following directories:

- `dio144/`
- `iso813/`
- `pcl711/`
- `sim000/`

`utils/`

The utilities include:

- `showsamp.c`
- `tag.c`

Each utility has a usage message in a separate file.

Copyright © 2014, QNX Software Systems Limited

RAM-disk and tar filesystem managers

These managers are in the same directory (fsys) in the archive because they rely on a common library.

fsys/lib/

The library includes the following:

- block.h—manifest constants and declarations for the block functions
- cfs.h—constants and declarations for the captive filesystem library
- mpool.h—manifest constants and declarations for the memory pool module
- block.c—block functions that deal with the data storage blocks
- cfs_attr_init.c, cfs_func_init.c—the captive filesystem utility library's initialization functions
- io_close.c, io_devctl.c, io_space.c—handler functions for the captive filesystem utility library
- mpool.c—memory pool library

The captive filesystem utility library isn't thread-safe, but the memory pool library is.

fsys/ramdisk/

The RAM-disk filesystem includes the following:

- cfs.h, ramdisk.h—manifest constants and declarations for the RAMDISK filesystem
- attr.c—attribute-handling module
- c_link.c, c_mknod.c, c_mount.c, c_open.c, c_readlink.c, c_rename.c, c_unlink.c—handlers for the connect functions
- debug.c—functions to help with debugging
- dirent.c—functions that handle directory entries
- io_read.c, io_write.c—handlers for the captive filesystem I/O functions
- main.c—main module
- pathname.c—functions that handle pathnames
- ramdisk_io_read.c, ramdisk_io_write.c—handlers for the RAM-disk I/O functions
- utils.c—utility functions

fsys/tarfs/

The TAR filesystem and its mount_tarfs helper program consist of the following:

- cfs.h, tarfs.h—manifest constants and declarations for the TAR filesystem
- attr.c—attribute-handling module
- c_link.c, c_mknod.c, c_mount.c, c_open.c, c_readlink.c, c_rename.c, c_unlink.c—handlers for the connect functions
- debug.c—functions to help with debugging
- dirent.c—functions that handle directory entries
- io_read.c, io_write.c— handlers for the I/O functions
- m_main.c—main module for the mount_tarfs utility
- main.c—main module for the TAR filesystem
- pathname.c—functions that handle pathnames
- tarfs.c—support routines
- tarfs_io_read.c—the io_read handler
- utils.c—utility functions

Copyright © 2014, QNX Software Systems Limited

The /proc filesystem

The /proc filesystem utility module is in procfs/procfs.c. This program dumps just about everything it can get its hands on from this filesystem.

Glossary

5 nines (high availability)

A system that's characterized as having a “5 nines” *availability* rating (99.999%). This means that the system has a downtime of just 5 minutes per year. A “6 nines” system will have a downtime of just 20 minutes every *forty* years. This is also known variously in the industry as “carrier-class” or “telco-class” availability.

analog (data acquisition)

Indicates an input or output signal that corresponds to a range of voltages. In the real world, an analog signal is continuously variable, meaning that it can take on any value within the range. When the analog value is used with a data acquisition card, it will be digitized, meaning that only a finite number of discrete values are represented. Analog inputs are digitized by an “analog to digital” (A/D) convertor, and analog outputs are synthesized by a “digital to analog” (D/A) convertor. Generally, most convertors have an accuracy of 8, 12, 16, or more bits. Compare *digital*.

asynchronous

Used to indicate that a given operation is not synchronized to another operation. For example, a *pulse* is a form of asynchronous *message-passing* in that the sender is not blocked waiting for the message to be received. Contrast with *synchronous*. See also *blocking* and *receive a message*.

availability (high availability)

Availability is a ratio that expresses the amount of time that a system is “up” and available for use. It is calculated by taking the *MTBF* and dividing it by the sum of the MTBF plus the *MTTR*, and is usually expressed as a percentage. To increase a system's availability, you need to raise the MTBF and/or lower the MTTR. The availability is usually stated as the number of leading 9s in the ratio (see *5 nines*). An availability of 100% (also known as *continuous availability*) is extremely difficult, if not impossible, to attain because that would imply that the value of MTTR was zero (and the availability was just MTBF divided by MTBF, or 1) or that the MTBF was infinity.

blocking

A means for a *thread* to synchronize with other threads or events. In the blocking state (of which there are about a dozen), a thread doesn't consume any CPU — it's waiting on a list maintained within the kernel. When the

