Getting Started with QNX® Neutrino®: A Guide for Realtime Programmers

BlackBerry | QNX

©1999–2020, QNX Software Systems Limited, a subsidiary of BlackBerry Limited. 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/

Trademarks, including but not limited to BLACKBERRY, EMBLEM Design, QNX, MOMENTICS, NEUTRINO, and QNX CAR, are the trademarks or registered trademarks of BlackBerry Limited, its subsidiaries and/or affiliates, used under license, and the exclusive rights to such trademarks are expressly reserved. All other trademarks are the property of their respective owners.

Patents per 35 U.S.C. § 287(a) and in other jurisdictions, where allowed: http://www.blackberry.com/patents

Electronic edition published: March 06, 2020

Contents

About This Guide....9
Typographical conventions....10
Technical support....12
Foreword to the First Edition by Peter van der Veen....13
Preface to the First Edition by Rob Krten....15
A little history....16
Who this book is for....17
What's in this book?....18
About Rob Krten....20
Acknowledgments....21

Chapter 1: Processes and Threads....23
Process and thread fundamentals....24
A process as a house....24
The occupants as threads....24
Back to processes and threads....24
Mutual exclusion....25
Priorities....25
Semaphores....26
A semaphore as a mutex....27
The kernel's role....28
Single CPU....28
Multiple CPU (SMP)....28
The kernel as arbiter....28
Kernel states....32
Threads and processes....34
Why processes?....34
Starting a process....35
Starting a thread....46
More on synchronization....68
Readers/writer locks....68
Sleepon locks....70
Condition variables....74
Additional OS services....80
Pools of threads....80
An example of synchronization....88
Scheduling and the real world....96
Rescheduling: hardware interrupts....96
Rescheduling: kernel calls....96
Rescheduling: exceptions....97
Summary....97

Chapter 2: Message Passing....99

A small microkernel and message passing....100
Message passing and client/server....102
Network-distributed message passing....105
What it means for you....106
Multiple threads....107
Server/subserver....107
Some examples....109
Using message passing....111
Architecture & structure....111
The client....112
The server....114
Identifying messages....118
The send-hierarchy....119
Receive IDs, channels, and other parameters....119
Multipart messages....129
Pulses....135
Receiving a pulse message....135
The *MsgDeliverEvent()* function....138
Channel flags....139
Message passing over a network....145
Networked message passing differences....147
Some notes on NDs....148
Priority inheritance....151
So what's the trick?....153
Summary....153

Chapter 3: Clocks, Timers, and Getting a Kick Every So Often....155
Clocks and timers....156
Operating periodically....156
Clock interrupt sources....158
Base timing resolution....158
Timing jitter....159
Types of timers....160
Notification schemes....160
Using timers....165
Creating a timer....165
Signal, pulse, or thread?....165
What kind of timer?....166
A server with periodic pulses....168
Timers delivering signals....177
Timers creating threads....177
Getting and setting the realtime clock and more....177
Advanced topics....180
Other clock sources....180
Kernel timeouts....182
Summary....185

Chapter 4: Interrupts....187
QNX Neutrino and interrupts....188
Interrupt service routine....189
Level-sensitivity versus edge-sensitivity....191
Writing interrupt handlers....194
Attaching an interrupt handler....194
Now that you've attached an interrupt....195
Detaching an interrupt handler....196
The *flags* parameter....196
The interrupt service routine....197
ISR functions....205
Summary....208

Chapter 5: Resource Managers....209
What is a resource manager?....210
Examples of resource managers....210
Characteristics of resource managers....210
The client's view....212
Finding the server....212
Finding the process manager....213
Handling directories....214
Unioned filesystems....215
Client summary....217
The resource manager's view....218
Registering a pathname....218
Handling messages....219
The resource manager library....220
The library really does what we just talked about....221
Behind the scenes at the library....222
Writing a resource manager....225
Data structures....225
Resource manager structure....231
POSIX-layer data structures....240
Handler routines....249
General notes....249
Connect functions notes....251
Alphabetical listing of connect and I/O functions....253
Access control list I/O function handler....253
Change file mode I/O function handler....254
Change ownership I/O function handler....256
Change timestamps I/O function handler....257
Close duplicate OCB I/O function handler....258
Close OCB I/O function handler....260
Create file link connect function handler....261
Device control I/O function handler....263
Duplicate a file descriptor I/O function handler....267

File descriptor information I/O function handler....268
File locking I/O function handler....269
Flush data I/O function handler....271
Lock OCB I/O function handler....272
Make filesystem node connect function handler....273
Map memory I/O function handler....275
Mount connect function handler....277
Open connect function handler....278
Open file descriptor I/O function handler....280
pathconf I/O function handler....281
Power I/O function handler....283
Read I/O function handler....283
Read link connect function handler....285
Rename connect function handler....287
Register for notification I/O function handler....289
Resize file I/O function handler....292
Set file position I/O function handler....293
Shutdown I/O function handler....295
stat I/O function handler....295
Unblock connect function handler....296
Unblock I/O function handler....297
Unlink connect function handler....298
Unlock OCB I/O function handler....300
User-defined message I/O function handler....301
Write I/O function handler....303
Examples....306
The basic skeleton of a resource manager....306
A simple read I/O function handler example....308
A simple write I/O function handler example....312
A simple device control I/O function handler example....316
Advanced topics....322
Extending the OCB....322
Locking in the resource manager....324
Extending the attributes structure....325
Blocking within the resource manager....326
Returning directory entries....326
Summary....338

Appendix A: QNX 4 to QNX Neutrino....339
Similarities....340
Improvements....341
Embeddability....341
Thread support....341
Message passing....342
Device driver model....343
ARM support....343
SMP support....344

Porting philosophy....345
Message passing considerations....345
Interrupt service routines....355
Summary....356

Appendix B: Calling 911....357
Read the manuals....358
Contact technical support....360
Describe the problem....360
Precise information....361
Versions....361
If you're using a beta....361
Reproduce the problem....362
Narrow it down....362
Training....363

Appendix C: Sample Programs....365
`atoz.c`....366
`time1.c`....371
`tp1.c`....376
`tt1.c`....379

Appendix D: Glossary....381

Index....389

About This Guide

Getting Started with QNX Neutrino: A Guide for Realtime Programmers is intended to introduce you to the QNX Neutrino RTOS and help you develop applications and resource managers for it.

EDITOR'S NOTE: Rob Krten wrote this book, then called *Getting Started with QNX Neutrino 2*, in 1999. In 2005, QNX Software Systems bought the rights to the book; this edition has been updated by the staff there.

The following table may help you find information quickly:

To find out about:	Go to:
Peter van der Veen's forward	*Foreword to the First Edition*
Rob Krten's preface	*Preface to the First Edition*
Using processes and threads	*Processes and Threads*
Sending, receiving, and replying to messages	*Message Passing*
Working with times and timers	*Clocks, Timers, and Getting a Kick Every So Often*
Interrupts	*Interrupts*
Writing resource managers	*Resource Managers*
Migrating from QNX 4 to QNX Neutrino	*QNX 4 to QNX Neutrino*
Getting help	*Calling 911*
Full source code for the examples	*Sample Programs*
Terms used in QNX docs	*Glossary*

Typographical conventions

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

The following table summarizes our conventions:

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

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

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

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

Notes point out something important or useful.

© 2020, QNX Software Systems Limited

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

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

Note to Windows users

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

Technical support

Technical assistance is available for all supported products.

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

© 2020, QNX Software Systems Limited

Foreword to the First Edition by Peter van der Veen

When I found myself staring at the first draft of this book I started thinking that it was going to be a difficult read because I'd spent so many years intimately involved with the design and development of QNX Neutrino. But I was wrong! I found this book easy to read and very enjoyable because of the way Rob combines the QNX philosophy ("Why things are the way they are") with good habits applicable to any realtime programming project. This book is suitable for people who've never seen Neutrino before, or those who've used it extensively.

For people who've never used Neutrino, the book gives an excellent tutorial on how to use it. Since Rob himself comes from a QNX 2 and QNX 4 background, his book is also great for people who've used a QNX operating system before, because they share a common ground.

As for myself, I was first introduced to QNX at an insurance company in the mid-1980s. This insurance company primarily used an IBM mainframe, but they wanted to shorten the time required to calculate quotes on corporate insurance. To do this they used networks of 8 MHz 80286 ATs running QNX 2. They distributed their data using QNX native networking, allowing access to all customer data files from any QNX machine. This system was well-designed using the QNX client/server philosophy and I was hooked on QNX.

When I joined QNX Software Systems at the start of 1991, QNX 4 had just been released. QNX 4 was developed to conform to the just-approved POSIX 1003.1 specification which would make it easier to port public domain UNIX code than it was with QNX 2, and it would conform to a genuine standard. In a few years we started thinking about the next-generation operating system. The current group of less than 15 developers started meeting to discuss anything we'd like to do differently and things that we'd need in the future. We wanted to support the newer POSIX specifications and make it easier to write drivers. We also didn't want to lock ourselves to the x86 processor or "fix" anything that wasn't broken. The ideas that Dan Dodge and Gordon Bell started out with when they created QNX are still in Neutrino today—ideas like message-passing, having a small, lean kernel, providing fast, realtime response, etc. Complicating the design was the goal of Neutrino being even more modular than QNX 4 (for example, we wanted to provide a fully-functional kernel that you could link against, allowing for more deeply embedded systems than QNX 4). In 1994 Dan Dodge and I started working on the updated kernel and process manager.

As those of you who've been using QNX products for a long time already know, writing device drivers for QNX 2 was a hair-raising experience. You had to be *very* careful! In fact, most developers started with the QNX Software Systems-supplied source for the spool device and carefully tweaked it to do whatever they wanted. Only a few people tried writing disk drivers, as this required specialized assembly language stubs. Because of this, almost nobody ended up writing drivers for QNX 2. In QNX 4, writing drivers was made *much* easier by making all I/O operations go through a standard, well-defined, message-passing interface. When you did an *open()*, the server received an open message. When you did a *read()*, the server received a read message. QNX 4 capitalized on the message passing theme by using it to decouple clients from servers. I remember when I first saw the beta version 3.99 (a QNX 4 pre-release version) and thinking, "Wow! This is elegant!" In fact, I was so enamored with this, that I immediately wrote a QNX 2 read-only filesystem using the new message-passing interface; it was easy now!

For Neutrino, the process manager was being designed with three main separate functions: pathname space management, process creation (attributes, destruction, etc.), and memory space management.

It also included several sub-services (**/dev/null**, **/dev/zero**, image filesystem, etc.). Each of these acted independently, but all shared the common code for processing the messages. This common code was very useful, so we decided to take all the common code and make a cover library for it. The "Resource Manager" library (or, as Rob likes to pronounce it, to my utter dismay, *rez-mugger* :-)) was born.

We also found that most resource managers wanted to provide POSIX semantics for their devices or filesystems, so we wrote another layer on top of the resource manager layer called the *iofunc*()* functions. This lets anybody write a resource manager, and have it automatically inherit POSIX functionality, without any additional work. At about this time Rob was writing the Neutrino courses, and he wanted to write a completely minimal resource manager example, **/dev/null**. His main slide was, "All you have to do is provide *read()* and *write()* message handlers, and you have a complete **/dev/null**!" I took that as a personal challenge, and removed even that requirement—the resource manager library now implements **/dev/null** in about half a dozen function calls. Since this library is shipped with Neutrino, everyone can write fully POSIX-compatible device drivers with minimal effort.

While the resource manager concept was significant in the evolution of Neutrino, and would indeed provide a solid base for the operating system, the fledgling OS needed more. Filesystems, connectivity (such as TCP/IP) and common devices (serial, console) were all being developed in parallel. After a lot of work, with lots of long hours, Neutrino 1.00 was released in early 1996. Over the next few years, more and more R&D staff were working on Neutrino. We've added SMP support, multiplatform support (x86, PowerPC and MIPS currently, with more to come), and the dispatch interface (that allows combining resource managers and other IPC methods), all covered in this book.

In August of 1999, we released QNX Neutrino 2.00; just in time for Rob's book! :-)

I think this book will be a "must have" for anyone who is writing programs for Neutrino.

Peter van der Veen (on a plane somewhere between Ottawa and San Jose, September 1999)

© 2020, QNX Software Systems Limited

Preface to the First Edition by Rob Krten

A few years after I started using computers, the very first IBM PC came out. I must have been one of the first people in Ottawa to buy this box, with 16 KB of RAM and no video card, because the salesman wasn't experienced enough to point out that the machine would be totally useless without the video card! Although the box wasn't useful, it did say “IBM” on it (at the time reserved solely for mainframes and the like), so it was impressive on its own. When I finally had enough money to buy the video card, I was able to run BASIC on my parents' TV. To me, this was the height of technology—especially with a 300-baud acoustically coupled modem! So, you can imagine my chagrin, when my friend Paul Trunley called me up and said, “Hey, log in to my computer!” I thought to myself, “Where did he get a VAX from?” since that was the only conceivable machine I knew about that would fit in his parents' house and let you “log in” to. So I called it up. It was a PC running an obscure operating system called “QUNIX,” with a revision number less than 1.00. It let me “log in.” I was hooked!

What has always struck me about the QNX family of operating systems is the small memory footprint, the efficiency, and the sheer elegance of the implementation. I would often entertain (or bore, more likely) dinner guests with stories about all the programs running concurrently on my machine in the basement, as we ate. Those who were knowledgeable about computers would speculate about how huge the disk must be, how I must have near infinite memory, etc. After dinner, I'd drag them downstairs and show them a simple PC with (at the time) 8 MB of RAM and a 70 MB hard disk. This would sometimes impress them. Those who where not impressed would then be shown how much RAM and disk space was still *available*, and how most of the used disk space was just data I had accumulated over the years.

As time passed, I've had the privilege of working at a number of companies, most of which were involved with some form of QNX development; (from telecoms, to process control, to frame grabber drivers, ...), with the single most striking characteristic being the simplicity of the designs and implementation. In my opinion, this is due to the key engineers on the projects having a good understanding of the QNX operating system—if you have a clean, elegant architecture to base your designs on, chances are that your designs will also end up being clean and elegant (unless the problem is *really* ugly).

In November, 1995, I had the good fortune to work directly for QNX Software Systems, writing the training material for their two QNX Neutrino courses, and presenting them over the next three years.

It's these past 19 years or so that gave me the inspiration and courage to write the first book, *Getting Started with QNX 4: A Guide for Realtime Programmers,* which was published in May, 1998. With this new book on QNX Neutrino, I hope to share some of the concepts and ideas I've learned, so that you can gain a good, solid understanding of how the QNX Neutrino OS works, and how you can use it to your advantage. Hopefully, as you read the book, light bulbs will turn on in your head, making you say “Aha! That's why they did it this way!”

A little history

QNX Software Systems, the company that created the QNX operating system, was founded in 1980 by Dan Dodge and Gordon Bell (both graduates of the University of Waterloo in Ontario, Canada). Initially, the company was called Quantum Software Systems Limited, and the product was called "QUNIX" ("Quantum UNIX"). After a polite letter from AT&T's lawyers (who owned the "UNIX" trademark at the time), the product's name changed to "QNX." Some time after that, the company's name itself changed to "QNX Software Systems"—in those days, everyone and their dog seemed to have a company called "Quantum" something or other.

The first commercially successful product was simply called "QNX" and ran on 8088 processors. Then, "QNX 2" (QNX version 2) came out in the early 1980s. It's still running in many mission-critical systems to this day. Around 1991, a new operating system, "QNX 4," was introduced, with enhanced 32-bit operations and POSIX support. In 1995, QNX Neutrino was introduced.

© 2020, QNX Software Systems Limited

Who this book is for

This book is suitable for anyone wishing to gain a good fundamental understanding of the key features of the QNX Neutrino RTOS and how it works. Readers with a modest computer background should still get a lot out of the book (although the discussion in each chapter gets more and more technical as the chapter progresses). Even diehard hackers should find some interesting twists, especially with the two fundamental features of QNX Neutrino, the message-passing nature of the operating system and the way device drivers are structured.

I've tried to explain things in an easy-to-read “conversational” style, anticipating some of the common questions that come up and answering them with examples and diagrams. Because a complete understanding of the C language isn't required, but is definitely an asset, there are quite a few code samples sprinkled throughout.

What's in this book?

This book introduces you to what the QNX Neutrino RTOS is and how it functions. It contains chapters covering process states, threads, scheduling policies, message passing, operating system modularity, and so on. If you've never used QNX Neutrino before, but are familiar with realtime operating systems, then you'll want to pay particular attention to the chapters on message passing and resource managers, since these are concepts fundamental to QNX Neutrino.

Processes and Threads

An introduction to processes and threads in QNX Neutrino, realtime, scheduling, and prioritization. You'll learn about scheduling states and QNX Neutrino's scheduling policies, as well as the functions you use to control scheduling, create processes and threads, and modify processes and threads that are already running. You'll see how QNX Neutrino implements SMP (Symmetrical Multiprocessing), and the advantages (and pitfalls) that this brings.

"Scheduling and the real world" discusses how threads are scheduled on a running system, and what sorts of things can cause a running thread to be rescheduled.

Message Passing

An introduction to QNX Neutrino's most fundamental feature, message passing. You'll learn what message passing is, how to use it to communicate between threads, and how to pass messages over a network. Priority inversion, the bane of realtime systems everywhere, and other advanced topics are also covered here.

This is one of the most important chapters in this book!

Clocks, Timers, and Getting a Kick Every So Often

Learn all about the system clock and timers, and how to get a timer to send you a message. Lots of practical information here, and code samples galore.

Interrupts

This chapter will teach you how to write interrupt handlers for QNX Neutrino, and how interrupt handlers affect thread scheduling.

Resource Managers

Learn all about QNX Neutrino resource managers (also known variously as "device drivers" and "I/O managers"). You'll need to read and understand the *Message Passing* chapter before you write your own resource managers. The source for several complete resource managers is included.

Resource managers are another important aspect of every QNX Neutrino-based system.

© 2020, QNX Software Systems Limited

QNX 4 to QNX Neutrino

This is an invaluable guide for anyone porting their QNX 4 application to QNX Neutrino, or having to maintain code on both platforms. (QNX 4 is QNX Software Systems' previous-generation operating system, also the subject of my previous book, *Getting Started with QNX 4*.) Even if you're designing a new application, there may be demand from your customer base to support it on *both* QNX 4 *and* QNX Neutrino or later—if that happens, this section will help you avoid common pitfalls and show you how to write code that's portable to both operating systems.

Calling 911

Where you can turn to when you get stuck, find a bug, or need help with your design.

Glossary

Contains definitions of the terms used throughout this book.

Other references

In addition to the custom kernel interface, QNX Neutrino implements a wide range of industry standards. This lets you support your favorite publishers when looking for information about standard functions from ANSI, POSIX, TCP/IP, etc.

About Rob Krten

Rob Krten has been doing embedded systems work, mostly on contract, since 1986 and systems-level programming since 1981.

During his three-year contract at QNX Software Systems, he designed and presented courses on "Realtime Programming under the Neutrino Kernel" and "Writing a Resource Manager." He also wrote the prototype version of the QNX Native Networking Manager (Qnet) software, as well as a significant portion of the *Building Embedded Systems* book.

Both this book and his previous book, *Getting Started with QNX 4: A Guide for Realtime Programmers,* have received a Society for Technical Communications (STC; **http://www.stc.org/**) Award of Merit.

© 2020, QNX Software Systems Limited

Acknowledgments

This book would not have been possible without the help and support I received from the following people, who contributed numerous suggestions and comments: Dave Athersych, Luc Bazinet, James Chang, Dan Dodge, Dave Donohoe, Steven Dufresne, Thomas Fletcher, David Gibbs, Marie Godfrey, Bob Hubbard, Mike Hunter, Pradeep Kathail, Steve Marsh, Danny N. Prairie, and Andrew Vernon. (Apologies in advance if I've missed anyone.)

I'd like to particularly thank Brian Stecher, who patiently reviewed at least three complete drafts of this book in detail, and Peter van der Veen, who spent many nights at my place (granted, I *did* bribe him with beer and pizza), giving me insight into the detailed operations of QNX Neutrino's resource managers.

Thanks to Kim Fraser for once again providing the cover artwork.

Additionally, my thanks goes out to John Ostrander for his excellent grammatical suggestions and detailed proof-reading of of the book `:-)`

And of course, a special thank-you goes to my editor, Chris Herborth, for finding the time to edit this book, help out with the sometimes obscure SGML/LaTeX tools, etc., all while doing dozens of other things at the same time! *[I told you to remind me not to do that again! – chrish]*

I'd also like to gratefully acknowledge the patience and understanding of my wife, Christine, for putting up with me while I crawled off into the basement and ignored her for hours on end!

Chapter 1
Processes and Threads

A thread is a flow of execution, and a process is a collection of threads. In the QNX Neutrino RTOS, the schedulable entity is a thread, not a process.

Process and thread fundamentals

Before we start talking about threads, processes, time slices, and all the other wonderful “scheduling concepts,” let's establish an analogy.

What I want to do first is illustrate *how* threads and processes work. The best way I can think of (short of digging into the design of a realtime system) is to imagine our threads and processes in some kind of situation.

A process as a house

Let's base our analogy for processes and threads using a regular, everyday object: a house.

A house is really a container, with certain attributes (such as the amount of floor space, the number of bedrooms, and so on).

If you look at it that way, the house really doesn't actively *do* anything on its own—it's a passive object. This is effectively what a process is. We'll explore this shortly.

The occupants as threads

The people living in the house are the “active” objects—they're the ones using the various rooms, watching TV, cooking, taking showers, and so on. We'll soon see that's how threads behave.