Copyright © 2014, QNX Software Systems Limited

event that the thread was waiting for occurs (or a *timeout* occurs), the thread is unblocked and is able to consume CPU again. See also *unblock*.

cascade failure (high availability)

A cascade failure is one in which several modules fail as the result of a single failure. A good example of this is if a *process* is using a driver, and the driver fails. If the process that used the driver isn't fault tolerant, then it too may fail. If other processes that depend on this driver aren't fault tolerant, then they too will fail. This chain of failures is called a “cascade failure.” The North American power outage of August 14, 2003 is a good example. See also *fault tolerance*.

client (message-passing)

QNX Neutrino's *message-passing* architecture is structured around a client/*server* relationship. In the case of the client, it's the one requesting services of a server. The client generally accesses these services using standard file-descriptor-based function calls (e.g., *lseek()*), which are *synchronous*, in that the client's call doesn't return until the request is completed by the server. A *thread* can be both a client and a server at the same time.

code (memory)

A code *segment* is one that is executable. This means that instructions can be executed from it. Also known as a “text” segment. Contrast with *data* or *stack* segments.

cold standby (high availability)

Cold standby mode refers to the way a failed software component is restarted. In cold-standby mode, the software component is generally restarted by loading it from media (disk, network), having the component go through initializations, and then having the component advertise itself as ready. Cold standby is the slowest of the three modes (cold, warm, and hot), and, while its timing is system specific, it usually takes on the order of tens of milliseconds to seconds. Cold standby is the simplest standby model to implement, but also the one that impacts *MTTR* the most negatively. See also *hot standby*, *restartability*, and *warm standby*.

continuous availability (high availability)

A system with an *availability* of 100%. The system has *no* downtime, and as such, is difficult, if not impossible, to attain with moderately complex systems. The reason it's difficult to attain is that every piece of software,

Copyright © 2014, QNX Software Systems Limited

hardware, and infrastructure has some kind of failure rate. There is always some non-zero probability of a catastrophic failure for the system as a whole.

data (memory)

A data *segment* is one that is not executable. It's typically used for storing data, and as such, can be marked read-only, write-only, read/write, or no access. Contrast with *code* or *stack* segments.

deadlock

A failure condition reached when two threads are mutually blocked on each other, with each thread waiting for the other to respond. This condition can be generated quite easily; simply have two threads send each other a message — at this point, both threads are waiting for the other thread to reply to the request. Since each thread is blocked, it will not have a chance to reply, hence deadlock. To avoid deadlock, clients and servers should be structured around a *send hierarchy*. (Of course, deadlock can occur with more than two threads; A sends to B, B sends to C, and C sends back to A, for example.) See also *blocking*, *client*, *reply to a message*, *send a message*, *server*, and *thread*.

digital (data acquisition)

Indicates an input or output signal that has two states only, usually identified as on or off (other names are commonly used as well, “energized” and “de-energized” for example). Compare *analog*.

exponential backoff (high availability)

A *policy* that's used to determine at what intervals a *process* should be restarted. Its use is to prevent overburdening the system in case a component keeps failing. See also *restartability*.

fault tolerance (high availability)

A term used in conjunction with high availability that refers to a system's ability to handle a fault. When a fault occurs in a fault-tolerant system, the software is able to work around the fault, for example, by retrying the operation or switching to an alternate server. Generally, fault tolerance is incorporated into a system to avoid cascade failures. See also *cascade failure*.

guard page (stack)

An inaccessible data area present at the end of the valid virtual address range for a stack. The purpose of the guard page is to cause a memory-access exception should the stack overflow past its defined range.

HA (or high availability)

A designation applied to a system to indicate that it has a high level of *availability*. A system that's designed for high-availability needs to consider cascade failures, *restartability*, and *fault tolerance*. Generally speaking, a system designated as “high availability” will have an availability of *5 nines* or better. See also *cascade failure*.

hot standby (high availability)

Hot-standby mode refers to the way in which a failed software component is restarted. In hot-standby mode, the software component is actively running, and effectively shadows the state of the *primary process*. The primary process feeds it updates, so that the *secondary* (or “standby”) process is ready to take over the instant that the primary process fails. Hot standby is the fastest, but most expensive to implement of the three modes (cold, warm, and hot), and, while its timing is system specific, is usually thought of as being on the order of microseconds to milliseconds. Hot standby is very expensive to implement, because it must continually be shadowing the data updates from the primary process, and must be able to assume operation when the primary dies. Hot standby, however, is the preferred solution to minimizing *MTTR* and hence increasing *availability*. See also *cold standby*, *restartability*, and *warm standby*.

in-service upgrade or ISU (high availability)

An upgrade performed on a live system, with the least amount of impact to the operation of the system. The basic algorithm is to simulate a fault and then, instead of having the *overlord* process restart the failed component, it instead starts a new version. In certain cases, the *policy* of the overlord may be to perform a version downgrade instead of an upgrade. See also *fault tolerance* and *restartability*.

message-passing

The QNX Neutrino operating system is based on a message-passing model, where all services are provided in a *synchronous* manner by passing messages around from *client* to *server*. The client will *send a message* to the server and block. The server will *receive a message* from the client, perform some amount of processing, and then reply to the client's message, which will *unblock* the client. See also *blocking* and *reply to a message*.

MTBF or Mean Time Between Failures (high availability)

The MTBF is expressed in hours and indicates the mean time that elapses between failures. MTBF is applied to both software and hardware, and is used, in conjunction with the *MTTR*, in the calculation of *availability*. A computer backplane, for example, may have an MTBF that's measured in

Copyright © 2014, QNX Software Systems Limited

the tens of thousands of hours of operation (several years). Software usually has a lower MTBF than hardware.

MTTR or Mean Time To Repair (high availability)

The MTTR is expressed in hours, and indicates the mean time required to repair a system. MTTR is applied to both software and hardware, and is used, in conjunction with the *MTBF*, in the calculation of *availability*. A server, for example, may have an MTTR that's measured in milliseconds, whereas a hardware component may have an MTTR that's measured in minutes or hours, depending on the component. Software usually has a much lower MTTR than hardware.

overlord (high availability)

A *process* responsible for monitoring the stability of various system processes, according to the *policy*, and performing actions (such as restarting processes based on a restart policy). The overlord may also be involved with an *in-service upgrade* or downgrade. See also *restartability*.

policy (high availability)

A set of rules used in a high-availability system to determine the limits that are enforced by the *overlord* process against other processes in the system. The policy also determines how such processes are restarted, and may include algorithms such as *exponential backoff*. See also *restartability*.

primary (high availability)

The “primary” designation refers to the active process when used in discussions of cold, warm, and hot standby. The primary system is running, and the *secondary* system(s) is/are the “backup” system(s). See also *cold standby*, *warm standby*, and *hot standby*.

process (*noun*)

A non-schedulable entity that occupies memory, effectively acting as a container for one or more threads. See also *thread*.

pulse (message-passing)

A nonblocking message received in a manner similar to a regular message. It is non-blocking for the sender, and can be waited on by the receiver using the standard message-passing functions *MsgReceive()* and *MsgReceivev()* or the special pulse-only receive function *MsgReceivePulse()*. While most messages are typically sent from *client* to *server*, pulses are generally sent in the opposite direction, so as not to break the *send hierarchy* (which could cause *deadlock*). See also *receive a message*.

QNX Software Systems

The company responsible for the QNX 2, QNX 4, and QNX Neutrino operating systems.

QSS

An abbreviation for *QNX Software Systems*.

receive a message (message-passing)

A *thread* can receive a message by calling *MsgReceive()* or *MsgReceivev()*. If there is no message available, the thread will block, waiting for one. A thread that receives a message is said to be a *server*. See also *blocking*.

reply to a message (message-passing)

A *server* will reply to a client's message to deliver the results of the client's request back to the client, and *unblock* the client. See also *client*.

resource manager

A *server process* that provides certain well-defined file-descriptor-based services to arbitrary clients. A resource manager supports a limited set of messages that correspond to standard client C library functions such as *open()*, *read()*, *write()*, *lseek()*, *devctl()*, etc. See also *client*.

restartability (high availability)

The characteristic of a system or *process* that lets it be gracefully restarted from a faulted state. Restartability is key in lowering *MTTR*, and hence in increasing *availability*. The *overlord* process is responsible for determining that another process has exceeded some kind of limit, and then, based on the *policy*, the overlord process may be responsible for restarting the component.

secondary (or standby) (high availability)

Refers to the inactive *process* when used in discussions of cold, warm, and hot standby. The *primary* system is the one that's currently running; the secondary system is the “backup” system. There may be more than one secondary process. See also *cold standby*, *warm standby*, and *hot standby*.

segment (memory)

A contiguous “chunk” of memory with the same accessibility permissions throughout. Note that this is different from the (now archaic) x86 term, which indicated something accessible via a segment register. In this definition, a segment can be of an arbitrary size. Segments typically represent *code* (or “text”), *data*, *stack*, or other uses.