Single threaded

If you've ever lived on your own, then you know what this is like—you *know* that you can do *anything* you want in the house at *any* time, because there's nobody else in the house. If you want to turn on the stereo, use the washroom, have dinner—whatever—you just go ahead and do it.

Multi threaded

Things change dramatically when you add another person into the house. Let's say you get married, so now you have a spouse living there too. You can't just march into the washroom at any given point; you need to check first to make sure your spouse isn't in there!

If you have two responsible adults living in a house, generally you can be reasonably lax about “security”—you know that the other adult will respect your space, won't try to set the kitchen on fire (deliberately!), and so on.

Now, throw a few kids into the mix and suddenly things get a lot more interesting.

Back to processes and threads

Just as a house occupies an area of real estate, a process occupies memory. And just as a house's occupants are free to go into any room they want, a processes' threads all have common access to that memory. If a thread allocates something (mom goes out and buys a game), all the other threads immediately have access to it (because it's present in the common address space—it's in the house). Likewise, if the process allocates memory, this new memory is available to all the threads as well. The trick here is to recognize whether the memory *should* be available to all the threads in the process. If

© 2020, QNX Software Systems Limited

it is, then you'll need to have all the threads synchronize their access to it. If it isn't, then we'll assume that it's specific to a particular thread. In that case, since only *that* thread has access to it, we can assume that no synchronization is required—the thread isn't going to trip itself up!

As we know from everyday life, things aren't quite that simple. Now that we've seen the basic characteristics (summary: everything is shared), let's take a look at where things get a little more interesting, and why.

The diagram below shows the way that we'll be representing threads and processes. The process is the circle, representing the “container” concept (the address space), and the three squiggly lines are the threads. You'll see diagrams like this throughout the book.

Figure 1: A process as a container of threads.

Mutual exclusion

If you want to take a shower, and there's someone already using the bathroom, you'll have to wait. How does a thread handle this?

It's done with something called *mutual exclusion*. It means pretty much what you think—a number of threads are mutually exclusive when it comes to a particular resource.

If you're taking a shower, you want to have *exclusive access* to the bathroom. To do this, you would typically go into the bathroom and lock the door from the inside. Anyone else trying to use the bathroom would get stopped by the lock. When you're done, you'd unlock the door, allowing someone else access.

This is just what a thread does. A thread uses an object called a *mutex* (an acronym for *MUT*ual *EX*clusion). This object is like the lock on a door—once a thread has the mutex locked, no other thread can get the mutex, until the owning thread releases (unlocks) it. Just like the door lock, threads waiting to obtain the mutex will be barred.

Another interesting parallel that occurs with mutexes and door locks is that the mutex is really an “advisory” lock. If a thread doesn't obey the *convention* of using the mutex, then the protection is useless. In our house analogy, this would be like someone breaking into the washroom through one of the walls ignoring the convention of the door and lock.

Priorities

What if the bathroom is currently locked and a number of people are waiting to use it? Obviously, all the people are sitting around outside, waiting for whoever is in the bathroom to get out. The real question is, What happens when the door unlocks? Who gets to go next?

You'd figure that it would be “fair” to allow whoever is waiting the longest to go next. Or it might be “fair” to let whoever is the oldest go next. Or tallest. Or most important. There are any number of ways to determine what's “fair.”

We solve this with threads via two factors: priority and length of wait.

Suppose two people show up at the (locked) bathroom door at the same time. One of them has a pressing deadline (they're already late for a meeting) whereas the other doesn't. Wouldn't it make sense to allow the person with the pressing deadline to go next? Well, of course it would. The only question is how you decide who's more "important." This can be done by assigning a *priority* (let's just use a number like QNX Neutrino does—one is the lowest usable priority, and 255 is the highest as of this version). The people in the house that have pressing deadlines would be given a *higher priority*, and those that don't would be given a *lower priority*.

Same thing with threads. A thread inherits its scheduling policy from its parent thread, but can (if it has the authority to do so) call *pthread_setschedparam()* to change its scheduling policy and priority, or *pthread_setschedprio()* to change just its priority.

If a number of threads are waiting, and the mutex becomes unlocked, we would give the mutex to the waiting thread with the highest priority. Suppose, however, that both people have the *same* priority. Now what do you do? Well, in that case, it would be "fair" to allow the person who's been waiting the longest to go next. This is not only "fair," but it's also what the microkernel does. In the case of a bunch of threads waiting, we go *primarily* by priority, and *secondarily* by length of wait.

The mutex is certainly not the only *synchronization* object that we'll encounter. Let's look at some others.

Semaphores

Let's move from the bathroom into the kitchen, since that's a socially acceptable location to have more than one person at the same time. In the kitchen, you may not want to have *everyone* in there at once. In fact, you probably want to limit the number of people you can have in the kitchen (too many cooks, and all that).

Let's say you don't ever want to have more than two people in there simultaneously. Could you do it with a mutex? Not as we've defined it. Why not? This is actually a very interesting problem for our analogy. Let's break it down into a few steps.

A semaphore with a count of 1

The bathroom can have one of two situations, with two states that go hand-in-hand with each other:

- the door is unlocked and nobody is in the room
- the door is locked and one person is in the room

No other combination is possible—the door can't be locked with nobody in the room (how would we unlock it?), and the door can't be unlocked with someone in the room (how would they ensure their privacy?). This is an example of a semaphore with a count of one—there can be at most only one person in that room, or one thread using the semaphore.

The key here (pardon the pun) is the way we characterize the lock. In your typical bathroom lock, you can lock and unlock it only from the inside—there's no outside-accessible key. Effectively, this means that ownership of the mutex is an *atomic* operation—there's *no chance* that while you're in the process of getting the mutex some other thread will get it, with the result that you *both* own the mutex. In our house analogy this is less apparent, because humans are just so much smarter than ones and zeros.

What we need for the kitchen is a different type of lock.

© 2020, QNX Software Systems Limited

A semaphore with a count greater than 1

Suppose we installed the traditional key-based lock in the kitchen. The way this lock works is that if you have a key, you can unlock the door and go in. Anyone who uses this lock agrees that when they get inside, they will immediately lock the door from the inside so that anyone on the outside will always require a key.

Well, now it becomes a simple matter to control how many people we want in the kitchen—hang two keys outside the door! The kitchen is always locked. When someone wants to go into the kitchen, they see if there's a key hanging outside the door. If so, they take it with them, unlock the kitchen door, go inside, and use the key to lock the door.

Since the person going into the kitchen must have the key with them when they're in the kitchen, we're directly controlling the number of people allowed into the kitchen at any given point by limiting the number of keys available on the hook outside the door.

With threads, this is accomplished via a *semaphore*. A “plain” semaphore works just like a mutex—you either own the mutex, in which case you have access to the resource, or you don't, in which case you don't have access. The semaphore we just described with the kitchen is a *counting semaphore*—it keeps track of the count (by the number of keys available to the threads).

A semaphore as a mutex

We just asked the question “Could you do it with a mutex?” in relation to implementing a lock with a count, and the answer was no. How about the other way around? Could we use a semaphore as a mutex?

Yes. In fact, in some operating systems, that's exactly what they do—they don't have mutexes, only semaphores! So why bother with mutexes at all?

To answer that question, look at *your* washroom. How did the builder of your house implement the “mutex”? I suspect you don't have a key hanging on the wall!

Mutexes are a “special purpose” semaphore. If you want one thread running in a particular section of code, a mutex is by far the most efficient implementation.

Later on, we'll look at other synchronization schemes—things called condvars, barriers, and sleepons.

Just so there's no confusion, realize that a mutex has other properties, such as priority inheritance, that differentiate it from a semaphore.

The kernel's role

The house analogy is excellent for getting across the concept of synchronization, but it falls down in one major area. In our house, we had many threads running *simultaneously*. However in a single-processor system, only one "thing" can run at once.

Single CPU

Let's look at what happens in the real world, and specifically, the "economy" case where we have one CPU in the system. In this case, since there's only one CPU present, only one thread can run at any given point in time. The kernel decides (using a number of rules, which we'll see shortly) which thread to run, and runs it.

Multiple CPU (SMP)

If you buy a system that has multiple, identical CPUs all sharing memory and devices, you have an *SMP box* (SMP stands for *S*ymmetrical *M*ulti *P*rocessor, with the "symmetrical" part indicating that all the CPUs in the system are identical). In this case, the number of threads that can run concurrently (simultaneously) is limited by the number of CPUs. (In reality, this was the case with the single-processor box too!) Since each processor can execute only one thread at a time, with multiple processors, multiple threads can execute simultaneously.

Let's ignore the number of CPUs present for now—a useful abstraction is to design the system as if multiple threads really were running simultaneously, even if that's not the case. A little later on, in the "*Things to watch out for when using SMP*" section, we'll see some of the non-intuitive impacts of SMP.

The kernel as arbiter

So who decides which thread is going to run at any given instant in time? That's the kernel's job.

The kernel determines which thread should be using the CPU at a particular moment, and *switches context* to that thread. Let's examine what the kernel does with the CPU.

The CPU has a number of registers (the exact number depends on the processor family, e.g., x86 versus ARM, and the specific family member, e.g., Pentium). When the thread is running, information is stored in those registers (e.g., the current program location).

When the kernel decides that another thread should run, it needs to:

1. save the currently running thread's registers and other context information
2. load the new thread's registers and context into the CPU

But how does the kernel decide that another thread should run? It looks at whether or not a particular thread is capable of using the CPU at this point. When we talked about mutexes, for example, we introduced a blocking state (this occurred when one thread owned the mutex, and another thread wanted to acquire it as well; the second thread would be blocked).

From the kernel's perspective, therefore, we have one thread that *can* consume CPU, and one that *can't*, because it's blocked, waiting for a mutex. In this case, the kernel lets the thread that can run

© 2020, QNX Software Systems Limited

consume CPU, and puts the other thread into an internal list (so that the kernel can track its request for the mutex).

Obviously, that's not a very interesting situation. Suppose that a number of threads *can* use the CPU. Remember that we delegated access to the mutex based on priority and length of wait? The kernel uses a similar scheme to determine which thread is going to run next. There are two factors: priority and scheduling policy, evaluated in that order.

Prioritization

Consider two threads capable of using the CPU. If these threads have different priorities, then the answer is really quite simple; the kernel gives the CPU to the highest priority thread. QNX Neutrino's priorities go from one (the lowest usable) and up, as we mentioned when we talked about obtaining mutexes. Note that priority zero is reserved for the idle thread—you can't use it. (If you want to know the minimum and maximum values for your system, use the functions *sched_get_priority_min()* and *sched_get_priority_max()*—they're prototyped in **<sched.h>**. In this book, we'll assume one as the lowest usable, and 255 as the highest.)

If another thread with a higher priority suddenly becomes able to use the CPU, the kernel will immediately context-switch to the higher priority thread. We call this *preemption*; the higher-priority thread preempted the lower-priority thread. When the higher-priority thread is done, and the kernel context-switches back to the lower-priority thread that was running before, we call this *resumption*; the kernel resumes running the previous thread.

Now, suppose that two threads are capable of using the CPU and have the exact same priority.

Scheduling policies

Let's assume that one of the threads is currently using the CPU. We'll examine the rules that the kernel uses to decide when to context-switch in this case. (Of course, this entire discussion really applies only to threads at the same priority—the *instant* that a higher-priority thread is ready to use the CPU it gets it; that's the whole point of having priorities in a realtime operating system.)

The two main *scheduling policies* (policies) that the microkernel understands are Round Robin (or just “RR”) and FIFO (First-In, First-Out). (There's also sporadic scheduling, but it's beyond the scope of this book; see “Sporadic scheduling” in the QNX Neutrino Microkernel chapter of the *System Architecture* guide.)

FIFO

In the FIFO scheduling policy, a thread is allowed to consume CPU for as long as it wants. This means that if that thread is doing a very long mathematical calculation, and no other thread of a higher priority is ready, that thread could potentially run *forever*. What about threads of the same priority? They're locked out as well. (It should be obvious at this point that threads of a lower priority are locked out too.)

If the running thread quits or voluntarily gives up the CPU, *then* the kernel looks for other threads at the same priority that are capable of using the CPU. If there are no such threads, *then* the kernel looks for lower-priority threads capable of using the CPU. Note that the term “voluntarily gives up the CPU” can mean one of two things. If the thread goes to sleep, or blocks on a semaphore, etc., then yes, a lower-priority thread could run (as described above). But there's also a “special” call, *sched_yield()*

(based on the kernel call *SchedYield()*), which gives up CPU *only* to another thread of the same priority—a lower-priority thread would never be given a chance to run if a higher-priority was ready to run. If a thread does in fact call *sched_yield()*, and no other thread at the same priority is ready to run, the original thread continues running. Effectively, *sched_yield()* is used to give another thread of the *same priority* a crack at the CPU.

In the diagram below, we see three threads operating in two different processes:

Figure 2: Three threads in two different processes.

If we assume that threads “A” and “B” are READY, and that thread “C” is blocked (perhaps waiting for a mutex), and that thread “D” (not shown) is currently executing, then this is what a portion of the READY queue that the microkernel maintains will look like:

Figure 3: Two threads on the READY queue, one blocked, one running.

This shows the kernel's internal READY queue, which the kernel uses to decide what to schedule next. Note that thread “C” is not on the READY queue, because it's blocked, and thread “D” isn't on the READY queue either because it's running.

Round Robin

The RR scheduling policy is identical to FIFO, *except* that the thread will not run forever *if* there's another thread at the same priority. It runs only for a system-defined *timeslice* whose value you can determine by using the function *sched_rr_get_interval()*. The timeslice is usually 4 ms, but it's actually 4 times the *ticksize*, which you can query or set with *ClockPeriod()*.

What happens is that the kernel starts an RR thread, and notes the time. If the RR thread is running for a while, the time allotted to it will be up (the timeslice will have expired). The kernel looks to see if there's another thread at the same priority that's ready. If there is, the kernel runs it. If not, then the kernel will continue running the RR thread (i.e., the kernel grants the thread another timeslice).

© 2020, QNX Software Systems Limited

The rules

Let's summarize the scheduling rules (for a single CPU), in order of importance:

- Only one thread can run at a time.
- The highest-priority ready thread will run.
- A thread will run until it blocks or exits.
- An RR thread will run for its timeslice, and then the kernel will reschedule it (if required).

The following flowchart shows the decisions that the kernel makes:

Figure 4: Scheduling roadmap.

For a multiple-CPU system, the rules are the same, except that multiple CPUs can run multiple threads concurrently. The order in which the threads run (i.e., which threads get to run on the multiple CPUs) is determined in the exact same way as with a single CPU: the highest-priority READY thread will run on a CPU. For lower-priority or longer-waiting threads, the kernel has some flexibility as to when to schedule them to avoid inefficiency in the use of the cache. For more information about SMP, see the Multicore Processing chapter of the QNX Neutrino *Programmer's Guide*.

Kernel states

We've been talking about “running,” “ready,” and “blocked” loosely—let's now formalize these thread states.

RUNNING

QNX Neutrino's RUNNING state simply means that the thread is now actively consuming the CPU. On an SMP system, there will be multiple threads running; on a single-processor system, there will be one thread running.

READY

The READY state means that this thread *could* run right now—except that it's not, because another thread, (at the same or higher priority), is running. If two threads were capable of using the CPU, one thread at priority 10 and one thread at priority 7, the priority 10 thread would be RUNNING, and the priority 7 thread would be READY.

The blocked states

What do we call the blocked state? The problem is, there's not just *one* blocked state. Under QNX Neutrino, there are in fact over a dozen blocking states.

Why so many? Because the kernel keeps track of *why* a thread is blocked.

We saw two blocking states already—when a thread is blocked waiting for a mutex, the thread is in the MUTEX state. When a thread is blocked waiting for a semaphore, it's in the SEM state. These states simply indicate which queue (and which resource) the thread is blocked on.

If a number of threads are blocked on a mutex (in the MUTEX blocked state), they get no attention from the kernel *until* the thread that owns the mutex releases it. At that point one of the blocked threads is made READY, and the kernel makes a rescheduling decision (if required).

Why “if required?” The thread that just released the mutex could very well still have other things to do *and* have a higher priority than that of the waiting threads. In this case, we go to the second rule, which states, “The highest-priority ready thread will run,” meaning that the scheduling order has not changed—the higher-priority thread continues to run.

Kernel states, the complete list

Here's the complete list of kernel blocking states, with brief explanations of each state.

By the way, this list is available in **<sys/states.h>**—you'll notice that the states are all prefixed with `STATE_` , but the prefix tends to be omitted in conversation and the documentation (for example, “READY” is really STATE_READY):

If the state is:	The thread is:
STATE_CONDVAR	Waiting for a condition variable to be signaled
STATE_DEAD	Dead. Kernel is waiting to release the thread's resources

© 2020, QNX Software Systems Limited

If the state is:	The thread is:
STATE_INTR	Waiting for an interrupt
STATE_JOIN	Waiting for the completion of another thread
STATE_MUTEX	Waiting to acquire a mutex
STATE_NANOSLEEP	Sleeping for a period of time
STATE_NET_REPLY	Waiting for a reply to be delivered across the network
STATE_NET_SEND	Waiting for a pulse or message to be delivered across the network
STATE_READY	Not running on a CPU, but is ready to run (one or more higher or equal priority threads are running)
STATE_RECEIVE	Waiting for a client to send a message
STATE_REPLY	Waiting for a server to reply to a message
STATE_RUNNING	Actively running on a CPU
STATE_SEM	Waiting to acquire a semaphore
STATE_SEND	Waiting for a server to receive a message
STATE_SIGSUSPEND	Waiting for a signal
STATE_SIGWAITINFO	Waiting for a signal
STATE_STACK	Waiting for more stack to be allocated
STATE_STOPPED	Suspended (SIGSTOP signal)
STATE_WAITCTX	Waiting for a register context (usually floating point) to become available (only on SMP systems)
STATE_WAITPAGE	Waiting for process manager to resolve a fault on a page
STATE_WAITTHREAD	Waiting for a thread to be created

The important thing to keep in mind is that when a thread is blocked, regardless of *which* state it's blocked in, it consumes *no* CPU. Conversely, the only state in which a thread consumes CPU is in the RUNNING state.

We'll see the SEND, RECEIVE, and REPLY blocked states in the *Message Passing* chapter. The NANOSLEEP state is used with functions like *sleep()*, which we'll look at in the chapter on *Clocks, Timers, and Getting a Kick Every So Often*. The INTR state is used with *InterruptWait()*, which we'll take a look at in the *Interrupts* chapter. Most of the other states are discussed in this chapter.

Threads and processes

Let's return to our discussion of threads and processes, this time from the perspective of a real live system. Then, we'll take a look at the function calls used to deal with threads and processes.

We know that a process can have one or more threads. (A process that had zero threads wouldn't be able to *do* anything—there'd be nobody home, so to speak, to actually perform any useful work.) A QNX Neutrino system can have one or more processes. (The same discussion applies—a QNX Neutrino system with zero processes wouldn't do anything.)

So what do these processes and threads do? Ultimately, they form a *system*—a collection of threads and processes that performs some goal.

At the highest level, the system consists of a number of processes. Each process is responsible for providing a service of some nature—whether it's a filesystem, a display driver, data acquisition module, control module, or whatever.

Within each process, there may be a number of threads. The number of threads varies. One designer using only one thread may accomplish the same functionality as another designer using five threads. Some problems lend themselves to being multithreaded, and are in fact relatively simple to solve, while other processes lend themselves to being single-threaded, and are difficult to make multithreaded.

The topic of designing with threads could easily occupy another book—we'll just stick with the basics here.

Why processes?

So why not just have one process with a zillion threads?

While some OSs force you to code that way, the advantages of breaking things up into multiple processes are many:

- decoupling and modularity
- maintainability
- reliability

The ability to “break the problem apart” into several independent problems is a powerful concept. It's also at the heart of QNX Neutrino. A QNX Neutrino system consists of many independent modules, each with a certain responsibility. These independent modules are distinct processes. The people at QNX Software Systems used this trick to develop the modules in isolation, without the modules relying on each other. The only “reliance” the modules would have on each other is through a small number of well-defined interfaces.

This naturally leads to enhanced maintainability, thanks to the lack of interdependencies. Since each module has its own particular definition, it's reasonably easy to fix one module—especially since it's not tied to any other module.

Reliability, though, is perhaps the most important point. A process, just like a house, has some well-defined “borders.” A person in a house has a pretty good idea when they're in the house, and when they're not. A thread has a *very* good idea—if it's accessing memory within the process, it can

© 2020, QNX Software Systems Limited

live. If it steps out of the bounds of the process's address space, it gets killed. This means that two threads, running in different processes, are effectively isolated from each other.

Figure 5: Memory protection.

The *process address space* is maintained and enforced by QNX Neutrino's process manager module. When a process is started, the process manager allocates some memory to it and starts a thread running. The memory is marked as being owned by that process.

This means that if there are multiple threads in that process, and the kernel needs to context-switch between them, it's a very efficient operation—we don't have to change the address space, just which thread is running. If, however, we have to change to another thread in another process, then the process manager gets involved and causes an address space switch as well. Don't worry—while there's a bit more overhead in this additional step, under QNX Neutrino this is still very fast.

Starting a process

Let's now turn our attention to the function calls available to deal with threads and processes. Any thread can start a process; the only restrictions imposed are those that stem from basic security (file access, privilege restrictions, etc.). In all probability, you've already started other processes; either from the system startup script, the shell, or by having a program start another program on your behalf.

Starting a process from the command line

For example, from the shell you can type:

```
$ program1
```

This instructs the shell to start a program called `program1` and to wait for it to finish. Or, you could type:

```
$ program2 &
```

This instructs the shell to start `program2` *without* waiting for it to finish. We say that `program2` is running “in the background.”

If you want to adjust the priority of a program before you start it, you could use the `nice` command, just like in UNIX:

```
$ nice program3
```

This instructs the shell to start `program3` at a reduced priority.

Or does it?

If you look at what really happens, we told the shell to run a program called `nice` at the regular priority. The `nice` command adjusted its *own* priority to be lower (this is where the name "nice" comes from), and then *it* ran `program3` at that lower priority.

Starting a process from within a program

You don't usually care about the fact that the shell creates processes—this is a basic assumption about the shell. In some application designs, you'll certainly be relying on *shell scripts* (batches of commands in a file) to do the work for you, but in other cases you'll want to create the processes yourself.

For example, in a large multiprocess system, you may want to have one master program start all the other processes for your application based on some kind of configuration file. Another example would include starting up processes when certain operating conditions (events) have been detected.

Let's take a look at the functions that QNX Neutrino provides for starting up other processes (or transforming into a different program):

- *system()*
- *exec()* family of functions
- *posix_spawn()* family of functions
- *spawn()* family of functions
- *fork()*

Which function you use depends on two requirements: portability and functionality. As usual, there's a trade-off between the two.

The common thing that happens in all the calls that create a new process is the following. A thread in the original process calls one of the above functions. Eventually, the function gets the process manager to create an address space for a new process. Then, the kernel starts a thread in the new process. This thread executes a few instructions, and calls *main()*. (In the case of *fork()*, of course, the new thread begins execution in the new process by returning from the *fork()*; we'll see how to deal with this shortly.)

Starting a process with the *system()* call

The *system()* function is the simplest; it takes a command line, the same as you'd type it at a shell prompt, and executes it.

In fact, *system()* actually starts up a shell to handle the command that you want to perform.

The editor that I'm using to write this book makes use of the *system()* call. When I'm editing, I may need to "shell out," check out some samples, and then come back into the editor, all without losing my place. In this editor, I may issue the command `:!pwd` for example, to display the current working directory. The editor runs this code for the `:!pwd` command:

```
system ("pwd");
```

Is *system()* suited for everything under the sun? Of course not, but it's useful for a lot of your process-creation requirements.

© 2020, QNX Software Systems Limited

Starting a process with the *exec()* and *spawn()* calls

Let's look at some of the other process-creation functions.

The next process-creation functions we should look at are the *exec()* and *spawn()* families. Before we go into the details, let's see what the differences are between these two groups of functions.

The *exec()* family *transforms* the current process into another one. What I mean by that is that when a process issues an *exec()* function call, that process ceases to run the current program and begins to run another program. The process ID doesn't change—that process *changed* into another program. What happened to all the threads in the process? We'll come back to that when we look at *fork()*.

The *spawn()* family, on the other hand, doesn't do that. Calling a member of the *spawn()* family creates *another* process (with a new process ID) that corresponds to the program specified in the function's arguments.

Let's look at the different variants of the *spawn()* and *exec()* functions. In the table that follows, you'll see which ones are POSIX and which aren't. Of course, for maximum portability, you'll want to use only the POSIX functions. (The *spawn()* and *spawnp()* functions were in a POSIX draft, but never made it into the standard. The POSIX versions are *posix_spawn()* and *posix_spawnp()*.)

Spawn	POSIX?	Exec	POSIX?
spawn()	No		
spawnl()	No	*execl()*	Yes
spawnle()	No	*execle()*	Yes
spawnlp()	No	*execlp()*	Yes
spawnlpe()	No	*execlpe()*	No
spawnp()	No		
spawnv()	No	*execv()*	Yes
spawnve()	No	*execve()*	Yes
spawnvp()	No	*execvp()*	Yes
spawnvpe()	No	*execvpe()*	No

While these variants might appear to be overwhelming, there *is* a pattern to their suffixes:

A suffix of:	Means:
`l` (lowercase “L”)	The argument list is specified via a list of parameters given in the call itself, terminated by a NULL argument.
`e`	An environment is specified.

A suffix of:	Means:
`p`	The *PATH* environment variable is used in case the full pathname to the program isn't specified.
`v`	The argument list is specified via a pointer to an argument vector.

The argument list is a list of command-line arguments passed to the program.

Also, note that in the C library, *spawnlp()*, *spawnvp()*, and *spawnlpe()* all call *spawnvpe()*, which in turn calls *spawnp()*. The functions *spawnle()*, *spawnv()*, and *spawnl()* all eventually call *spawnve()*, which then calls *spawn()*. Finally, *spawnp()* calls *spawn()*. So, the root of all spawning functionality is the *spawn()* call.

Figure 6: How the *spawn*()* functions are related.

Let's now take a look at the various *spawn()* and *exec()* variants in detail so that you can get a feel for the various suffixes used. Then, we'll see the *spawn()* call itself.

"l" suffix

For example, if I want to invoke the `ls` command with the arguments `-t`, `-r`, and `-l` (meaning "sort the output by time, in reverse order, and show me the long version of the output"), I could specify it as either:

```
/* To run ls and keep going: */
spawnl (P_WAIT, "/bin/ls", "/bin/ls", "-t", "-r", "-l", NULL);

/* To transform into ls: */
execl ("/bin/ls", "/bin/ls", "-t", "-r", "-l", NULL);
```


© 2020, QNX Software Systems Limited

or, using the `v` suffix variant:

```
char *argv [] =
{
    "/bin/ls",
    "-t",
    "-r",
    "-l",
    NULL
};

/* To run ls and keep going: */
spawnv (P_WAIT, "/bin/ls", argv);

/* To transform into ls: */
execv ("/bin/ls", argv);
```

Why the choice? It's provided as a convenience. You may have a parser already built into your program, and it would be convenient to pass around arrays of strings. In that case, I'd recommend using the "`v`" suffix variants. Or, you may be coding up a call to a program where you know what the parameters are. In that case, why bother setting up an array of strings when you *know* exactly what the arguments are? Just pass them to the "`l`" suffix variant.

Note that we passed the actual pathname of the program (**/bin/ls**) *and* the name of the program *again* as the first argument. We passed the name twice to support programs that behave differently based on how they're invoked.

For example, the GNU compression and decompression utilities (`gzip` and `gunzip`) are actually links to the same executable. When the executable starts, it looks at `argv [0]` (passed to *main()*) and decides whether it should compress or decompress.

"e" suffix

The "e" suffix versions pass an *environment* to the program. An environment is just that—a kind of "context" for the program to operate in. For example, you may have a spelling checker that has a dictionary of words. Instead of specifying the dictionary's location every time on the command line, you could provide it in the environment:

```
$ export DICTIONARY=/home/rk/.dict

$ spellcheck document.1
```

The `export` command tells the shell to create a new environment variable (in this case, *DICTIONARY*), and assign it a value (**/home/rk/.dict**).

If you ever wanted to use a different dictionary, you'd have to alter the environment before running the program. This is easy from the shell:

```
$ export DICTIONARY=/home/rk/.altdict

$ spellcheck document.1
```

But how can you do this from your own programs? To use the "e" versions of *spawn()* and *exec()*, you specify an array of strings representing the environment:

```
char *env [] =
{
    "DICTIONARY=/home/rk/.altdict",
    NULL
};

// To start the spell-checker:
spawnle (P_WAIT, "/usr/bin/spellcheck", "/usr/bin/spellcheck",
         "document.1", NULL, env);

// To transform into the spell-checker:
execle ("/usr/bin/spellcheck", "/usr/bin/spellcheck",
        "document.1", NULL, env);
```

"p" suffix

The "p" suffix versions will search the directories in your *PATH* environment variable to find the executable. You've probably noticed that all the examples have a hard-coded location for the executable: **/bin/ls** and **/usr/bin/spellcheck**. What about other executables? Unless you want to first find out the *exact* path for that particular program, it would be best to have the user tell your program all the places to search for executables. The standard *PATH* environment variable does just that. Here's the one from a minimal system:

```
PATH=/proc/boot:/bin
```

This tells the shell that when I type a command, it should first look in the directory **/proc/boot**, and if it can't find the command there, it should look in the binaries directory **/bin** part. *PATH* is a colon-separated list of places to look for commands. You can add as many elements to the *PATH* as you want, but keep in mind that all pathname components will be searched (in order) for the executable.

If you don't know the path to the executable, then you can use the "p" variants. For example:

```
// Using an explicit path:
execl ("/bin/ls", "/bin/ls", "-l", "-t", "-r", NULL);

// Search your PATH for the executable:
execlp ("ls", "ls", "-l", "-t", "-r", NULL);
```

If *execl()* can't find `ls` in **/bin**, it returns an error. The *execlp()* function will search all the directories specified in the *PATH* for `ls`, and will return an error only if it can't find `ls` in any of those directories.

© 2020, QNX Software Systems Limited

This is also great for multiplatform support—your program doesn't have to be coded to know about the different CPU names, it just finds the executable.

What if you do something like this?

```
execlp ("/bin/ls", "ls", "-l", "-t", "-r", NULL);
```

Does it search the environment? No. You told *execlp()* to use an *explicit* pathname, which overrides the normal *PATH* searching rule. If it doesn't find `ls` in **/bin** that's it, no other attempts are made (this is identical to the way *execl()* works in this case).

Is it dangerous to mix an explicit path with a plain command name (e.g., the path argument **/bin/ls**, and the command name argument **ls**, instead of **/bin/ls**)? This is usually pretty safe, because:

- a large number of programs ignore `argv[0]` anyway
- those that do care usually call *basename()*, which strips off the directory portion of `argv[0]` and returns just the name.

The only compelling reason for specifying the full pathname for the first argument is that the program can display diagnostics including this first argument, which can instantly tell you where the program was invoked from. This may be important when the program can be found in multiple locations along the *PATH*.

The *spawn()* functions all have an extra parameter; in all the above examples, I've always specified P_WAIT. There are four flags you can pass to *spawn()* to change its behavior:

P_WAIT

The calling process (your program) is blocked until the newly created program has run to completion and exited.

P_NOWAIT

The calling program *doesn't* block while the newly created program runs. This allows you to start a program in the background, and continue running while the other program does its thing.

P_NOWAITO

Identical to P_NOWAIT, except that the SPAWN_NOZOMBIE flag is set, meaning that you don't have to worry about doing a *waitpid()* to clear the process's exit code.

P_OVERLAY

This flag turns the *spawn()* call into the corresponding *exec()* call! Your program transforms into the specified program, with no change in process ID.

It's generally clearer to use the *exec()* call if that's what you meant—it saves the maintainer of the software from having to look up P_OVERLAY in the *C Library Reference*!

Plain *spawn()*

As we mentioned above, all *spawn()* functions eventually call the plain *spawn()* function. Here's the prototype for the *spawn()* function:

```
#include <spawn.h>

pid_t
spawn (const char *path,
       int fd_count,
       const int fd_map [],
       const struct inheritance *inherit,
       char * const argv [],
       char * const envp []);
```

We can immediately dispense with the *path*, *argv*, and *envp* parameters—we've already seen those above as representing the location of the executable (the *path* member), the argument vector (*argv*), and the environment (*envp*).

The *fd_count* and *fd_map* parameters go together. If you specify zero for *fd_count*, then *fd_map* is ignored, and it means that all file descriptors (except those modified by *fcntl()*'s FD_CLOEXEC flag) will be inherited in the newly created process. If the *fd_count* is nonzero, then it indicates the number of file descriptors contained in *fd_map*; only the specified ones will be inherited.

The *inherit* parameter is a pointer to a structure that contains a set of flags, signal masks, and so on. For more details, you should consult the QNX Neutrino *C Library Reference*.

Starting a process with the *fork()* call

Suppose you want to create a new process that's identical to the currently running process and have it run concurrently. You could approach this with a *spawn()* (and the P_NOWAIT parameter), giving the newly created process enough information about the exact state of your process so it could set itself up. However, this can be extremely complicated; describing the “current state” of the process can involve lots of data.

There is an easier way—the *fork()* function, which duplicates the current process. All the code is the same, and the data is the same as the creating (or *parent*) process's data.

Of course, it's impossible to create a process that's *identical in every way* to the parent process. Why? The most obvious difference between these two processes is going to be the process ID—we can't create two processes with the same process ID. If you look at *fork()*'s documentation in the QNX Neutrino *C Library Reference*, you'll see that there is a list of differences between the two processes. You should read this list to be sure that you know these differences if you plan to use *fork()*.

If both sides of a *fork()* look alike, how do you tell them apart? When you call *fork()*, you create another process executing the same code at the same location (i.e., both are about to return from the *fork()* call) as the parent process. Let's look at some sample code:

```
int main (int argc, char **argv)
{
    int retval;
```


© 2020, QNX Software Systems Limited

```
    printf ("This is most definitely the parent process\n");
    fflush (stdout);
    retval = fork ();
    printf ("Which process printed this?\n");

    return (EXIT_SUCCESS);
}
```

After the *fork()* call, *both* processes are going to execute the second *printf()* call! If you run this program, it prints something like this:

```
This is most definitely the parent process
Which process printed this?
Which process printed this?
```

Both processes print the second line.

The only way to tell the two processes apart is the *fork()* return value in *retval*. In the newly created *child* process, *retval* is zero; in the parent process, *retval* is the child's process ID.

Clear as mud? Here's another code snippet to clarify:

```
printf ("The parent is pid %d\n", getpid ());
fflush (stdout);

if (child_pid = fork ()) {
    printf ("This is the parent, child pid is %d\n",
            child_pid);
} else {
    printf ("This is the child, pid is %d\n",
            getpid ());
}
```

This program will print something like:

```
The parent is pid 4496
This is the parent, child pid is 8197
This is the child, pid is 8197
```

You can tell which process you are (the parent or the child) after the *fork()* by looking at *fork()*'s return value.

Using *fork()* in a multithreaded process

Using *fork()* in a multithreaded process can be tricky, especially if one thread calls *fork()* while other threads are in critical sections of code.

POSIX says that following a *fork()*, the child's memory state is a copy of the parent's, but the child contains only a single thread; any other threads in the parent aren't duplicated in the child. If one of those parent threads was in the middle of manipulating a data structure when the *fork()* happened,

that data structure will be in an undefined, possibly inconsistent, state in the child following the *fork()*. Regular mutexes can be used to protect different threads from interfering with each other, but protecting against *fork()* requires the forking code to be aware of the internals of your data structure synchronization.

For example, suppose the parent process has two threads, T1 and T2, and T2 has locked a mutex. If T1 calls *fork()*, the child process is a copy of the parent, including any mutexes, but with only one thread. If the thread in the child tries to lock the mutex, the operation fails because T2 has it locked—and there is no T2 in the child process, so no one can unlock the mutex. This can lead to a deadlock.

The simplest solution is to call *fork()* only from a single-threaded process, or to call it before you create additional threads. But, if you insist on using *fork()* in a multithreaded process, there are some things you can do:

- Use at-fork handlers
- Use forksafe mutexes
- Call *exec*()* soon after forking. POSIX requires that the child process use only functions that are async-signal-safe until it calls one of the *exec*()* functions.

You can use *pthread_atfork()* to register the following handlers:

- *prepare* — called before forking
- *parent* — called in the parent after forking
- *child* — called in the child after forking

You can call *pthread_atfork()* more than once if you want to register additional handlers; the *parent* and *child* handlers are called in the order that you registered them, and the *prepare* handlers are called in the reverse order.

In the *child* handler, you can call only functions that are async-signal-safe (i.e., functions that you can safely call from a signal handler).

In the *prepare* handler, you can lock all the mutexes, and then unlock them in the *parent* and *child* handlers. This ensures that no other threads can hold the mutexes across the *fork()*. However, if you have many different mutexes in different code that is only loosely coupled, getting the locking order correct to avoid deadlock can be tricky.

In QNX Neutrino 6.6 and later, we've implemented a *forksafe mutex* as another way to deal with this problem. A `forksafe_mutex_t` is a replacement for a `pthread_mutex` but is mutually exclusive with *fork()* calls. The *fork()* function is aware of mutexes of this type and can't execute while any forksafe mutexes are held: *fork()* blocks while any forksafe mutexes are held, and while a *fork()* call is underway, any attempts to lock a forksafe mutex block until the *fork()* has completed.

You can use the following functions to work with forksafe mutexes:

- *forksafe_mutex_destroy()*
- *forksafe_mutex_init()*
- *forksafe_mutex_lock()*
- *forksafe_mutex_trylock()*

© 2020, QNX Software Systems Limited

- *forksafe_mutex_unlock()*

They replace the corresponding *pthread_mutex_*()* functions; for more information, see the *C Library Reference*.

Note the following:

- There's a performance cost to using forksafe mutexes.
- Forksafe mutexes have priority over *fork()* calls. This means that if a forksafe mutex is held by one thread, and a second thread is blocked trying to *fork()*, and then a third thread tries to lock a forksafe mutex, the third thread succeeds—it doesn't wait until the *fork()* call has been given a chance to run. This means that if you have a lot of different threads grabbing and releasing forksafe mutexes, an attempt to call *fork()* may be blocked indefinitely; it isn't allowed to proceed until *all* forksafe mutexes are released.
- While forksafe mutexes support priority inheritance between users of the mutex, there's no priority inheritance between the users of the mutex and a caller of *fork()*. A high-priority thread attempting to call *fork()* can be blocked by a low-priority thread that's holding a forksafe mutex.
- You mustn't hold a `forksafe_mutex_t` when you invoke *fork()*, or an obvious deadlock will occur. The same is true of a `pthread_mutex_t` protected by an at-fork handler.

At-fork handlers and forksafe mutexes will help, but it can still be tricky to correctly set up the mutexes in the child, so our recommendation is not to use *fork()* in a multithreaded process, or to use *posix_spawn()* instead.

Starting a process with the *posix_spawn()* call

As we mentioned earlier, *spawn()* and *spawnp()* were in a POSIX draft, but never made it into the standard. Instead, POSIX added *posix_spawn()* and *posix_spawnp()*.

The *posix_spawn()* and *posix_spawnp()* functions were designed to let you specify how and what the child process should run. They were also designed to be easily extended; their arguments include a pointer to a `posix_spawnattr_t` structure that specifies *spawn attributes*, and a pointer to a `posix_spawn_file_actions_t` structure that specifies *file actions* (e.g., which file descriptors to close or duplicate in the child process). These are both opaque structures, so there are functions that you can use to initialize, get, and set values within them.

For the attributes—some of which are POSIX, and some of which are QNX Neutrino extensions—the first step is to set the members of the attribute structure to their default values by calling *posix_spawnattr_init()*. To set an attribute, call the appropriate *posix_spawnattr_*()* function *and* set the corresponding flag. To set the POSIX flags, call *posix_spawnattr_setflags()*; to set the QNX Neutrino flags, call *posix_spawnattr_setxflags()*. For example:

- To set the child's process group (a POSIX attribute), include POSIX_SPAWN_SETPGROUP in the call to *posix_spawnattr_setflags()*, and call *posix_spawnattr_setpgroup()* to set the process group.
- To set the child's process runmask (a QNX Neutrino attribute that controls which processors the child can run on), include POSIX_SPAWN_EXPLICIT_CPU in the call to *posix_spawnattr_setxflags()*, and call *posix_spawnattr_setrunmask()* to set the runmask.

Setting the file actions is simpler because there are no flags to set; you just call *posix_spawn_file_actions_init()*, and then call the appropriate *posix_spawn_file_actions_*()* functions.

When you're ready to create the new process, call *posix_spawn()* or *posix_spawnp()*. The difference between these functions is as follows:

- for *posix_spawn()* you specify the absolute path to the executable
- for *posix_spawnp()* you specify the name of the executable file, and the function looks for it in the directories listed in your *PATH* environment variable

For more information, see the entry for these functions in the *C Library Reference*.

So what should you use?

Obviously, if you're porting existing code, you'll want to use whatever the existing code uses. For new code, you should avoid *fork()* if at all possible. Here's why:

- Although *fork()* works with multiple threads, you need to register a *pthread_atfork()* handler and lock every single mutex *before* you fork, complicating the design.
- The child of *fork()* duplicates *all* open file descriptors. As we'll see in the *Resource Manager* chapter later, this causes a lot of work—most of which will be unnecessary if the child then immediately does an *exec()* and closes all the file descriptors anyway.

Your best choice for portability is *posix_spawn()*.

Starting a thread

Now that we've seen how to start another process, let's see how to start another thread.

Any thread can create another thread in the same process; there are no restrictions (short of memory space, of course!). The most common way of doing this is via the POSIX *pthread_create()* call:

```
#include <pthread.h>

int
pthread_create (pthread_t *thread,
                const pthread_attr_t *attr,
                void *(*start_routine) (void *),
                void *arg);
```

The *pthread_create()* function takes four arguments:

thread

A pointer to a `pthread_t` where the thread ID is stored.

attr

An *attributes structure*.

start_routine

The routine where the thread begins.

arg

An argument passed to the thread's *start_routine*.

© 2020, QNX Software Systems Limited

Note that the *thread* pointer and the attributes structure (*attr*) are optional—you can pass them as NULL.

The *thread* parameter can be used to store the thread ID of the newly created thread. You'll notice that in the examples below, we'll pass a NULL, meaning that we don't care what the ID is of the newly created thread. If we did care, we could do something like this:

```
pthread_t tid;

pthread_create (&tid, …
printf ("Newly created thread id is %d\n", tid);
```

This use is actually quite typical, because you'll often want to know which thread ID is running which piece of code.

A small subtle point. It's possible that the newly created thread may be running *before* the thread ID (the *tid* parameter) is filled. This means that you should be careful about using the *tid* as a global variable. The usage shown above is okay, because the *pthread_create()* call has returned, which means that the *tid* value is stuffed correctly.