Copyright © 2014, QNX Software Systems Limited

send a message (message-passing)

A *thread* can send a message to another thread. The *MsgSend*()* series of functions are used to send the message; the sending thread blocks until the receiving thread replies to the message. A thread that sends a message is said to be a *client*. See also *blocking*, *message-passing*, and *reply to a message*.

send hierarchy

A design paradigm where messages are sent in one direction, and replies flow in the opposite direction. The primary purpose of having a send hierarchy is to avoid *deadlock*. A send hierarchy is accomplished by assigning clients and servers a "level," and ensuring that messages that are being sent go only to a higher level. This avoids the potential for deadlock where two threads would send to each other, because it would violate the send hierarchy — one thread should not have sent to the other thread, because that other thread must have been at a lower level. See also *client*, *reply to a message*, *send a message*, *server*, and *thread*.

server (message-passing)

A regular, user-level *process* that provides certain types of functionality (usually file-descriptor-based) to clients. Servers are typically resource managers. QNX Neutrino provides an extensive library that performs much of the functionality of a resource manager for you. The server's job is to receive messages from clients, process them, and then reply to the messages, which unblocks the clients. A *thread* within a process can be both a client and a server at the same time. See also *client*, *receive a message*, *reply to a message*, *resource manager*, and *unblock*.

stack (memory)

A stack *segment* is one used for the stack of a *thread*. It generally is placed at a special virtual address location, can be grown on demand, and has a *guard page*. Contrast with *data* or *code* segments.

synchronous

Used to indicate that a given operation has some synchronization to another operation. For example, during a message-passing operation, when the *server* does a *MsgReply()* (to reply to the *client*), unblocking the client is said to be synchronous to the reply operation. Contrast with *asynchronous*. See also *message-passing* and *unblock*.

timeout

Copyright © 2014, QNX Software Systems Limited

Many kernel calls support the concept of a timeout, which limits the time spent in a blocked state. The blocked state will be exited if whatever condition was being waited upon has been satisfied, or the timeout time has elapsed. See also *blocking*.

thread

A single, schedulable, flow of execution. Threads are implemented directly within the QNX Neutrino kernel and are manipulated by the POSIX *pthread*()* function calls. A thread will need to synchronize with other threads (if any) by using various synchronization primitives such as mutexes, condition variables, semaphores, etc. Threads are scheduled in either FIFO or Round Robin scheduling mode. A thread is always associated with a *process*.

unblock

A *thread* that had been blocked will be unblocked when the condition it has been blocked on is met, or a *timeout* occurs. For example, a thread might be blocked waiting to *receive a message*. When the message is sent, the thread will be unblocked. See also *blocking* and *send a message*.

warm standby (high availability)

Warm-standby mode refers to the way a failed software component is restarted. In warm-standby mode, the software component is lying in a “dormant” state, perhaps having performed some rudimentary initialization. The component is waiting for the failure of its *primary* component; when that happens, the component completes its initializations, and then advertises itself as being ready to serve requests. Warm standby is the middle-of-the-road version of the three modes (cold, warm, and hot). While its timing is system-specific, this is usually thought of as being on the order of milliseconds. Warm standby is relatively easy to implement, because it performs its usual initializations (as if it were running in primary mode), then halts and waits for the failure of the primary before continuing operation. See also *cold standby*, *hot standby*, *restartability*, and *server*.

Copyright © 2014, QNX Software Systems Limited

Index

Copyright © 2014, QNX Software Systems Limited

Copyright © 2014, QNX Software Systems Limited

Copyright © 2014, QNX Software Systems Limited

Copyright © 2014, QNX Software Systems Limited

Copyright © 2014, QNX Software Systems Limited

Copyright © 2014, QNX Software Systems Limited

Copyright © 2014, QNX Software Systems Limited

Copyright © 2014, QNX Software Systems Limited