The new thread begins executing at *start_routine()*, with the argument *arg*.

The thread attributes structure

When you start a new thread, it can assume some well-defined defaults, or you can explicitly specify its characteristics.

Before we jump into a discussion of the thread attribute functions, let's look at the `pthread_attr_t` data type:

```
typedef struct {
    int                 __flags;
    size_t              __stacksize;
    void                *__stackaddr;
    void                (*__exitfunc)(void *status);
    int                 __policy;
    struct sched_param  __param;
    unsigned            __guardsize;
} pthread_attr_t;
```

Basically, the fields are used as follows:

__flags

Non-numerical (Boolean) characteristics (e.g., whether the thread should run “detached” or “joinable”).

__stacksize, ***__stackaddr***, and ***__guardsize***

Stack specifications.

__exitfunc

Function to execute at thread exit.

__policy **and** ***__param***

Scheduling parameters.

The following functions are available:

Attribute administration

- *pthread_attr_destroy()*
- *pthread_attr_init()*

Flags (Boolean characteristics)

- *pthread_attr_getdetachstate()*
- *pthread_attr_setdetachstate()*
- *pthread_attr_getinheritsched()*
- *pthread_attr_setinheritsched()*
- *pthread_attr_getscope()*
- *pthread_attr_setscope()*
- *pthread_attr_getsuspendstate_np()*
- *pthread_attr_setsuspendstate_np()*

Stack related

- *pthread_attr_getguardsize()*
- *pthread_attr_setguardsize()*
- *pthread_attr_getstack()*
- *pthread_attr_setstack()*
- *pthread_attr_getstackaddr()*
- *pthread_attr_setstackaddr()*
- *pthread_attr_getstacklazy()*
- *pthread_attr_setstacklazy()*
- *pthread_attr_getstackprealloc()*
- *pthread_attr_setstackprealloc()*
- *pthread_attr_getstacksize()*
- *pthread_attr_setstacksize()*

Scheduling related

- *pthread_attr_getschedparam()*
- *pthread_attr_setschedparam()*
- *pthread_attr_getschedpolicy()*
- *pthread_attr_setschedpolicy()*

© 2020, QNX Software Systems Limited

This looks like a pretty big list, but in reality we have to worry about only half of them, because they're paired: “get” and “set” (with the exception of *pthread_attr_init()* and *pthread_attr_destroy()*).

Before we examine the attribute functions, there's one thing to note. You must call *pthread_attr_init()* to initialize the attribute structure before using it, set it with the appropriate *pthread_attr_set*()* function(s), and *then* call *pthread_create()* to create the thread. Changing the attribute structure *after* the thread's been created has no effect.

Thread attribute administration

The function *pthread_attr_init()* must be called to initialize the attribute structure before using it:

```
...
pthread_attr_t  attr;
...
pthread_attr_init (&attr);
```

You *could* call *pthread_attr_destroy()* to “uninitialize” the thread attribute structure, but almost no one ever does (unless you have POSIX-compliant code).

In the descriptions that follow, I've marked the default values with “*(default)*.”

The “flags” thread attribute

Let's start with the boolean thread attributes.

- To create a “joinable” thread (meaning that another thread can synchronize to its termination via *pthread_join()*), you'd use *pthread_attr_setdetachstate()* like this:

  ```
  (default)
  pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_JOINABLE);
  ```

  To create one that can't be joined (called a “detached” thread), you'd use this:

  ```
  pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_DETACHED);
  ```

- If you want the thread to inherit the scheduling attributes of the creating thread (that is, to have the same scheduling policy and the same priority), you'd use *pthread_attr_setinheritsched()* like this:

  ```
  (default)
  pthread_attr_setinheritsched (&attr, PTHREAD_INHERIT_SCHED);
  ```

  To create one that uses the scheduling attributes specified in the attribute structure itself (which you'd set using *pthread_attr_setschedparam()* and *pthread_attr_setschedpolicy()*), you'd use this:

  ```
  pthread_attr_setinheritsched (&attr, PTHREAD_EXPLICIT_SCHED);
  ```

- In QNX Neutrino 7.0.1 or later, if you don't want the thread to be put into a suspended state when it's created, you'd call *pthread_attr_setsuspendstate_np()* like this:

  ```
  (default)
  pthread_attr_setsuspendstate_np (&attr, PTHREAD_NOT_SUSPENDED);
  ```

  If you want the thread to be suspended when it's created, you'd use this:

  ```
  pthread_attr_setsuspendstate_np (&attr, PTHREAD_SUSPENDED);
  ```

- Finally, you'd never call *pthread_attr_setscope()*. Why? Because QNX Neutrino supports only “system” scope, and it's the default when you initialize the attribute. (“System” scope means that all threads in the system compete against each other for CPU; the other value, “process,” means

that threads compete against each other for CPU within the process, and the kernel schedules the processes.)

If you do insist on calling it, you can call it only as follows:

```
(default)
pthread_attr_setscope (&attr, PTHREAD_SCOPE_SYSTEM);
```

The “stack” thread attributes

The thread attribute stack parameters are prototyped as follows:

```
int
pthread_attr_setstack (pthread_attr_t *attr, void *addr, size_t ssize );

int
pthread_attr_setstackaddr (pthread_attr_t *attr, void *addr);

int
pthread_attr_setguardsize (pthread_attr_t *attr, size_t gsize);

int
pthread_attr_setstacklazy (pthread_attr_t *attr, int lazystack);

int
pthread_attr_setstackprealloc (pthread_attr_t * attr, size_t psize);

int
pthread_attr_setstacksize (pthread_attr_t *attr, size_t ssize);
```

These functions all take the attribute structure as their first parameter; the other parameters are selected from the following:

addr

The address of the stack, if you're providing one.

gsize

The size of the “guard” area.

lazystack

Indicates if the stack should be allocated on demand or up front from physical memory.

psize

The the amount of memory to preallocate for a thread's MAP_LAZY stack.

ssize

The size of the stack.

© 2020, QNX Software Systems Limited

The guard area is a memory area immediately after the stack that the thread can't write to. If it does (meaning that the stack was about to overflow), the thread will get hit with a SIGSEGV. If the guardsize is 0, it means that there's no guard area. This also implies that there's no stack overflow checking. If the guardsize is nonzero, then it's set to at least the system-wide default guardsize (which you can obtain with a call to *sysconf()* with the constant _SC_PAGESIZE). Note that the guardsize will be at least as big as a "page" (for example, 4 KB on an x86 processor). Also, note that the guard page doesn't take up any *physical* memory—it's done as a virtual address (MMU) "trick."

The *addr* is the address of the stack, in case you're providing it. You can set it to NULL meaning that the system will allocate (and will free!) the stack for the thread. The advantage of specifying a stack is that you can do postmortem stack depth analysis. This is accomplished by allocating a stack area, filling it with a "signature" (for example, the string "STACK" repeated over and over), and letting the thread run. When the thread has completed, you'd look at the stack area and see how far the thread had scribbled over your signature, giving you the maximum depth of the stack used during this particular run.

The *ssize* parameter specifies how big the stack is. If you provide the stack in *addr*, then *ssize* should be the size of that data area. If you don't provide the stack in *addr* (meaning you passed a NULL), then the *ssize* parameter tells the system how big a stack it should allocate for you. If you specify a 0 for *ssize*, the system will select the default stack size for you. Obviously, it's bad practice to specify a 0 for *ssize and* specify a stack using *addr*—effectively you're saying "Here's a pointer to an object, and the object is some default size." The problem is that there's no binding between the object size and the passed value.

If a stack is being provided via *addr*, no automatic stack overflow protection exists for that thread (i.e., there's no guard area). However, you can certainly set this up yourself using *mmap()* and *mprotect()*.

Finally, the *lazystack* parameter indicates if the physical memory should be allocated as required (use the value PTHREAD_STACK_LAZY) or all up front (use the value PTHREAD_STACK_NOTLAZY). The advantage of allocating the stack "on demand" (as required) is that the thread won't use up more physical memory than it absolutely has to. The disadvantage (and hence the advantage of the "all up front" method) is that in a low-memory environment the thread won't mysteriously die some time during operating when it needs that extra bit of stack, and there isn't any memory left. If you are using PTHREAD_STACK_NOTLAZY, you'll most likely want to set the actual size of the stack instead of accepting the default, because the default is quite large.

The "scheduling" thread attributes

Finally, if you do specify PTHREAD_EXPLICIT_SCHED for *pthread_attr_setinheritsched()*, then you'll need a way to specify both the scheduling policy and the priority of the thread you're about to create.

This is done with the two functions:

```
int
pthread_attr_setschedparam (pthread_attr_t *attr,
                            const struct sched_param *param);

int
pthread_attr_setschedpolicy (pthread_attr_t *attr,
                             int policy);
```

The *policy* is simple; it's one of SCHED_FIFO, SCHED_RR, or SCHED_OTHER.

SCHED_OTHER is currently mapped to SCHED_RR.

The *param* is a structure that contains one member of relevance here: *sched_priority*. Set this value via direct assignment to the desired priority.

A common bug to watch out for is specifying PTHREAD_EXPLICIT_SCHED and then setting only the scheduling policy. The problem is that in an initialized attribute structure, the value of *param.sched_priority* is 0. This is the same priority as the IDLE process, meaning that your newly created thread will be competing for CPU with the IDLE process.

Been there, done that, got the T-shirt. `:-)`

Enough people have been bitten by this that QNX Software Systems has made priority zero reserved for only the idle thread. You simply cannot run a thread at priority zero.

A few examples

Let's take a look at some examples. We'll assume that the proper include files (**<pthread.h>** and **<sched.h>**) have been included, and that the thread to be created is called *new_thread()* and is correctly prototyped and defined.

The most common way of creating a thread is to simply let the values default:

```
pthread_create (NULL, NULL, new_thread, NULL);
```

In the above example, we've created our new thread with the defaults, and passed it a NULL as its one and only parameter (that's the third NULL in the *pthread_create()* call above).

Generally, you can pass anything you want (via the *arg* field) to your new thread. Here we're passing the number 123:

```
pthread_create (NULL, NULL, new_thread, (void *) 123);
```

A more complicated example is to create a non-joinable thread with round-robin scheduling at priority 15:

```
pthread_attr_t attr;

// initialize the attribute structure
pthread_attr_init (&attr);

// set the detach state to "detached"
pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_DETACHED);

// override the default of INHERIT_SCHED
pthread_attr_setinheritsched (&attr, PTHREAD_EXPLICIT_SCHED);
pthread_attr_setschedpolicy (&attr, SCHED_RR);
attr.param.sched_priority = 15;
```


© 2020, QNX Software Systems Limited

```
// finally, create the thread
pthread_create (NULL, &attr, new_thread, NULL);
```

To see what a multithreaded program "looks like," you could run the `pidin` command from the shell. Say our program was called `spud`. If we run `pidin` once before `spud` created a thread and once after `spud` created two more threads (for three total), here's what the output would look like (I've shortened the `pidin` output to show only `spud`):

```
# pidin
pid    tid name               prio STATE       Blocked
 12301   1 spud                 10r READY

# pidin
pid    tid name               prio STATE       Blocked
 12301   1 spud                 10r READY
 12301   2 spud                 10r READY
 12301   3 spud                 10r READY
```

As you can see, the process `spud` (process ID 12301) has three threads (under the "tid" column). The three threads are running at priority 10 with a scheduling algorithm of round robin (indicated by the "r" after the 10). All three threads are READY, meaning that they're able to use CPU but aren't currently running on the CPU (another, higher-priority thread, is currently running).

Now that we know all about creating threads, let's take a look at how and where we'd use them.

Where a thread is a good idea

There are two classes of problems where the application of threads is a good idea.

Threads are like overloading operators in C++; it may seem like a good idea (at the time) to overload every single operator with some interesting use, but it makes the code hard to understand. Similarly with threads, you could create piles of threads, but the additional complexity will make your code hard to understand, and therefore hard to maintain. Judicious use of threads, on the other hand, will result in code that is functionally very clean.

Threads are great where you can *parallelize operations*—a number of mathematical problems spring to mind (graphics, digital signal processing, etc.). Threads are also great where you want a program to perform several *independent functions* while sharing data, such as a web-server that's serving multiple clients simultaneously. We'll examine these two classes.

Threads in mathematical operations

Suppose that we have a graphics program that performs ray tracing. Each raster line on the screen is dependent on the main database (which describes the actual picture being generated). The key here is this: *each raster line is independent of the others*. This immediately causes the problem to stand out as a threadable program.

Here's the single-threaded version:

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

```
    int x1;

    ...    // perform initializations

    for (x1 = 0; x1 < num_x_lines; x1++) {
        do_one_line (x1);
    }

    ...    // display results
}
```

Here we see that the program will iterate *x1* over all the raster lines that are to be calculated.

On an SMP system, this program will use only one of the CPUs. Why? Because we haven't told the operating system to do anything in parallel. The operating system isn't smart enough to look at the program and say, "Hey, hold on a second! We have 4 CPUs, and it looks like there are independent execution flows here. I'll run it on all 4 CPUs!"

So, it's up to the system designer (you) to tell QNX Neutrino which parts can be run in parallel. The easiest way to do that would be:

```
int
main (int argc, char **argv)
{
    int x1;

    ...    // perform initializations

    for (x1 = 0; x1 < num_x_lines; x1++) {
        pthread_create (NULL, NULL, do_one_line, (void *) x1);
    }

    ...    // display results
}
```

There are a number of problems with this simplistic approach. First of all (and this is most minor), the *do_one_line()* function would have to be modified to take a `void *` instead of an `int` as its argument. This is easily remedied with a typecast.

The second problem is a little bit trickier. Let's say that the screen resolution that you were computing the picture for was 1280 by 1024. We'd be creating 1280 threads! This is not a problem for QNX Neutrino—the OS "limits" you to 32767 threads per process! However, each thread *must* have a unique stack. If your stack is a reasonable size (say 8 KB), you'll have used 1280 × 8 KB (10 megabytes!) of stack. And for what? There are only 4 processors in your SMP system. This means that only 4 of the 1280 threads will run at a time—the other 1276 threads are waiting for a CPU. (In reality, the stack will "fault in," meaning that the space for it will be allocated only as required. Nonetheless, it's a waste—there are still other overheads.)

© 2020, QNX Software Systems Limited

A much better solution to this would be to break the problem up into 4 pieces (one for each CPU), and start a thread for each piece:

```
int num_lines_per_cpu;
int num_cpus;

int
main (int argc, char **argv)
{
    int cpu;

    ...    // perform initializations

    // get the number of CPUs
    num_cpus = _syspage_ptr -> num_cpu;
    num_lines_per_cpu = num_x_lines / num_cpus;
    for (cpu = 0; cpu < num_cpus; cpu++) {
        pthread_create (NULL, NULL,
                        do_one_batch, (void *) cpu);
    }

    ...    // display results
}

void *
do_one_batch (void *c)
{
    int cpu = (int) c;
    int x1;

    for (x1 = 0; x1 < num_lines_per_cpu; x1++) {
        do_line_line (x1 + cpu * num_lines_per_cpu);
    }
}
```

Here we're starting only *num_cpus* threads. Each thread will run on one CPU. And since we have only a small number of threads, we're not wasting memory with unnecessary stacks. Notice how we got the number of CPUs by dereferencing the "System Page" global variable *_syspage_ptr*. (For more information about what's in the system page, see the System Page chapter of *Building Embedded Systems*, or the **<sys/syspage.h>** include file).

Coding for SMP or single processor

The best part about this code is that it will function just fine on a single-processor system—you'll create only one thread, and have it do all the work. The additional overhead (one stack) is well worth the flexibility of having the software "just work faster" on an SMP box.

Synchronizing to the termination of a thread

I mentioned that there were a number of problems with the simplistic code sample initially shown. Another problem with it is that *main()* starts up a bunch of threads and then displays the results. How does the function *know* when it's safe to display the results?

To have the *main()* function poll for completion would defeat the purpose of a realtime operating system:

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

    // start threads as before

    while (num_lines_completed < num_x_lines) {
        sleep (1);
    }
}
```

Don't even consider writing code like this!

There are two elegant solutions to this problem: *pthread_join()* and *pthread_barrier_wait()*.

Joining

The simplest method of synchronization is to *join* the threads as they terminate. Joining really means waiting for termination.

Joining is accomplished by one thread waiting for the termination of another thread. The waiting thread calls *pthread_join()*:

```
#include <pthread.h>

int
pthread_join (pthread_t thread, void **value_ptr);
```

To use *pthread_join()*, you pass it the thread ID of the thread that you wish to join, and an optional *value_ptr*, which can be used to store the termination return value from the joined thread. (You can pass in a NULL if you aren't interested in this value—we're not, in this case.)

© 2020, QNX Software Systems Limited

Where did the thread ID came from? We ignored it in the *pthread_create()*—we passed in a NULL for the first parameter. Let's now correct our code:

```
int num_lines_per_cpu, num_cpus;

int main (int argc, char **argv)
{
    int cpu;
    pthread_t *thread_ids;

    ...    // perform initializations
    thread_ids = malloc (sizeof (pthread_t) * num_cpus);

    num_lines_per_cpu = num_x_lines / num_cpus;
    for (cpu = 0; cpu < num_cpus; cpu++) {
        pthread_create (&thread_ids [cpu], NULL,
                        do_one_batch, (void *) cpu);
    }

    // synchronize to termination of all threads
    for (cpu = 0; cpu < num_cpus; cpu++) {
        pthread_join (thread_ids [cpu], NULL);
    }

    ...    // display results
}
```

You'll notice that this time we passed the first argument to *pthread_create()* as a pointer to a `pthread_t`. This is where the thread ID of the newly created thread gets stored. After the first `for` loop finishes, we have *num_cpus* threads running, plus the thread that's running *main()*. We're not too concerned about the *main()* thread consuming all our CPU; it's going to spend its time waiting.

The waiting is accomplished by doing a *pthread_join()* to each of our threads in turn. First, we wait for *thread_ids [0]* to finish. When it completes, the *pthread_join()* will unblock. The next iteration of the `for` loop will cause us to wait for *thread_ids [1]* to finish, and so on, for all *num_cpus* threads.

A common question that arises at this point is, “What if the threads finish in the reverse order?” In other words, what if there are 4 CPUs, and, for whatever reason, the thread running on the last CPU (CPU 3) finishes first, and then the thread running on CPU 2 finishes next, and so on? Well, the beauty of this scheme is that nothing bad happens.

The first thing that's going to happen is that the *pthread_join()* will block on *thread_ids [0]*. Meanwhile, *thread_ids [3]* finishes. This has absolutely no impact on the *main()* thread, which is still waiting for the first thread to finish. Then *thread_ids [2]* finishes. Still no impact. And so on, until *finally thread_ids [0]* finishes, at which point, the *pthread_join()* unblocks, and we immediately proceed to the next iteration of the `for` loop. The second iteration of the `for` loop executes a *pthread_join()* on *thread_ids [1]*, which will not block—it returns immediately. Why? Because the thread identified by *thread_ids [1]* is *already finished*. Therefore, our `for` loop will “whip” through the other threads, and then exit.

At that point, we know that we've synched up with all the computational threads, so we can now display the results.

Using a barrier

When we talked about the synchronization of the *main()* function to the completion of the worker threads (in "Synchronizing to the termination of a thread," above), we mentioned two methods: *pthread_join()*, which we've looked at, and a barrier.

Returning to our house analogy, suppose that the family wanted to take a trip somewhere. The driver gets in the minivan and starts the engine. And waits. The driver waits until *all* the family members have boarded, and only then does the van leave to go on the trip—we can't leave anyone behind!

This is exactly what happened with the graphics example. The main thread needs to wait until all the worker threads have completed, and only then can the next part of the program begin.

Note an important distinction, however. With *pthread_join()*, we're waiting for the *termination* of the threads. This means that the threads are no longer with us; they've exited.

With the *barrier*, we're waiting for a certain number of threads to rendezvous at the barrier. Then, when the requisite number are present, we unblock *all of them*. (Note that the threads continue to run.)

You first create a barrier with *pthread_barrier_init()*:

```
#include <pthread.h>

int
pthread_barrier_init (pthread_barrier_t *barrier,
                      const pthread_barrierattr_t *attr,
                      unsigned int count);
```

This creates a barrier object at the passed address (pointer to the barrier object is in *barrier*), with the attributes as specified by *attr* (we'll just use NULL to get the defaults). The number of threads that must call *pthread_barrier_wait()* is passed in *count*.

Once the barrier is created, we then want each of the threads to call *pthread_barrier_wait()* to indicate that it has completed:

```
#include <pthread.h>

int
pthread_barrier_wait (pthread_barrier_t *barrier);
```

When a thread calls *pthread_barrier_wait()*, it will block until the number of threads specified initially in the *pthread_barrier_init()* have called *pthread_barrier_wait()* (and blocked too). When the correct number of threads have called *pthread_barrier_wait()*, all those threads will "simultaneously" unblock.

Here's an example:

```
/*
 *  barrier1.c
*/

#include <stdio.h>
```


© 2020, QNX Software Systems Limited

```
#include <time.h>
#include <pthread.h>
#include <sys/neutrino.h>

pthread_barrier_t   barrier; // the barrier synchronization object

void *
thread1 (void *not_used)
{
    time_t  now;
    char    buf [27];

    time (&now);
    printf ("thread1 starting at %s", ctime_r (&now, buf));

    // do the computation
    // let's just do a sleep here...
    sleep (20);
    pthread_barrier_wait (&barrier);
    // after this point, all three threads have completed.
    time (&now);
    printf ("barrier in thread1() done at %s", ctime_r (&now, buf));
}

void *
thread2 (void *not_used)
{
    time_t  now;
    char    buf [27];

    time (&now);
    printf ("thread2 starting at %s", ctime_r (&now, buf));

    // do the computation
    // let's just do a sleep here...
    sleep (40);
    pthread_barrier_wait (&barrier);
    // after this point, all three threads have completed.
    time (&now);
    printf ("barrier in thread2() done at %s", ctime_r (&now, buf));
}

main () // ignore arguments
{
    time_t  now;
    char    buf [27];

    // create a barrier object with a count of 3
    pthread_barrier_init (&barrier, NULL, 3);

    // start up two threads, thread1 and thread2
    pthread_create (NULL, NULL, thread1, NULL);
    pthread_create (NULL, NULL, thread2, NULL);
```

```
    // at this point, thread1 and thread2 are running

    // now wait for completion
    time (&now);
    printf ("main () waiting for barrier at %s", ctime_r (&now, buf));
    pthread_barrier_wait (&barrier);

    // after this point, all three threads have completed.
    time (&now);
    printf ("barrier in main () done at %s", ctime_r (&now, buf));
}
```

The main thread created the barrier object and initialized it with a count of how many threads (*including itself*!) should be synchronized to the barrier before it "breaks through." In our sample, this was a count of three: one for the *main()* thread, one for *thread1()*, and one for *thread2()*. Then the graphics computational threads (*thread1()* and *thread2()* in our case here) are started, as before. For illustration, instead of showing source for graphics computations, we just stuck in a `sleep (20);` and `sleep (40);` to cause a delay, as if computations were occurring. To synchronize, the main thread simply blocks itself on the barrier, knowing that the barrier will unblock only after the worker threads have joined it as well.

As mentioned earlier, with the *pthread_join()*, the worker threads are done and dead in order for the main thread to synchronize with them. But with the barrier, the threads are alive and well. In fact, they've just unblocked from the *pthread_barrier_wait()* when all have completed. The wrinkle introduced here is that you should be prepared to do something with these threads! In our graphics example, there's nothing for them to do (as we've written it). In real life, you may wish to start the next frame calculations.

Multiple threads on a single CPU

Suppose that we modify our example slightly so that we can illustrate why it's also sometimes a good idea to have multiple threads even on a single-CPU system.

In this modified example, one node on a network is responsible for calculating the raster lines (same as the graphics example, above). However, when a line is computed, its data should be sent over the network to another node, which will perform the display functions. Here's our modified *main()* (from the original example, without threads):

```
int
main (int argc, char **argv)
{
    int x1;

    …    // perform initializations

    for (x1 = 0; x1 < num_x_lines; x1++) {
        do_one_line (x1);           // "C" in our diagram, below
        tx_one_line_wait_ack (x1);  // "X" and "W" in diagram below
    }
}
```

You'll notice that we've eliminated the display portion and instead added a *tx_one_line_wait_ack()* function. Let's further suppose that we're dealing with a reasonably slow network, but that the CPU

© 2020, QNX Software Systems Limited

doesn't really get involved in the transmission aspects—it fires the data off to some hardware that then worries about transmitting it. The *tx_one_line_wait_ack()* uses a bit of CPU to get the data to the hardware, but then uses *no CPU* while it's waiting for the acknowledgment from the far end.

Here's a diagram showing the CPU usage (we've used “C” for the graphics compute part, “X” for the transmit part, and “W” for waiting for the acknowledgment from the far end):

Figure 7: Serialized, single CPU.

Wait a minute! We're wasting precious seconds waiting for the hardware to do its thing!

If we made this multithreaded, we should be able to get much better use of our CPU, right?

Figure 8: Multithreaded, single CPU.

This is much better, because now, even though the second thread spends a bit of its time waiting, we've reduced the total overall time required to compute.

If our times were $T_{compute}$ to compute, T_{tx} to transmit, and T_{wait} to let the hardware do its thing, in the first case our total running time would be:

$(T_{compute} + T_{tx} + T_{wait}) \times num_x_lines$

whereas with the two threads it would be

$(T_{compute} + T_{tx}) \times num_x_lines + T_{wait}$

which is shorter by

$T_{wait} \times (num_x_lines - 1)$

assuming of course that T_{wait} is less than or equal to $T_{compute}$.

Note that we will ultimately be constrained by:

$T_{compute} + T_{tx} \times num_x_lines$

because we'll have to incur at least one full computation, and we'll have to transmit the data out the hardware—while we can use multithreading to overlay the computation cycles, we have only one hardware resource for the transmit.

Now, if we created a four-thread version and ran it on an SMP system with four CPUs, we'd end up with something that looked like this:

Figure 9: Four threads, four CPUs.

Notice how each of the four CPUs is underutilized (as indicated by the empty rectangles in the "utilization" graph). There are two interesting areas in the figure above. When the four threads start, they each compute. Unfortunately, when the threads are finished each computation, they're contending for the transmit hardware (the "X" parts in the figure are offset—only one transmission may be in progress at a time). This gives us a small anomaly in the startup part. Once the threads are past this stage, they're naturally synchronized to the transmit hardware, since the time to transmit is much smaller than ¼ of a compute cycle. Ignoring the small anomaly at the beginning, this system is characterized by the formula:

$$(T_{compute} + T_{tx} + T_{wait}) \times num_x_lines \,/\, num_cpus$$

This formula states that using four threads on four CPUs will be approximately 4 times faster than the single-threaded model we started out with.

By combining what we learned from simply having a multithreaded single-processor version, we would ideally like to have more threads than CPUs, so that the extra threads can "soak up" the idle CPU time from the transmit acknowledge waits (and the transmit slot contention waits) that naturally occur. In that case, we'd have something like this:

© 2020, QNX Software Systems Limited

Figure 10: Eight threads, four CPUs.

This figure assumes a few things:

- threads 5, 6, 7, and 8 are bound to processors 1, 2, 3, and 4 (for simplification)
- once a transmit begins it does so at a higher priority than a computation
- a transmit is a non-interruptible operation

Notice from the diagram that even though we now have twice as many threads as CPUs, we still run into places where the CPUs are underutilized. In the diagram, there are three such places where the CPU is “stalled”; these are indicated by numbers in the individual CPU utilization bar graphs:

1. Thread 1 was waiting for the acknowledgment (the “W” state), while thread 5 had completed a calculation and was waiting for the transmitter.
2. Both thread 2 and thread 6 were waiting for an acknowledgment.
3. Thread 3 was waiting for the acknowledgment while thread 7 had completed a calculation and was waiting for the transmitter.

This example also serves as an important lesson—you can't just keep adding CPUs in the hopes that things will keep getting faster. There are limiting factors. In some cases, these limiting factors are simply governed by the design of the multi-CPU motherboard—how much memory and device contention occurs when many CPUs try to access the same area of memory. In our case, notice that the “TX Slot Utilization” bar graph was starting to become full. If we added enough CPUs, they would eventually run into problems because their threads would be stalled, waiting to transmit.

In any event, by using “soaker” threads to “soak up” spare CPU, we now have much better CPU utilization. This utilization approaches:

$(T_{compute} + T_{tx}) \times num_x_lines / num_cpus$

In the computation *per se*, we're limited only by the amount of CPU we have; we're not idling any processor waiting for acknowledgment. (Obviously, that's the ideal case. As you saw in the diagram there are a few times when we're idling one CPU periodically. Also, as noted above,

$T_{compute} + T_{tx} \times num_x_lines$

is our limit on how fast we can go.)

Things to watch out for when using SMP

While in general you can simply "ignore" whether or not you're running on an SMP architecture or a single processor, there are certain things that *will* bite you. Unfortunately, they may be such low-probability events that they won't show up during development but rather during testing, demos, or the worst: out in the field. Taking a few moments now to program defensively will save problems down the road.

Here are the kinds of things that you're going to run up against on an SMP system:

- Threads really *can* and *do* run concurrently—relying on things like FIFO scheduling or prioritization for synchronization is a no-no.
- Threads and Interrupt Service Routines (ISRs) also *do* run concurrently—this means that not only will you have to protect the thread from the ISR, but you'll also have to protect the ISR from the thread. See the *Interrupts* chapter for more details.
- Some operations that you'd expect to be atomic aren't, depending on the operation and processor. Notable operations in this list are things that do a read-modify-write cycle (e.g., `++`, `--`, `|=`, `&=` etc.). See the include file **<atomic.h>** for replacements. (Note that this isn't purely an SMP issue; most RISC processors don't necessarily perform the above code in an atomic manner.)

Threads in independent situations

As discussed above in the "Where a thread is a good idea" section, threads also find use where a number of independent processing algorithms are occurring with shared data structures. While strictly speaking you *could* have a number of *processes* (each with one thread) explicitly sharing data, in some cases it's far more convenient to have a number of threads in one process instead. Let's see why and where you'd use threads in this case.

For our examples, we'll evolve a standard input/process/output model. In the most generic sense, one part of the model is responsible for getting input from somewhere, another part is responsible for processing the input to produce some form of output (or control), and the third part is responsible for feeding the output somewhere.

© 2020, QNX Software Systems Limited

Multiple processes

Let's first understand the situation from a multiple process, one-thread-per-process outlook. In this case, we'd have three processes, literally an input process, a “processing” process, and an output process:

Figure 11: System 1: Multiple operations, multiple processes.

This is the most highly abstracted form, and also the most “loosely coupled.” The “input” process has no real “binding” with either of the “processing” or “output” processes—it's simply responsible for gathering input and somehow giving it to the next stage (the “processing” stage). We could say the same thing of the “processing” and “output” processes—they too have no real binding with each other. We are also assuming in this example that the communication path (i.e., the input-to-processing and the processing-to-output data flow) is accomplished over some connectioned protocol (e.g., pipes, POSIX message queues, native QNX Neutrino message passing—whatever).

Multiple processes with shared memory

Depending on the volume of data flow, we may want to optimize the communication path.

The easiest way of doing this is to make the coupling between the three processes tighter. Instead of using a general-purpose connectioned protocol, we now choose a shared memory scheme (in the diagram, the thick lines indicate data flow; the thin lines, control flow):

Figure 12: System 2: Multiple operations, shared memory between processes.

In this scheme, we've tightened up the coupling, resulting in faster and more efficient data flow. We may still use a “general-purpose” connectioned protocol to transfer “control” information around—we're not expecting the control information to consume a lot of bandwidth.

Multiple threads

The most tightly-coupled system is represented by the following scheme:

Figure 13: System 3: Multiple operations, multiple threads.

Here we see one process with three threads. The three threads share the data areas implicitly. Also, the control information may be implemented as it was in the previous examples, or it may also be implemented via some of the thread synchronization primitives (we've seen mutexes, barriers, and semaphores; we'll see others in a short while).

Comparisons

Now, let's compare the three methods using various categories, and we'll also describe some of the trade-offs.

With system 1, we see the loosest coupling. This has the advantage that each of the three processes can be easily (i.e., via the command line, as opposed to recompile/redesign) replaced with a different module. This follows naturally, because the "unit of modularity" is the entire module itself. System 1 is also the only one that can be distributed among multiple nodes in a QNX Neutrino network. Since the communications pathway is abstracted over some connectioned protocol, it's easy to see that the three processes can be executing on any machine in the network. This may be a very powerful *scalability* factor for your design—you may need your system to scale up to having hundreds of machines distributed geographically (or in other ways, e.g., for peripheral hardware capability) and communicating with each other.

Once we commit to a shared memory region, however, we lose the ability to distribute over a network. QNX Neutrino doesn't support network-distributed shared memory objects. So in system 2, we've effectively limited ourselves to running all three processes on the same box. We haven't lost the ability to easily remove or change a component, because we still have separate processes that can be controlled from the command line. But we have added the constraint that all the removable components need to conform to the shared-memory model.

In system 3, we've lost all the above abilities. We definitely can't run different threads from one process on multiple nodes (we *can* run them on different processors in an SMP system, though). And we've lost our configurability aspects—now we need to have an explicit mechanism to define which "input," "processing," or "output" algorithm we want to use (which we can solve with shared objects, also known as DLLs.)

So why would I design my system to have multiple threads like system 3? Why not go for the maximally flexible system 1?

Well, even though system 3 is the most *inflexible*, it *is* most likely going to be the fastest. There are no thread-to-thread context switches for threads in different processes, I don't have to set up memory sharing explicitly, and I don't have to use *abstracted* synchronization methods like pipes, POSIX message

© 2020, QNX Software Systems Limited

queues, or message passing to deliver the data or control information—I can use basic kernel-level thread-synchronization primitives. Another advantage is that when the system described by the one process (with the three threads) starts, I *know* that everything I need has been loaded off the storage medium (i.e., I'm not going to find out later that “Oops, the processing driver is missing from the disk!”). Finally, system 3 is also most likely going to be the smallest, because we won't have three individual copies of “process” information (e.g., file descriptors).

To sum up: know what the trade-offs are, and use what works for your design.

More on synchronization

We've already seen:

- *mutexes*
- *semaphores*
- *barriers*

Let's now finish up our discussion of synchronization by talking about:

- *readers/writer locks*
- *sleepon locks*
- *condition variables*
- *additional OS services*

followed by an *example*.

Readers/writer locks

Readers and writer locks are used for exactly what their name implies: multiple readers can be using a resource, with no writers, or one writer can be using a resource with no other writers or readers.

This situation occurs often enough to warrant a special kind of synchronization primitive devoted exclusively to that purpose.

Often you'll have a data structure that's shared by a bunch of threads. Obviously, only one thread can be writing to the data structure at a time. If more than one thread was writing, then the threads could potentially overwrite each other's data. To prevent this from happening, the writing thread would obtain the “rwlock” (the readers/writer lock) in an exclusive manner, meaning that it and only it has access to the data structure. Note that the exclusivity of the access is controlled strictly by voluntary means. It's up to you, the system designer, to ensure that *all* threads that touch the data area synchronize by using the rwlocks.

The opposite occurs with readers. Since reading a data area is a non-destructive operation, any number of threads can be reading the data (even if it's the same piece of data that another thread is reading). An implicit point here is that *no* threads can be writing to the data area while *any* thread or threads are reading from it. Otherwise, the reading threads may be confused by reading a part of the data, getting preempted by a writing thread, and then, when the reading thread resumes, continue reading data, but from a newer “update” of the data. A data inconsistency would then result.

Let's look at the calls that you'd use with rwlocks.

The first two calls are used to initialize the library's internal storage areas for the rwlocks:

```
int
pthread_rwlock_init (pthread_rwlock_t *lock,
                     const pthread_rwlockattr_t *attr);

int
pthread_rwlock_destroy (pthread_rwlock_t *lock);
```


© 2020, QNX Software Systems Limited

The *pthread_rwlock_init()* function takes the *lock* argument (of type `pthread_rwlock_t`) and initializes it based on the attributes specified by *attr*. We're just going to use an attribute of NULL in our examples, which means, "Use the defaults." For detailed information about the attributes, see the entries in the QNX Neutrino *C Library Reference* for *pthread_rwlockattr_init()*, *pthread_rwlockattr_destroy()*, *pthread_rwlockattr_getpshared()*, and *pthread_rwlockattr_setpshared()*.

When done with the rwlock, you'd typically call *pthread_rwlock_destroy()* to destroy the lock, which invalidates it. You should never use a lock that is either destroyed or hasn't been initialized yet.

Next we need to fetch a lock of the appropriate type. As mentioned above, there are basically two modes of locks: a reader will want "non-exclusive" access, and a writer will want "exclusive" access. To keep the names simple, the functions are named after the user of the locks:

```
int
pthread_rwlock_rdlock (pthread_rwlock_t *lock);

int
pthread_rwlock_tryrdlock (pthread_rwlock_t *lock);

int
pthread_rwlock_wrlock (pthread_rwlock_t *lock);

int
pthread_rwlock_trywrlock (pthread_rwlock_t *lock);
```

There are four functions instead of the two that you may have expected. The "expected" functions are *pthread_rwlock_rdlock()* and *pthread_rwlock_wrlock()*, which are used by readers and writers, respectively. These are blocking calls—if the lock isn't available for the selected operation, the thread will block. When the lock becomes available in the appropriate mode, the thread will unblock. Because the thread unblocked from the call, it can now assume that it's safe to access the resource protected by the lock.

Sometimes, though, a thread won't want to block, but instead will want to see if it *could* get the lock. That's what the "try" versions are for. It's important to note that the "try" versions *will obtain the lock if they can*, but if they can't, then they won't block, but instead will just return an error indication. The reason they have to obtain the lock if they can is simple. Suppose that a thread wanted to obtain the lock for reading, but didn't want to wait in case it wasn't available. The thread calls *pthread_rwlock_tryrdlock()*, and is told that it could have the lock. If the *pthread_rwlock_tryrdlock() didn't* allocate the lock, then bad things could happen—another thread could preempt the one that was told to go ahead, and the second thread could lock the resource in an incompatible manner. Since the first thread wasn't actually given the lock, when the first thread goes to actually acquire the lock (because it was told it could), it would use *pthread_rwlock_rdlock()*, and now it would block, because the resource was no longer available in that mode. So, if we didn't lock it if we could, the thread that called the "try" version could still potentially block anyway!

Finally, regardless of the way that the lock was used, we need some way of releasing the lock:

```
int
pthread_rwlock_unlock (pthread_rwlock_t *lock);
```

Once a thread has done whatever operation it wanted to do on the resource, it would release the lock by calling *pthread_rwlock_unlock()*. If the lock is now available in a mode that corresponds to the mode requested by another waiting thread, then that thread would be made READY.

Note that we can't implement this form of synchronization with just a mutex. The mutex acts as a single-threading agent, which would be okay for the writing case (where you want only one thread to be using the resource at a time) but would fall flat in the reading case, because only one reader would be allowed. A semaphore couldn't be used either, because there's no way to distinguish the two modes of access—a semaphore would allow multiple readers, but if a writer were to acquire the semaphore, as far as the semaphore is concerned this would be no different from a reader acquiring it, and now you'd have the ugly situation of multiple readers and one *or more* writers! We'll see this in "*An example of synchronization*," later in this chapter.

One final note on the use of rwlocks: they aren't recursive and therefore aren't promotive. In other words, you can't lock your rwlock over and over again without unlocking first. You also can't lock your rwlock for reading and then for writing without unlocking first. This tradeoff was done to keep the rwlock as lightweight (in terms of size and speed) as possible.

Sleepon locks

Another common situation that occurs in multithreaded programs is the need for a thread to wait until "something happens." This "something" could be anything! It could be the fact that data is now available from a device, or that a conveyor belt has now moved to the proper position, or that data has been committed to disk, or whatever. Another twist to throw in here is that several threads may need to wait for the given event.

To accomplish this, we'd use either a *condition variable* (which we'll see next) or the much simpler "sleepon" lock.

To use sleepon locks, you actually need to perform several operations. Let's look at the calls first, and then look at how you'd use the locks.

```
int
pthread_sleepon_lock (void);

int
pthread_sleepon_unlock (void);

int
pthread_sleepon_broadcast (void *addr);

int
pthread_sleepon_signal (void *addr);

int
pthread_sleepon_wait (void *addr);
```


© 2020, QNX Software Systems Limited

Don't be tricked by the prefix *pthread_* into thinking that these are POSIX functions—they're not.

As described above, a thread needs to wait for something to happen. The most obvious choice in the list of functions above is the *pthread_sleepon_wait()*. But first, the thread needs to check if it really *does* have to wait. Let's set up an example. One thread is a producer thread that's getting data from some piece of hardware. The other thread is a consumer thread that's doing some form of processing on the data that just arrived. Let's look at the consumer first:

```
volatile int data_ready = 0;

consumer ()
{
    while (1) {
        while (!data_ready) {
            // WAIT
        }
        // process data
    }
}
```

The consumer is sitting in its main processing loop (the `while (1)`); it's going to do its job forever. The first thing it does is look at the *data_ready* flag. If this flag is a `0`, it means there's no data ready. Therefore, the consumer should wait. Somehow, the producer will wake it up, at which point the consumer should reexamine its *data_ready* flag. Let's say that's exactly what happens, and the consumer looks at the flag and decides that it's a `1`, meaning data is now available. The consumer goes off and processes the data, and then goes to see if there's more work to do, and so on.

We're going to run into a problem here. How does the consumer reset the *data_ready* flag in a synchronized manner with the producer? Obviously, we're going to need some form of exclusive access to the flag so that only one of those threads is modifying it at a given time. The method that's used in this case is built with a mutex, but it's a mutex that's buried in the implementation of the sleepon library, so we can access it only via two functions: *pthread_sleepon_lock()* and *pthread_sleepon_unlock()*. Let's modify our consumer:

```
consumer ()
{
    while (1) {
        if (pthread_sleepon_lock () == EOK)
        {
           while (!data_ready)
           {
               // WAIT
           }
           // process data
           data_ready = 0;
```

```
            pthread_sleepon_unlock ();
        }
    }
}
```

Now we've added the lock and unlock around the operation of the consumer. This means that the consumer can now *reliably* test the *data_ready* flag, with no race conditions, and also reliably set the flag.

Okay, great. Now what about the "WAIT" call? As we suggested earlier, it's effectively the *pthread_sleepon_wait()* call. Here's the second `while` loop:

```
        while (!data_ready) {
            pthread_sleepon_wait (&data_ready);
        }
```

The *pthread_sleepon_wait()* actually does three distinct steps!

1. Unlock the sleepon library mutex.
2. Perform the waiting operation.
3. Re-lock the sleepon library mutex.

The reason it has to unlock and lock the sleepon library's mutex is simple—since the whole idea of the mutex is to ensure mutual exclusion to the *data_ready* variable, this means that we want to lock out the producer from touching the *data_ready* variable while we're testing it. But, if we don't do the unlock part of the operation, the producer would never be able to set it to tell us that data is indeed available! The re-lock operation is done purely as a convenience; this way the user of the *pthread_sleepon_wait()* doesn't have to worry about the state of the lock when it wakes up.

Let's switch over to the producer side and see how it uses the sleepon library. Here's the full implementation:

```
producer ()
{
    while (1) {
        // wait for interrupt from hardware here...
        if (pthread_sleepon_lock () == EOK)
        {
            data_ready = 1;
            pthread_sleepon_signal (&data_ready);
            pthread_sleepon_unlock ();
        }
    }
}
```

As you can see, the producer locks the mutex as well so that it can have exclusive access to the *data_ready* variable in order to set it.

© 2020, QNX Software Systems Limited

It's *not* the act of writing a `1` to *data_ready* that awakens the client! It's the call to *pthread_sleepon_signal()* that does it.

We've identified the consumer and producer states as:

State	Meaning
CONDVAR	Waiting for the underlying condition variable associated with the sleepon
MUTEX	Waiting for a mutex
READY	Capable of using, or already using, the CPU
INTERRUPT	Waiting for an interrupt from the hardware

Let's examine in detail what happens:

Action	Mutex owner	Consumer state	Producer state
Consumer locks mutex	Consumer	READY	INTERRUPT
Consumer examines *data_ready*	Consumer	READY	INTERRUPT
Consumer calls *pthread_sleepon_wait()*	Consumer	READY	INTERRUPT
pthread_sleepon_wait() unlocks mutex	Free	READY	INTERRUPT
pthread_sleepon_wait() blocks	Free	CONDVAR	INTERRUPT
Time passes	Free	CONDVAR	INTERRUPT
Hardware generates data	Free	CONDVAR	READY
Producer locks mutex	Producer	CONDVAR	READY
Producer sets *data_ready*	Producer	CONDVAR	READY
Producer calls *pthread_sleepon_signal()*	Producer	CONDVAR	READY
Consumer wakes up, *pthread_sleepon_wait()* tries to lock mutex	Producer	MUTEX	READY
Producer releases mutex	Free	MUTEX	READY
Consumer gets mutex	Consumer	READY	READY
Consumer processes data	Consumer	READY	READY
Producer waits for more data	Consumer	READY	INTERRUPT

Action	Mutex owner	Consumer state	Producer state
Time passes (consumer processing)	Consumer	READY	INTERRUPT
Consumer finishes processing, unlocks mutex	Free	READY	INTERRUPT
Consumer loops back to top, locks mutex	Consumer	READY	INTERRUPT

The last entry in the table is a repeat of the first entry—we've gone around one complete cycle.

What's the purpose of the *data_ready* variable? It actually serves two purposes:

- It's the status flag between the consumer and the producer that indicates the state of the system. If it's set to a `1`, it means that data is available for processing; if it's set to a `0`, it means that no data is available, and the consumer should block.
- It serves as "the place where sleepon synchronization occurs." More formally, the *address* of *data_ready* is used as a unique identifier, that serves as the rendezvous object for sleepon locks. We just as easily could have used "`(void *) 12345`" instead of "`&data_ready`"—so long as the identifier is unique and used consistently, the sleepon library really doesn't care. Actually, using the address of a variable in a process is a guaranteed way to generate a process-unique number—after all, no two variables in a process will have the same address!

We'll defer the discussion of "What's the difference between *pthread_sleepon_signal()* and *pthread_sleepon_broadcast()*" to the discussion of condition variables next.

Condition variables

Condition variables (or *condvars*) are remarkably similar to the sleepon locks we just saw above. In fact, sleepon locks are built on top of condvars, which is why we had a state of CONDVAR in the explanation table for the sleepon example. It bears repeating that the *pthread_cond_wait()* function releases the mutex, waits, and then reacquires the mutex, just like the *pthread_sleepon_wait()* function did.

Let's skip the preliminaries and redo the example of the producer and consumer from the sleepon section, using condvars instead. Then we'll discuss the calls.

```
/*
 * cp1.c
*/

#include <stdio.h>
#include <pthread.h>

int data_ready = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  condvar = PTHREAD_COND_INITIALIZER;

void *
```


© 2020, QNX Software Systems Limited

```
consumer (void *notused)
{
    printf ("In consumer thread...\n");
    while (1) {
        pthread_mutex_lock (&mutex);
        while (!data_ready) {
            pthread_cond_wait (&condvar, &mutex);
        }
        // process data
        printf ("consumer:  got data from producer\n");
        data_ready = 0;
        pthread_cond_signal (&condvar);
        pthread_mutex_unlock (&mutex);
    }
}

void *
producer (void *notused)
{
    printf ("In producer thread...\n");
    while (1) {
        // get data from hardware
        // we'll simulate this with a sleep (1)
        sleep (1);
        printf ("producer:  got data from h/w\n");
        pthread_mutex_lock (&mutex);
        while (data_ready) {
            pthread_cond_wait (&condvar, &mutex);
        }
        data_ready = 1;
        pthread_cond_signal (&condvar);
        pthread_mutex_unlock (&mutex);
    }
}

main ()
{
    printf ("Starting consumer/producer example...\n");

    // create the producer and consumer threads
    pthread_create (NULL, NULL, producer, NULL);
    pthread_create (NULL, NULL, consumer, NULL);
```

```
    // let the threads run for a bit
    sleep (20);
}
```

Pretty much identical to the sleepon example we just saw, with a few variations (we also added some *printf()* functions and a *main()* so that the program would run!). Right away, the first thing that we see is a new data type: `pthread_cond_t`. This is simply the declaration of the condition variable; we've called ours *condvar*.

Next thing we notice is that the structure of the consumer is identical to that of the consumer in the previous sleepon example. We've replaced the *pthread_sleepon_lock()* and *pthread_sleepon_unlock()* with the standard mutex versions (*pthread_mutex_lock()* and *pthread_mutex_unlock()*). The *pthread_sleepon_wait()* was replaced with *pthread_cond_wait()*. The main difference is that the sleepon library has a mutex buried deep within it, whereas when we use condvars, we explicitly pass the mutex. We get a lot more flexibility this way.

Finally, we notice that we've got *pthread_cond_signal()* instead of *pthread_sleepon_signal()* (again with the mutex passed explicitly).

Signal versus broadcast

In the sleepon section, we promised to talk about the difference between the *pthread_sleepon_signal()* and *pthread_sleepon_broadcast()* functions. In the same breath, we'll talk about the difference between the two condvar functions *pthread_cond_signal()* and *pthread_cond_broadcast()*.

The short story is this: the “signal” version will wake up only one thread. So, if there were multiple threads blocked in the “wait” function, and a thread did the “signal,” then only one of the threads would wake up. Which one? The highest priority one. If there are two or more at the same priority, the ordering of wakeup is indeterminate. With the “broadcast” version, *all* blocked threads will wake up.

It may seem wasteful to wake up all threads. On the other hand, it may seem sloppy to wake up only one (effectively random) thread.

Therefore, we should look at where it makes sense to use one over the other. Obviously, if you have only one thread waiting, as we did in either version of the consumer program, a “signal” will do just fine—one thread will wake up and, guess what, it'll be the only thread that's currently waiting.

In a multithreaded situation, we've got to ask: “Why are these threads waiting?” There are usually two possible answers:

- All the threads are considered equivalent and are effectively forming a “pool” of available threads that are ready to handle some form of request.

Or:

- The threads are all unique and are each waiting for a very specific condition to occur.

In the first case, we can imagine that all the threads have code that might look like the following:

```
/*
 * cv1.c
*/

#include <stdio.h>
#include <pthread.h>
```


© 2020, QNX Software Systems Limited

```
pthread_mutex_t mutex_data = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cv_data = PTHREAD_COND_INITIALIZER;
int data;

thread1 ()
{
    for (;;) {
        pthread_mutex_lock (&mutex_data);
        while (data == 0) {
            pthread_cond_wait (&cv_data, &mutex_data);
        }
        // do something
        pthread_mutex_unlock (&mutex_data);
    }
}

// thread2, thread3, etc have the identical code.
```

In this case, it really doesn't matter *which* thread gets the data, provided that one of them gets it and does something with it.

However, if you have something like this, things are a little different:

```
/*
 * cv2.c
*/

#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mutex_xy = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cv_xy = PTHREAD_COND_INITIALIZER;
int x, y;

int isprime (int);

thread1 ()
{
    for (;;) {
        pthread_mutex_lock (&mutex_xy);
        while ((x > 7) && (y != 15)) {
            pthread_cond_wait (&cv_xy, &mutex_xy);
        }
        // do something
        pthread_mutex_unlock (&mutex_xy);
    }
}

thread2 ()
{
    for (;;) {
        pthread_mutex_lock (&mutex_xy);
        while (!isprime (x)) {
            pthread_cond_wait (&cv_xy, &mutex_xy);
```

```
        }
        // do something
        pthread_mutex_unlock (&mutex_xy);
    }
}

thread3 ()
{
    for (;;) {
        pthread_mutex_lock (&mutex_xy);
        while (x != y) {
            pthread_cond_wait (&cv_xy, &mutex_xy);
        }
        // do something
        pthread_mutex_unlock (&mutex_xy);
    }
}
```

In these cases, waking up one thread isn't going to cut it! We *must* wake up all three threads and have each of them check to see if its predicate has been satisfied or not.

This nicely reflects the second case in our question above ("Why are these threads waiting?"). Since the threads are all waiting on different conditions (*thread1()* is waiting for *x* to be less than or equal to 7 or *y* to be 15, *thread2()* is waiting for *x* to be a prime number, and *thread3()* is waiting for *x* to be equal to *y*), we have no choice but to wake them all.

Sleepons versus condvars

Sleepons have one principal advantage over condvars. Suppose that you want to synchronize many objects. With condvars, you'd typically associate one condvar per object. Therefore, if you had *M* objects, you'd most likely have *M* condvars. With sleepons, the underlying condvars (on top of which sleepons are implemented) are allocated dynamically as threads wait for a particular object. Therefore, using sleepons with *M* objects and *N* threads blocked, you'd have (at most) *N* condvars (instead of *M*).

However, condvars are more flexible than sleepons, because:

1. Sleepons are built on top of condvars anyway.
2. Sleepons have the mutex buried in the library; condvars allow you to specify it explicitly.

The first point might just be viewed as being argumentative. `:-)` The second point, however, is significant. When the mutex is buried in the library, this means that there can be only *one* per process—regardless of the number of threads in that process, or the number of different "sets" of data variables. This can be a very limiting factor, especially when you consider that you must use the *one and only* mutex to access *any and all* data variables that *any* thread in the process needs to touch!

A much better design is to use multiple mutexes, one for each data set, and explicitly combine them with condition variables as required. The true power and danger of this approach is that there is absolutely *no* compile- or runtime checking to make sure that you:

- have locked the mutex before manipulating a variable
- are using the correct mutex for the particular variable
- are using the correct condvar with the appropriate mutex and variable

© 2020, QNX Software Systems Limited

The easiest way around these problems is to have a good design and a design review, and also to borrow techniques from object-oriented programming (like having the mutex contained in a data structure, having routines to access the data structure, etc.). Of course, how much of one or both you apply depends not only on your personal style, but also on performance requirements.

The key points to remember when using condvars are:

1. The mutex is to be used for testing and accessing the variables.
2. The condvar is to be used as a rendezvous point.

Here's a picture:

Figure 14: One-to-one mutex and condvar associations.

One interesting note. Since there *is* no checking, you can do things like associate one set of variables with mutex “ABC,” and another set of variables with mutex “DEF,” while associating *both* sets of variables with condvar “ABCDEF:”

Figure 15: Many-to-one mutex and condvar associations.

This is actually quite useful. Since the mutex is always to be used for “access and testing,” this implies that I have to choose the correct mutex whenever I want to look at a particular variable. Fair enough—if I'm examining variable “C,” I obviously need to lock mutex “MutexABC.” What if I changed variable “E”? Well, before I change it, I had to acquire the mutex “MutexDEF.” Then I changed it, and hit condvar “CondvarABCDEF” to tell others about the change. Shortly thereafter, I would release the mutex.

Now, consider what happens. Suddenly, I have a bunch of threads that had been waiting on “CondvarABCDEF” that now wake up (from their *pthread_cond_wait()*). The waiting function immediately attempts to reacquire the mutex. The critical point here is that there are two mutexes to acquire. This means that on an SMP system, *two concurrent streams* of threads can run, each examining what it considers to be independent variables, using independent mutexes. Cool, eh?

Additional OS services

QNX Neutrino lets you do something else that's elegant. POSIX says that a mutex must operate between threads in the same process, and lets a conforming implementation extend that. QNX Neutrino extends this by allowing a mutex to operate between threads in *different* processes. To understand why this works, recall that there really are two parts to what's viewed as the "operating system"—the kernel, which deals with scheduling, and the process manager, which worries about memory protection and "processes" (among other things). A mutex is really just a synchronization object used between threads. Since the kernel worries only about threads, it really doesn't care that the threads are operating in different processes—this is an issue for the process manager.

So, if you've set up a shared memory area between two processes, and you've initialized a mutex in that shared memory, there's nothing stopping you from synchronizing multiple threads in those two (or more!) processes via the mutex. The same *pthread_mutex_lock()* and *pthread_mutex_unlock()* functions will still work.

Pools of threads

Another thing that QNX Neutrino has added is the concept of thread pools. You'll often notice in your programs that you want to be able to run a certain number of threads, but you also want to be able to control the behavior of those threads within certain limits. For example, in a server you may decide that initially just one thread should be blocked, waiting for a message from a client. When that thread gets a message and is off servicing a request, you may decide that it would be a good idea to create another thread, so that it could be blocked waiting in case another request arrived. This second thread would then be available to handle that request. And so on. After a while, when the requests had been serviced, you would now have a large number of threads sitting around, waiting for further requests. In order to conserve resources, you may decide to kill off some of those "extra" threads.

This is in fact a common operation, and QNX Neutrino provides a library to help with this. We'll see the thread pool functions again in the *Resource Managers* chapter.

It's important for the discussions that follow to realize there are really two distinct operations that threads (that are used in thread pools) perform:

- a blocking (waiting operation)
- a processing operation

The blocking operation doesn't generally consume CPU. In a typical server, this is where the thread is waiting for a message to arrive. Contrast that with the processing operation, where the thread may or may not be consuming CPU (depending on how the process is structured). In the thread pool functions that we'll look at later, you'll see that we have the ability to control the number of threads in the blocking operation as well as the number of threads that are in the processing operations.

QNX Neutrino provides the following functions to deal with thread pools:

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

thread_pool_t *
thread_pool_create (thread_pool_attr_t *attr,
                    unsigned flags);
```


© 2020, QNX Software Systems Limited

```
int
thread_pool_destroy (thread_pool_t *pool);

int
thread_pool_start (void *pool);

int
thread_pool_limits (thread_pool_t *pool,
                    int lowater,
                    int hiwater,
                    int maximum,
                    int increment,
                    unsigned flags);

int
thread_pool_control (thread_pool_t *pool,
                     thread_pool_attr_t *attr,
                     uint16_t lower,
                     uint16_t upper,
                     unsigned flags);
```

As you can see from the functions provided, you first create a thread pool definition using *thread_pool_create()*, and then start the thread pool via *thread_pool_start()*. When you're done with the thread pool, you can use *thread_pool_destroy()* to clean up after yourself. Note that you might never call *thread_pool_destroy()*, as in the case where the program is a server that runs "forever." The *thread_pool_limits()* function is used to specify thread pool behavior and adjust attributes of the thread pool, and the *thread_pool_control()* function is a convenience wrapper for the *thread_pool_limits()* function.

So, the first function to look at is *thread_pool_create()*. It takes two parameters, *attr* and *flags*. The *attr* is an attributes structure that defines the operating characteristics of the thread pool (from **<sys/dispatch.h>**):

```
typedef struct _thread_pool_attr {
    // thread pool functions and handle
    THREAD_POOL_HANDLE_T    *handle;

    THREAD_POOL_PARAM_T
        *(*block_func)(THREAD_POOL_PARAM_T *ctp);

    void
        (*unblock_func)(THREAD_POOL_PARAM_T *ctp);

    int
```

```
        (*handler_func)(THREAD_POOL_PARAM_T *ctp);

    THREAD_POOL_PARAM_T
        *(*context_alloc)(THREAD_POOL_HANDLE_T *handle);

    void
        (*context_free)(THREAD_POOL_PARAM_T *ctp);

    // thread pool parameters
    pthread_attr_t          *attr;
    unsigned short          lo_water;
    unsigned short          increment;
    unsigned short          hi_water;
    unsigned short          maximum;
} thread_pool_attr_t;
```

I've broken the `thread_pool_attr_t` type into two sections, one that contains the functions and handle for the threads in the thread pool, and another that contains the operating parameters for the thread pool.

Controlling the number of threads

Let's first look at the "thread pool parameters" to see how you control the number and attributes of threads that will be operating in this thread pool. Keep in mind that we'll be talking about the "blocking operation" and the "processing operation" (when we look at the callout functions, we'll see how these relate to each other).

The following diagram illustrates the relationship of the *lo_water*, *hi_water*, and *maximum* parameters:

Figure 16: Thread flow when using thread pools.

(Note that "CA" is the *context_alloc()* function, "CF" is the *context_free()* function, "blocking operation" is the *block_func()* function, and "processing operation" is the *handler_func()*.)

attr

This is the attributes structure that's used during thread creation. We've already discussed this structure above (in *"The thread attributes structure"*). You'll recall that this is the structure that controls things about the newly created thread like priority, stack size, and so on.

© 2020, QNX Software Systems Limited

lo_water

There should always be at least *lo_water* threads sitting in the blocking operation. In a typical server, this would be the number of threads waiting to receive a message, for example. If there are less than *lo_water* threads sitting in the blocking operation (because, for example, we just received a message and have started the processing operation on that message), then more threads are created, according to the *increment* parameter. This is represented in the diagram by the first step labeled "create thread."

increment

Indicates how many threads should be created at once if the count of blocking operation threads ever drops under *lo_water*. In deciding how to choose a value for this, you'd most likely start with `1`. This means that if the number of threads in the blocking operation drops under *lo_water*, exactly one more thread would be created by the thread pool. To fine-tune the number that you've selected for *increment*, you could observe the behavior of the process and determine whether this number needs to be anything other than one. If, for example, you notice that your process gets "bursts" of requests, then you might decide that once you've dropped below *lo_water* blocking operation threads, you're probably going to encounter this "burst" of requests, so you might decide to request the creation of more than one thread at a time.

hi_water

Indicates the upper limit on the number of threads that should be in the blocking operation. As threads complete their processing operations, they will normally return to the blocking operation. However, the thread pool library keeps count of how many threads are currently in the blocking operation, and if that number ever exceeds *hi_water*, the thread pool library will kill the thread that caused the overflow (i.e., the thread that had just finished and was about to go back to the blocking operation). This is shown in the diagram as the "split" out of the "processing operation" block, with one path going to the "blocking operation" and the other path going to "CF" to destroy the thread. The combination of *lo_water* and *hi_water*, therefore, allows you to specify a range indicating how many threads should be in the blocking operation.

maximum

Indicates the absolute maximum number of threads that will ever run concurrently as a result of the thread pool library. For example, if threads were being created as a result of an underflow of the *lo_water* mark, the *maximum* parameter would limit the total number of threads.

One other key parameter to controlling the threads is the *flags* parameter passed to the *thread_pool_create()* function. It can have one of the following values:

POOL_FLAG_EXIT_SELF

The *thread_pool_start()* function will not return, nor will the calling thread be incorporated into the pool of threads.

POOL_FLAG_USE_SELF

The *thread_pool_start()* function will not return, but the calling thread will be incorporated into the pool of threads.

0

The *thread_pool_start()* function will return, with new threads being created as required.

The above descriptions may seem a little dry. Let's look at an example.

You can find the complete version of *tp1.c* in the Sample Programs appendix. Here, we'll just focus on the *lo_water*, *hi_water*, *increment*, and the *maximum* members of the thread pool control structure:

```
/*
 * part of  tp1.c
*/

#include <sys/dispatch.h>

int
main ()
{
    thread_pool_attr_t  tp_attr;
    void                *tpp;

    ...
    tp_attr.lo_water  = 3;
    tp_attr.increment = 2;
    tp_attr.hi_water  = 7;
    tp_attr.maximum   = 10;
    ...

    tpp = thread_pool_create (&tp_attr, POOL_FLAG_USE_SELF);
    if (tpp == NULL) {
        fprintf (stderr,
                 "%s:  can't thread_pool_create, errno %s\n",
                 progname, strerror (errno));
        exit (EXIT_FAILURE);
    }

    thread_pool_start (tpp);
    ...
```

After setting the members, we call *thread_pool_create()* to create a thread pool. This returns a pointer to a thread pool control structure (*tpp*), which we check against NULL (which would indicate an error). Finally we call *thread_pool_start()* with the *tpp* thread pool control structure.

© 2020, QNX Software Systems Limited

I've specified POOL_FLAG_USE_SELF which means that the thread that called *thread_pool_start()* will be considered an available thread for the thread pool. So, at this point, there is only that one thread in the thread pool library. Since we have a *lo_water* value of 3, the library immediately creates *increment* number of threads (2 in this case). At this point, 3 threads are in the library, and all 3 of them are in the blocking operation. The *lo_water* condition is satisfied, because there are at least that number of threads in the blocking operation; the *hi_water* condition is satisfied, because there are less than that number of threads in the blocking operation; and finally, the *maximum* condition is satisfied as well, because we don't have more than that number of threads in the thread pool library.

Now, one of the threads in the blocking operation unblocks (e.g., in a server application, a message was received). This means that now one of the three threads is no longer in the blocking operation (instead, that thread is now in the processing operation). Since the count of blocking threads is less than the *lo_water*, it trips the *lo_water* trigger and causes the library to create *increment* (2) threads. So now there are 5 threads total (4 in the blocking operation, and 1 in the processing operation).

More threads unblock. Let's assume that none of the threads in the processing operation none completes any of their requests yet. Here's a table illustrating this, starting at the initial state (we've used “Proc Op” for the processing operation, and “Blk Op” for the blocking operation, as we did in the previous diagram, “Thread flow when using thread pools.”):

Event	Proc Op	Blk Op	Total
Initial	0	1	1
lo_water trip	0	3	3
Unblock	1	2	3
lo_water trip	1	4	5
Unblock	2	3	5
Unblock	3	2	5
lo_water trip	3	4	7
Unblock	4	3	7
Unblock	5	2	7
lo_water trip	5	4	9
Unblock	6	3	9
Unblock	7	2	9
lo_water trip	7	3	10
Unblock	8	2	10
Unblock	9	1	10

Event	Proc Op	Blk Op	Total
Unblock	10	0	10

As you can see, the library always checks the *lo_water* variable and creates *increment* threads at a time *until* it hits the limit of the *maximum* variable (as it did when the "Total" column reached 10—no more threads were being created, even though the count had underflowed the *lo_water*).

This means that at this point, there are no more threads waiting in the blocking operation. Let's assume that the threads are now finishing their requests (from the processing operation); watch what happens with the *hi_water* trigger:

Event	Proc Op	Blk Op	Total
Completion	9	1	10
Completion	8	2	10
Completion	7	3	10
Completion	6	4	10
Completion	5	5	10
Completion	4	6	10
Completion	3	7	10
Completion	2	8	10
hi_water trip	2	7	9
Completion	1	8	9
hi_water trip	1	7	8
Completion	0	8	8
hi_water trip	0	7	7

Notice how nothing really happened during the completion of processing for the threads *until* we tripped over the *hi_water* trigger. The implementation is that as soon as the thread finishes, it looks at the number of receive blocked threads and decides to kill itself if there are too many (i.e., more than *hi_water*) waiting at that point. The nice thing about the *lo_water* and *hi_water* limits in the structures is that you can effectively have an "operating range" where a sufficient number of threads are available, and you're not unnecessarily creating and destroying threads. In our case, after the operations performed by the above tables, we now have a system that can handle up to 4 requests simultaneously without creating more threads (7 - 4 = 3, which is the *lo_water* trip).

© 2020, QNX Software Systems Limited

The thread pool functions

Now that we have a good feel for how the number of threads is controlled, let's turn our attention to the other members of the thread pool attribute structure (from above):

```
// thread pool functions and handle
THREAD_POOL_HANDLE_T     *handle;

THREAD_POOL_PARAM_T
     *(*block_func)(THREAD_POOL_PARAM_T *ctp);

void
     (*unblock_func)(THREAD_POOL_PARAM_T *ctp);

int
     (*handler_func)(THREAD_POOL_PARAM_T *ctp);

THREAD_POOL_PARAM_T
     *(*context_alloc)(THREAD_POOL_HANDLE_T *handle);

void
     (*context_free)(THREAD_POOL_PARAM_T *ctp);
```

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

Recall from the diagram "Thread flow when using thread pools," that the *context_alloc()* function gets called for every new thread being created. (Similarly, the *context_free()* function gets called for every thread being destroyed.)

The *handle* member of the structure (above) is passed to the *context_alloc()* function as its sole parameter. The *context_alloc()* function is responsible for performing any *per-thread* setup required and for returning a context pointer (called *ctp* in the parameter lists). Note that the contents of the context pointer are entirely up to you—the library doesn't care what you put into the context pointer.

Now that the context has been created by *context_alloc()*, the *block_func()* function is called to perform the blocking operation. Note that the *block_func()* function gets passed the results of the *context_alloc()* function. Once the *block_func()* function unblocks, it returns a context pointer, which gets passed by the library to the *handler_func()*. The *handler_func()* is responsible for performing the "work"—for example, in a typical server, this is where the message from the client is processed. The *handler_func()* must return a zero for now—nonzero values are reserved for future expansion by QNX Software Systems. The *unblock_func()* is also reserved at this time; just leave it as NULL. Perhaps this pseudo code sample will clear things up (it's based on the same flow as shown in "Thread flow when using thread pools," above):

```
FOREVER DO
     IF (#threads < lo_water) THEN
```

```
        IF (#threads_total < maximum) THEN
            create new thread
            context = (*context_alloc) (handle);
        ENDIF
    ENDIF
    retval = (*block_func) (context);
    (*handler_func) (retval);
    IF (#threads > hi_water) THEN
        (*context_free) (context)
        kill thread
    ENDIF
DONE
```

Note that the above is greatly simplified; its only purpose is to show you the data flow of the *ctp* and *handle* parameters and to give some sense of the algorithms used to control the number of threads.

An example of synchronization

Let's look at an example of synchronization in a multithreaded program.

If the data is the most important part of your application, and that data gets corrupted, then your application might be of little use. It's pretty straightforward to make sure a single-threaded application doesn't accidentally corrupt its data. Multithreaded applications are quite another story.

Without synchronizing

The problem with multithreaded applications is that you could have one section of code modifying your data, while another section is reading and using that same data. If the write and read overlap, you could have some serious complications!

Let's illustrate this with a short application:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

typedef struct {
   int a;
   int b;
   int result;
   int result2;
   int use_count;
   int use_count2;
   int max_use;
   int max_use2;
} app_data;

void *user_thread(void *data)
```


© 2020, QNX Software Systems Limited

```
{
   int uses = 0;
   app_data *td = (app_data*)data;

   while(uses < td->max_use) {
      if (td->a == 5) {
         td->result += (td->a + td->b);
         td->use_count++;
         uses++;
      }
      usleep(1);
   }
   return 0;
}

void *changer_thread(void *data)
{
   app_data *td = (app_data*)data;

   while ((td->use_count + td->use_count2) < (td->max_use + td->max_use2)) {
      if (td->a == 5) {
         td->a = 50;
         td->b = td->a + usleep(1000);
      } else {
         td->a = 5;
         td->b = td->a + usleep(1000);
      }
      usleep(1);
   }
   return 0;
}

int main(int argc, char **argv)
{
   pthread_t ct, ut;
   app_data td = {5,5,0,0,0,0,100,0};
   void *retval;

   pthread_create(&ut, NULL, user_thread, &td);
   pthread_create(&ct, NULL, changer_thread, &td);
   pthread_join(ct, &retval);
   pthread_join(ut, &retval);
   printf("result should be %d; it's %d\n", td.max_use * (5 + 5), td.result);

   return EXIT_SUCCESS;
}
```

The *user_thread()* function reads the data in the passed `app_data` structure and uses the *a* and *b* variables *max_use* times whenever *a* is set to five. Its local *uses* variable counts the number of times the function has used the data. The function also increments the application's *use_count*, in case someone else wants to keep track of how many times the data has been used. This function adds a small *usleep()* to make sure the scheduler gives other threads a chance to run.

The *changer_thread()* function modifies the data, changing *a* and *b* until they've been used elsewhere—in *user_thread()*, in this case—the number of times specified by *max_use* plus *max_use2*. The change is simple: *changer_thread()* simply toggles the value of *a* between 5 and 50 and fakes a CPU-intensive calculation for *b* by calling `usleep(1000)`, which returns 0 unless an error occurs. This means that *a* will be changed, and 1 ms later *b* will be changed. Therefore there's a 1 ms gap where *a* and *b* shouldn't be used elsewhere in the application since their values are still in flux.

The *main()* routine creates an instance of the `app_data` structure, spawns the two threads, waits for them to finish, and then prints the results. If you build the program and run it, the results are something like this:

```
result should be 1000; it's 3205
```

Your actual result might be different, depending on your CPU speed. Why isn't it what we expected?

The problem is that *user_thread()* is waiting until *a* is set to 5, and when it is, it adds the *a* and *b* values (which should both be 5) to the *result* variable. However, since the *b* variable is taking so long to get calculated, *user_thread()* sometimes uses the old *b* value (50) instead of the new one (5). You need to protect your data somehow so that the *user_thread()* can't use the data until you've set it!

Using a mutex

Let's use a mutex to make sure that the data is accessed by only one thread at a time.

Mutexes are very easy to use. Once you've created a mutex, you simply put a *pthread_mutex_lock()* in front of the code you want to protect, and *pthread_mutex_unlock()* right after that code. In this case, you want to protect both the modification and use of the *a* and *b* variables.

Here's a summary of the changes to make to the program:

- Add the mutex to the `app_data` structure.
- Put the lock/unlock functions around the code you want to protect (i.e., where you're reading or writing the *a* and *b* variables) in both *user_thread()* and *changer_thread()*.
- Create and destroy the mutex in the *main()* function. In this example, we'll pass NULL as the *attr* argument to *pthread_mutex_init()* because we want to use the default attributes for the mutex.

Here's the program, with the updates shown in bold:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

typedef struct {
   int a;
   int b;
   int result;
```

 © 2020, QNX Software Systems Limited

```
   int result2;
   int use_count;
   int use_count2;
   int max_use;
   int max_use2;
   pthread_mutex_t mutex;
} app_data;

void *user_thread(void *data)
{
   int uses = 0;
   app_data *td = (app_data*)data;

   while(uses < td->max_use) {
      pthread_mutex_lock(&td->mutex);
      if (td->a == 5) {
         td->result += (td->a + td->b);
         td->use_count++;
         uses++;
      }
      pthread_mutex_unlock(&td->mutex);
      usleep(1);
   }
   return 0;
}

void *changer_thread(void *data)
{
   app_data *td = (app_data*)data;

   while ((td->use_count + td->use_count2) < (td->max_use + td->max_use2)) {
      pthread_mutex_lock(&td->mutex);
      if (td->a == 5) {
         td->a = 50;
         td->b = td->a + usleep(1000);
      } else {
         td->a = 5;
         td->b = td->a + usleep(1000);
      }
      pthread_mutex_unlock(&td->mutex);
      usleep(1);
   }
   return 0;
}

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

```
    pthread_t ct, ut;
    app_data td = {5,5,0,0,0,0,100,0};
    void *retval;

    pthread_mutex_init(&td.mutex, NULL);
    pthread_create(&ut, NULL, user_thread, &td);
    pthread_create(&ct, NULL, changer_thread, &td);
    pthread_join(ct, &retval);
    pthread_join(ut, &retval);
    pthread_mutex_destroy(&td.mutex);
    printf("result should be %d; it's %d\n", td.max_use * (5 + 5), td.result);

    return EXIT_SUCCESS;
}
```

This program should give the results we want:

```
result should be 1000; it's 1000
```

More than one reader

What if you have one thread that's changing the data, but more than one other thread that just wants to read it? Let's add a thread that needs to read *a* and *b* as well:

```
void *subtracter_thread(void *data)
{
    int use=0;
    app_data *td=(app_data*)data;

    while(use < td->max_use2) {
        pthread_mutex_lock(&td->mutex);
        if (td->a == 50) {
            td->result2 -= (td->a + td->b);
            use++;
            td->use_count2++;
        }
        pthread_mutex_unlock(&td->mutex);
        usleep(1);
    }
    return 0;
}
```

We need to modify *main()* to spawn the new thread and set the *max_use2* member in the `app_data` structure:

```
int main(int argc, char **argv)
{
    pthread_t ct, ut, st;
    app_data td = {5,5,0,0,0,0,100,100};
    void *retval;
```


© 2020, QNX Software Systems Limited

```
    pthread_mutex_init(&td.mutex, NULL);
    pthread_create(&ut, NULL, user_thread, &td);
    pthread_create(&ct, NULL, changer_thread, &td);
    pthread_create(&st, NULL, subtracter_thread, &td);
    pthread_join(ct, &retval);
    pthread_join(ut, &retval);
    pthread_join(st, &retval);
    pthread_mutex_destroy(&td.mutex);
    printf("result should be %d; it's %d\n", td.max_use * (5 + 5), td.result);
    printf("result2 should be %d; it's %d\n", -(td.max_use2 * (50 + 50)),
           td.result2);

    return EXIT_SUCCESS;
}
```

If you compile and run this, you get just what you wanted:

```
result should be 1000; it's 1000
result2 should be -10000; it's -10000
```

But wait: shouldn't *user_thread()* and *subtracter_thread()* be able to read *a* and *b* at the same time? (Remember that with mutexes, anything wrapped in the lock/unlock pair can't be executed at the same time as another piece of code wrapped in the lock/unlock pair.) Neither of them changes *a* or *b*, so it would be nice if you could let both of them read these members at the same time—just not when *changer_thread()* is changing them.

Using read-write locks

With readers/writer locks (or rwlocks), you can easily implement this kind of behavior. Unlike mutexes, rwlocks can be locked either as read or as write. As long as they're not locked for write access, any threads can lock for read and unlock as much as they want. However, if they're locked for write access, all read locks occurring afterwards are blocked until the write lock is unlocked.

Here's how to make our program use an rwlock:

- Change the mutex to an rwlock in the `app_data` structure.
- In *user_thread()* and *subtracter_thread()*, which read the data, change:
  - *pthread_mutex_lock()* to *pthread_rwlock_rdlock()*
  - *pthread_mutex_unlock()* to *pthread_rwlock_unlock()*
- In *changer_thread()*, which writes the data, change:
  - *pthread_mutex_lock()* to *pthread_rwlock_wrlock()*
  - *pthread_mutex_unlock()* to *pthread_rwlock_unlock()*
- Modify *main()* so that the rwlock is initialized instead of a mutex. We'll use NULL again as the *attr* argument to *pthread_rwlock_init()* because we want to use the default attributes for the rwlock.

Here's the updated code, with the changes in bold:

```
#include <stdio.h>
#include <stdlib.h>
```

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

typedef struct {
   int a;
   int b;
   int result;
   int result2;
   int use_count;
   int use_count2;
   int max_use;
   int max_use2;
   pthread_rwlock_t rwl;
} app_data;

void *user_thread(void *data)
{
   int uses = 0;
   app_data *td = (app_data*)data;

   while(uses < td->max_use) {
      pthread_rwlock_rdlock(&td->rwl);
      if (td->a == 5) {
         td->result += (td->a + td->b);
         td->use_count++;
         uses++;
      }
      pthread_rwlock_unlock(&td->rwl);
      usleep(1);
   }
   return 0;
}

void *changer_thread(void *data)
{
   app_data *td = (app_data*)data;

   while ((td->use_count + td->use_count2) < (td->max_use + td->max_use2)) {
      pthread_rwlock_wrlock(&td->rwl);
      if (td->a == 5) {
         td->a = 50;
         td->b = td->a + usleep(1000);
      } else {
         td->a = 5;
         td->b = td->a + usleep(1000);
      }
      pthread_rwlock_unlock(&td->rwl);
```


© 2020, QNX Software Systems Limited

```
      usleep(1);
   }
   return 0;
}

void *subtracter_thread(void *data)
{
   int use=0;
   app_data *td=(app_data*)data;

   while(use < td->max_use2) {
      pthread_rwlock_rdlock(&td->rwl);
      if (td->a == 50) {
         td->result2 -= (td->a + td->b);
         use++;
         td->use_count2++;
      }
      pthread_rwlock_unlock(&td->rwl);
      usleep(1);
   }
   return 0;
}

int main(int argc, char **argv)
{
   pthread_t ct, ut, st;
   app_data td = {5,5,0,0,0,0,100,100};
   void *retval;

   pthread_rwlock_init(&td.rwl, NULL);
   pthread_create(&ut, NULL, user_thread, &td);
   pthread_create(&ct, NULL, changer_thread, &td);
   pthread_create(&st, NULL, subtracter_thread, &td);
   pthread_join(ct, &retval);
   pthread_join(ut, &retval);
   pthread_join(st, &retval);
   pthread_rwlock_destroy(&td.rwl);
   printf("result should be %d; it's %d\n", td.max_use * (5 + 5), td.result);
   printf("result2 should be %d; it's %d\n", -(td.max_use2 * (50 + 50)),
          td.result2);

   return EXIT_SUCCESS;
}
```

Now *subtracter_thread()* and *user_thread()* can read the data at the same time, but *changer_thread()* can't access the data at the same time as either *subtracter_thread()* or *user_thread()*.

Scheduling and the real world

So far we've talked about scheduling policies and thread states, but we haven't said much yet about *why* and *when* things are rescheduled.

There's a common misconception that rescheduling just "occurs," without any real causes. Actually, this is a useful abstraction during design! But it's important to understand the conditions that cause rescheduling. Recall the diagram "Scheduling roadmap" (in "*The rules*" in the section on the kernel's role).

Rescheduling occurs *only* because of:

- a hardware interrupt
- a kernel call
- a fault

Rescheduling: hardware interrupts

Rescheduling due to a hardware interrupt occurs in two cases:

- timers
- other hardware

The realtime clock generates periodic interrupts for the kernel, causing time-based rescheduling.

For example, if you issue a `sleep (10);` call, a number of realtime clock interrupts will occur; the kernel increments the time-of-day clock at each interrupt. When the time-of-day clock indicates that 10 seconds have elapsed, the kernel reschedules your thread as READY. (This is discussed in more detail in the Clocks, Timers, and Getting a Kick Every So Often chapter.)

Other threads might wait for hardware interrupts from peripherals, such as the serial port, a hard disk, or an audio card. In this case, they are blocked in the kernel waiting for a hardware interrupt; the thread will be rescheduled by the kernel only after that "event" is generated.

Rescheduling: kernel calls

If the rescheduling is caused by a thread issuing a kernel call, the rescheduling is done immediately and can be considered asynchronous to the timer and other interrupts.

For example, above we called `sleep(10);`. This C library function is eventually translated into a kernel call. At that point, the kernel made a rescheduling decision to move your thread from the RUNNING state, and then schedule another thread that was READY.

There are many kernel calls that cause a process to be rescheduled. Most of them are fairly obvious. Here are a few:

- timer functions (e.g., *sleep()*)
- messaging functions (e.g., *MsgSendv()*)
- thread primitives, (e.g., *pthread_cancel()*, *pthread_join()*)

© 2020, QNX Software Systems Limited

Rescheduling: exceptions

The final cause of rescheduling, a CPU fault, is an *exception*, somewhere between a hardware interrupt and a kernel call.

It operates asynchronously to the kernel (like an interrupt) but operates synchronously with the user code that caused it (like a kernel call—for example, a divide-by-zero exception). The same discussion as above (for hardware interrupts and kernel calls) applies to faults.

Summary

QNX Neutrino offers a rich set of scheduling options with threads, the primary scheduling elements. Processes are defined as a unit of resource ownership (e.g., a memory area) and contain one or more threads.

Threads can use any of the following synchronization methods:

- mutexes—allow only one thread to own the mutex at a given point in time.
- semaphores—allow a fixed number of threads to “own” the semaphore.
- sleepons—allow a number of threads to block on a number of objects, while allocating the underlying condvars dynamically to the blocked threads.
- condvars—similar to sleepons except that the allocation of the condvars is controlled by the programmer.
- joining—allows a thread to synchronize to the termination of another thread.
- barriers—allows threads to wait until a number of threads have reached the synchronization point.

Note that mutexes, semaphores, and condition variables can be used between threads in the same or different processes, but that sleepons can be used only between threads in the same process (because the library has a mutex “hidden” in the process's address space).

As well as synchronization, threads can be scheduled (using a priority and a scheduling algorithm), and they'll automatically run on a single-processor box or an SMP box.

Whenever we talk about creating a “process” (mainly as a means of porting code from single-threaded implementations), we're really creating an address space with one thread running in it—that thread starts at *main()* or at *fork()*, depending on the function called.

Chapter 2
Message Passing

In this chapter, we'll look at the most distinctive feature of QNX Neutrino, *message passing*. Message passing lies at the heart of the operating system's microkernel architecture, giving the OS its modularity.

A small microkernel and message passing

One of the principal advantages of QNX Neutrino is that it's scalable. By “scalable” I mean that it can be tailored to work on tiny embedded boxes with tight memory constraints, right up to large networks of multiprocessor SMP boxes with almost unlimited memory.

QNX Neutrino achieves its scalability by making each service-providing component *modular*. This way, you can include only the components you need in the final system. By using threads in the design, you'll also help to make it scalable to SMP systems (we'll see some more uses for threads in this chapter).

This is the philosophy that was used during the initial design of the QNX family of operating systems and has been carried through to this day. The key is a small *microkernel* architecture, with modules that would traditionally be incorporated into a monolithic kernel as optional components.

Figure 17: QNX Neutrino's modular architecture.

You, the system architect, decide which modules you want. Do you need a filesystem in your project? If so, then add one. If you don't need one, then don't bother including one. Do you need a serial port driver? Whether the answer is yes or no, this doesn't affect (nor is it affected by) your previous decision about the filesystem.

At run time, you can decide which system components are included in the running system. You can dynamically remove components from a live system and reinstall them, or others, at some other time. Is there anything special about these “drivers”? Nope, they're just regular, user-level programs that happen to perform a specific job with the hardware. In fact, we'll see how to write them in the Resource Managers chapter.

The key to accomplishing this is message passing. Instead of having the OS modules bound directly into the kernel, and having some kind of “special” arrangement with the kernel, under QNX Neutrino the modules communicate via message passing among themselves. The kernel is basically responsible only for thread-level services (e.g., scheduling). In fact, message passing isn't used just for this installation and deinstallation trick—it's the fundamental building block for almost all other services (for example, memory allocation is performed by a message to the process manager). Of course, some services are provided by direct kernel calls.

Consider opening a file and writing a block of data to it. This is accomplished by a number of messages sent from the application to an installable component of QNX Neutrino called the filesystem. The message tells the filesystem to open a file, and then another message tells it to write some data (and

© 2020, QNX Software Systems Limited

contains that data). Don't worry though—the QNX Neutrino operating system performs message passing *very* quickly.

Message passing and client/server

Imagine an application reading data from the filesystem. In QNX lingo, the application is a *client* requesting the data from a *server*.

This client/server model introduces several process states associated with message passing (we talked about these in the *Processes and Threads* chapter). Initially, the server is waiting for a message to arrive from somewhere. At this point, the server is said to be *receive-blocked* (also known as the RECEIVE state). Here's some sample `pidin` output:

```
 pid     tid name                 prio STATE       Blocked
       4   1 devc-pty              10r RECEIVE     1
```

In the above sample, the pseudo-tty server (called `devc-pty`) is process ID 4, has one thread (thread ID 1), is running at priority 10 Round-Robin, and is receive-blocked, waiting for a message from channel ID 1 (we'll see all about “channels” shortly).

Figure 18: State transitions of server.

When a message is received, the server goes into the READY state, and is capable of running. If it happens to be the highest-priority READY process, it gets the CPU and can perform some processing. Since it's a server, it looks at the message it just got and decides what to do about it. At some point, the server will complete whatever job the message told it to do, and then will “reply” to the client.

Let's switch over to the client. Initially the client was running along, consuming CPU, until it decided to send a message. The client changed from READY to either *send-blocked* or *reply-blocked*, depending on the state of the server that it sent a message to.

Figure 19: State transitions of clients.

Generally, you'll see the reply-blocked state much more often than the send-blocked state. That's because the reply-blocked state means:

> The server has received the message and is now processing it. At some point, the server will complete processing and will reply to the client. The client is blocked waiting for this reply.

Contrast that with the send-blocked state:

© 2020, QNX Software Systems Limited

> The server hasn't yet received the message, most likely because it was busy handling another message first. When the server gets around to "receiving" your (client) message, then you'll go from the send-blocked state to the reply-blocked state.

In practice, if you see a process that is send-blocked it means one of two things:

1. You happened to take a snapshot of the system in a situation where the server was busy servicing a client, and a new request arrived for that server.

   This is a normal situation; you can verify it by running `pidin` again to get a new snapshot. This time you'll probably see that the process is no longer send-blocked.

2. The server has encountered a bug and for whatever reason isn't listening to requests anymore.

   When this happens, you'll see many processes that are send-blocked on one server. To verify this, run `pidin` again, observing that there's no change in the blocked state of the client processes.

Here's a sample showing a reply-blocked client and the server it's blocked on:

```
    pid tid name                          prio STATE       Blocked
      1   1 to/x86/sys/procnto-smp-instr   0f READY
      1   2 to/x86/sys/procnto-smp-instr  10r RECEIVE     1
      1   3 to/x86/sys/procnto-smp-instr  10r NANOSLEEP
      1   4 to/x86/sys/procnto-smp-instr  10r RUNNING
      1   5 to/x86/sys/procnto-smp-instr  15r RECEIVE     1
  16426   1 esh                           10r REPLY       1
```

This shows that the program `esh` (the embedded shell) has sent a message to process number 1 (the kernel and process manager, `procnto-smp-instr`) and is now waiting for a reply.

Now you know the basics of message passing in a client/server architecture.

So now you might be thinking, "Do I have to write special QNX Neutrino message-passing calls just to open a file or write some data?!?"

You don't have to write *any* message-passing functions, unless you want to get "under the hood" (which I'll talk about a little later). In fact, let me show you some client code that does message passing:

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

int
main (void)
{
    int     fd;

    fd = open ("filename", O_WRONLY);
    write (fd, "This is message passing\n", 24);
    close (fd);
```

```
    return (EXIT_SUCCESS);
}
```

See? Standard C code, nothing tricky.

The message passing is done by the QNX Neutrino C library. You simply issue standard POSIX 1003.1 or ANSI C function calls, and the C library does the message-passing work for you.

In the above example, we saw three functions being called and three distinct messages being sent:

- *open()* sent an “open” message
- *write()* sent a “write” message
- *close()* sent a “close” message

We'll be discussing the messages themselves in a lot more detail when we look at resource managers (in the *Resource Managers* chapter), but for now all you need to know is the fact that different types of messages were sent.

Let's step back for a moment and contrast this to the way the example would have worked in a traditional operating system.

The client code would remain the same and the differences would be hidden by the C library provided by the vendor. On such a system, the *open()* function call would invoke a kernel function, which would then call directly into the filesystem, which would execute some code, and return a file descriptor. The *write()* and *close()* calls would do the same thing.

So? Is there an advantage to doing things this way? Keep reading!

© 2020, QNX Software Systems Limited

Network-distributed message passing

Suppose we want to change our example above to talk to a different node on the network. You might think that we'll have to invoke special function calls to “get networked.” Here's the network version's code:

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

int
main (void)
{
    int     fd;

    fd = open ("/net/wintermute/home/rk/filename", O_WRONLY);
    write (fd, "This is message passing\n", 24);
    close (fd);

    return (EXIT_SUCCESS);
}
```

You're right if you think the code is almost the same in both versions. It is.

In a traditional OS, the C library *open()* calls into the kernel, which looks at the filename and says “oops, this is on a different node.” The kernel then calls into the network filesystem (NFS) code, which figures out where **/net/wintermute/home/rk/filename** actually is. Then, NFS calls into the network driver and *sends a message* to the kernel on node **wintermute**, which then repeats the process that we described in our original example. Note that in this case, there are really two filesystems involved; one is the NFS client filesystem, and one is the remote filesystem. Unfortunately, depending on the implementation of the remote filesystem and NFS, certain operations may not work as expected (e.g., file locking) due to incompatibilities.

Under QNX Neutrino, the C library *open()* creates the *same* message that it would have sent to the local filesystem and sends it to the filesystem on node **wintermute**. In the local and remote cases, the *exact* same filesystem is used.

This is another fundamental characteristic of QNX Neutrino: network-distributed operations are essentially “free,” as the work to decouple the functionality requirements of the clients from the services provided by the servers is already done, by virtue of message passing.

On a traditional kernel there's a “double standard” where local services are implemented one way, and remote (network) services are implemented in a totally different way.

What it means for you

Message passing is elegant and network-distributed. So what? What does it buy you, the programmer?

Well, it means that your programs inherit those characteristics—they too can become network-distributed with far less work than on other systems. But the benefit that I find most useful is that they let you test software in a nice, modular manner.

You've probably worked on large projects where many people have to provide different pieces of the software. Of course, some of these people are done sooner or later than others.

These projects often have problems at two stages: initially at project definition time, when it's hard to decide where one person's development effort ends and another's begins, and then at testing/integration time, when it isn't possible to do full systems integration testing because all the pieces aren't available.

With message passing, the individual components of a project can be decoupled very easily, leading to a very simple design and reasonably simple testing. If you want to think about this in terms of existing paradigms, it's very similar to the concepts used in Object Oriented Programming (OOP).

What this boils down to is that testing can be performed on a piece-by-piece basis. You can set up a simple program that sends messages to your server process, and since the inputs and outputs of that server process are (or should be!) well documented, you can determine if that process is functioning. Heck, these test cases can even be automated and placed in a regression suite that runs periodically!

The philosophy of QNX Neutrino

Message passing is at the heart of the philosophy of QNX Neutrino. Understanding the uses and implications of message passing will be the key to making effective use of the OS. Before we go into the details, let's look at a little bit of theory first.

© 2020, QNX Software Systems Limited

Multiple threads

Although the client/server model is easy to understand, and the most commonly used, there are two other variations on the theme. The first is the use of multiple threads (the topic of this section), and the second is a model called *server/subserver* that's sometimes useful for general design, but really shines in network-distributed designs. The combination of the two can be extremely powerful, especially on a network of SMP boxes!

As we discussed in the *Processes and Threads* chapter, QNX Neutrino has the ability to run multiple threads of execution in the same process. How can we use this to our advantage when we combine this with message passing?

The answer is fairly simple. We can start a *pool of threads* (using the *thread_pool_*()* functions that we talked about in the *Processes and Threads* chapter), each of which can handle a message from a client:

Figure 20: Clients accessing threads in a server.

This way, when a client sends us a message, we really don't care *which* thread gets it, as long as the work gets done. This has a number of advantages. The ability to service multiple clients with multiple threads, versus servicing multiple clients with just one thread, is a powerful concept. The main advantage is that the kernel can multitask the server among the various clients, without the server itself having to perform the multitasking.

On a single-processor machine, having a bunch of threads running means that they're all competing with each other for CPU time.

But, on an SMP box, we can have multiple threads competing for multiple CPUs, while sharing the same data area across those multiple CPUs. This means that we're limited only by the number of available CPUs on that particular machine.

Server/subserver

Let's now look at the server/subserver model, and then we'll combine it with the multiple threads model.

In this model, a server still provides a service to clients, but because these requests may take a long time to complete, we need to be able to start a request and still be able to handle new requests as they arrive from other clients.

If we tried to do this with the traditional single-threaded client/server model, once one request was received and started, we wouldn't be able to receive any more requests unless we periodically stopped what we were doing, took a quick peek to see if there were any other requests pending, put those on

a work queue, and then continued on, distributing our attention over the various jobs in the work queue. Not very efficient. You're practically duplicating the work of the kernel by “time slicing” between multiple jobs!

Imagine what this would look like if *you* were doing it. You're at your desk, and someone walks up to you with a folder full of work. You start working on it. As you're busy working, you notice that someone else is standing in the doorway of your cubicle with more work of equally high priority (of course)! Now you've got two piles of work on your desk. You're spending a few minutes on one pile, switching over to the other pile, and so on, all the while looking at your doorway to see if someone else is coming around with even more work.

The server/subserver model would make a lot more sense here. In this model, we have a server that creates several other processes (the subservers). These subservers each send a message to the server, but the server doesn't reply to them until it gets a request from a client. Then it passes the client's request to one of the subservers by replying to it with the job that it should perform. The following diagram illustrates this. Note the direction of the arrows—they indicate the direction of the sends!

Figure 21: Server/subserver model.

If you were doing a job like this, you'd start by hiring some extra employees. These employees would all come to you (just as the subservers send a message to the server—hence the note about the arrows in the diagram above), looking for work to do. Initially, you might not have any, so you wouldn't reply to their query. When someone comes into your office with a folder full of work, you say to one of your employees, “Here's some work for you to do.” That employee then goes off and does the work. As other jobs come in, you'd delegate them to the other employees.

The trick to this model is that it's *reply-driven*—the work starts when you *reply* to your subservers. The standard client/server model is *send-driven* because the work starts when you send the server a message.

So why would the clients march into *your* office, and not the offices of the employees that you hired? Why are you “arbitrating” the work? The answer is fairly simple: you're the coordinator responsible for performing a particular task. It's up to you to ensure that the work is done. The clients that come to you with their work know you, but they don't know the names or locations of your (perhaps temporary) employees.

As you probably suspected, you can certainly mix multithreaded servers with the server/subserver model. The main trick is going to be determining which parts of the “problem” are best suited to being distributed over a network (generally those parts that won't use up the network bandwidth too much) and which parts are best suited to being distributed over the SMP architecture (generally those parts that want to use common data areas).

So why would we use one over the other? Using the server/subserver approach, we can distribute the work over multiple machines on a network. This effectively means that we're limited only by the number

© 2020, QNX Software Systems Limited

of available machines on the network (and network bandwidth, of course). Combining this with multiple threads on a bunch of SMP boxes distributed over a network yields “clusters of computing,” where the central “arbitrator” delegates work (via the server/subserver model) to the SMP boxes on the network.

Some examples

Now we'll consider a few examples of each method.

Send-driven (client/server)

Filesystems, serial ports, consoles, and sound cards all use the client/server model. A C language application program takes on the role of the client and sends requests to these servers. The servers perform whatever work was specified, and reply with the answer.

Some of these traditional “client/server” servers may in fact actually be reply-driven (server/subserver) servers! This is because, to the ultimate client, they *appear* as a standard server, even though the server itself uses server/subserver methods to get the work done. What I mean by that is, the client still sends a message to what it thinks is the “service providing process.” What actually happens is that the “service providing process” simply delegates the client's work to a different process (the subserver).

Reply-driven (server/subserver)

One of the more popular reply-driven programs is a fractal graphics program distributed over the network. The master program divides the screen into several areas, for example, 64 regions. At startup, the master program is given a list of nodes that can participate in this activity. The master program starts up worker (subserver) programs, one on each of the nodes, and then waits for the worker programs to send to the master.

The master then repeatedly picks “unfilled” regions (of the 64 on screen) and delegates the fractal computation work to the worker program on another node by replying to it. When the worker program has completed the calculations, it sends the results back to the master, which displays the result on the screen.

Because the worker program sent to the master, it's now up to the master to again reply with more work. The master continues doing this until all 64 areas on the screen have been filled.

An important subtlety

Because the master program is delegating work to worker programs, the master program can't afford to become blocked on any one program! In a traditional send-driven approach, you'd expect the master to create a program and then send to it. Unfortunately, the master program wouldn't be replied to until the worker program was done, meaning that the master program couldn't send simultaneously to another worker program, effectively negating the advantages of having multiple worker nodes.

Figure 22: One master, multiple workers.

The solution to this problem is to have the worker programs start up, and *ask* the master program if there's any work to do by sending it a message. Once again, we've used the direction of the arrows in the diagram to indicate the direction of the send. Now the worker programs are waiting for the master to reply. When something tells the master program to do some work, it replies to one or more of the workers, which causes them to go off and do the work. This lets the workers go about their business; the master program can still respond to new requests (it's not blocked waiting for a reply from one of the workers).

Multithreaded server

Multithreaded servers are indistinguishable from single-threaded servers from the client's point of view. In fact, the designer of a server can just “turn on” multithreading by starting another thread.

In any event, the server can still make use of multiple CPUs in an SMP configuration, even if it is servicing only one “client.” What does that mean? Let's revisit the fractal graphics example. When a subserver gets a request from the server to “compute,” there's absolutely nothing stopping the subserver from starting up multiple threads on multiple CPUs to service the one request. In fact, to make the application scale better across networks that have some SMP boxes and some single-CPU boxes, the server and subserver can initially exchange a message whereby the subserver tells the server how many CPUs it has—this lets it know how many requests it can service simultaneously. The server would then queue up more requests for SMP boxes, allowing the SMP boxes to do more work than single-CPU boxes.

© 2020, QNX Software Systems Limited

Using message passing

Now that we've seen the basic concepts involved in message passing, and learned that even common everyday things like the C library use it, let's take a look at some of the details.

Architecture & structure

We've been talking about “clients” and “servers.” I've also used three key phrases:

- “The client *sends* to the server.”
- “The server *receives* from the client.”
- “The server *replies* to the client.”

I specifically used those phrases because they closely reflect the actual function names used in QNX Neutrino message-passing operations.

Here's the complete list of functions dealing with message passing available under QNX Neutrino (in alphabetical order):

- *ChannelCreate()*, *ChannelDestroy()*
- *ConnectAttach()*, *ConnectDetach()*
- *MsgDeliverEvent()*
- *MsgError()*
- *MsgRead()*, *MsgReadv()*
- *MsgReceive()*, *MsgReceivePulse()*, *MsgReceivev()*
- *MsgReply()*, *MsgReplyv()*
- *MsgSend()*, *MsgSendnc()*, *MsgSendsv()*, *MsgSendsvnc()*, *MsgSendv()*, *MsgSendvnc()*, *MsgSendvs()*, *MsgSendvsnc()*
- *MsgWrite()*, *MsgWritev()*

Don't let this list overwhelm you! You can write perfectly useful client/server applications using just a small subset of the calls from the list—as you get used to the ideas, you'll see that some of the other functions can be very useful in certain cases.

A useful minimal set of functions is *ChannelCreate()*, *ConnectAttach()*, *MsgReply()*, *MsgSend()*, and *MsgReceive()*.

We'll break our discussion up into the functions that apply on the client side, and those that apply on the server side.

The client

The client wants to send a request to a server, block until the server has completed the request, and then when the request is completed and the client is unblocked, to get at the "answer."

This implies two things: the client needs to be able to establish a connection to the server and then to transfer data via messages—a message from the client to the server (the "send" message) and a message back from the server to the client (the "reply" message, the server's reply).

Establishing a connection

So, let's look at these functions in turn. The first thing we need to do is to establish a connection. We do this with the function *ConnectAttach()*, which looks like this:

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

int ConnectAttach (int nd,
                   pid_t pid,
                   int chid,
                   unsigned index,
                   int flags);
```

ConnectAttach() is given three identifiers: the *nd*, which is the Node Descriptor, the *pid*, which is the process ID, and the *chid*, which is the channel ID. These three IDs, commonly referred to as "ND/PID/CHID," uniquely identify the server that the client wants to connect to. We'll ignore the *index* and *flags* (just set them to 0).

So, let's assume that we want to connect to process ID 77, channel ID 1 on our node. Here's the code sample to do that:

```
int coid;

coid = ConnectAttach (0, 77, 1, 0, 0);
```

As you can see, by specifying a *nd* of zero, we're telling the kernel that we wish to make a connection on our node.

How did I figure out I wanted to talk to process ID 77 and channel ID 1? We'll see that shortly (see "*Finding the server's ND/PID/CHID,*" below).

At this point, I have a *connection ID*, a small integer that uniquely identifies a connection from my client to a specific channel on a particular server.

I can use this connection ID when sending to the server as many times as I like. When I'm done with it, I can destroy it via:

```
ConnectDetach (coid);
```

So let's see how I actually use it.

© 2020, QNX Software Systems Limited

Sending messages

Message passing on the client is achieved using some variant of the *MsgSend*()* function family. We'll look at the simplest member, *MsgSend()*:

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

long MsgSend (int coid,
              const void *smsg,
              size_t sbytes,
              void *rmsg,
              size_t rbytes);
```

MsgSend()'s arguments are:

- the connection ID of the target server (*coid*),
- a pointer to the send message (*smsg*),
- the size of the send message (*sbytes*),
- a pointer to the reply message (*rmsg*), and
- the size of the reply message (*rbytes*).

It couldn't get any simpler than that!

Let's send a simple message to process ID 77, channel ID 1:

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

char *smsg = "This is the outgoing buffer";
char rmsg [200];
int  coid;

// establish a connection
coid = ConnectAttach (0, 77, 1, 0, 0);
if (coid == -1) {
    fprintf (stderr, "Couldn't ConnectAttach to 0/77/1!\n");
    perror (NULL);
    exit (EXIT_FAILURE);
}

// send the message
if (MsgSend (coid,
             smsg,
             strlen (smsg) + 1,
             rmsg,
             sizeof (rmsg)) == -1) {
    fprintf (stderr, "Error during MsgSend\n");
    perror (NULL);
```

```
    exit (EXIT_FAILURE);
}

if (strlen (rmsg) > 0) {
    printf ("Process ID 77 returns \"%s\"\n", rmsg);
}
```

Let's assume that process ID 77 was an active server expecting that particular format of message on its channel ID 1. After the server received the message, it would process it and at some point reply with a result. At that point, the *MsgSend()* would return a 0 indicating that everything went well. If the server sends us any data in the reply, we'd print it with the last line of code (we're assuming we're getting NUL-terminated ASCII data back).

The server

Now that we've seen the client, let's look at the server. The client used *ConnectAttach()* to create a connection to a server, and then used *MsgSend()* for all its message passing.

Creating the channel

This implies that the server has to create a channel—this is the thing that the client connected to when it issued the *ConnectAttach()* function call. Once the channel has been created, the server usually leaves it up forever.

The channel gets created via the *ChannelCreate()* function, and destroyed via the *ChannelDestroy()* function:

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

int ChannelCreate  (unsigned flags);

int ChannelDestroy (int chid);
```

We'll come back to the *flags* argument later (in the "*Channel flags*" section, below). For now, let's just use a 0. Therefore, to create a channel, the server issues:

```
int  chid;

chid = ChannelCreate (0);
```

So we have a channel. At this point, clients could connect (via *ConnectAttach()*) to this channel and start sending messages:

© 2020, QNX Software Systems Limited

Figure 23: Relationship between a server channel and a client connection.

Message handling

As far as the message-passing aspects are concerned, the server handles message passing in two stages; a “receive” stage and a “reply” stage:

Figure 24: Relationship of client and server message-passing functions.

We'll look initially at two simple versions of these functions, *MsgReceive()* and *MsgReply()*, and then later see some of the variants.

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

int MsgReceive (int chid,
                void *rmsg,
                size_t rbytes,
                struct _msg_info *info);

int MsgReply (int rcvid,
              int status,
              const void *msg,
              int nbytes);
```

Let's look at how the parameters relate:

Figure 25: Message data flow.

As you can see from the diagram, there are four things we need to talk about:

1. The client issues a *MsgSend()* and specifies its transmit buffer (the *smsg* pointer and the *sbytes* length). This gets transferred into the buffer provided by the server's *MsgReceive()* function, at *rmsg* for *rbytes* in length. The client is now blocked.
2. The server's *MsgReceive()* function unblocks, and returns with a *rcvid*, which the server will use later for the reply. At this point, the data is available for the server to use.
3. The server has completed the processing of the message, and now uses the *rcvid* it got from the *MsgReceive()* by passing it to the *MsgReply()*. Note that the *MsgReply()* function takes a buffer (*smsg*) with a defined size (*sbytes*) as the location of the data to transmit to the client. The data is now transferred by the kernel.
4. Finally, the *sts* parameter is transferred by the kernel, and shows up as the return value from the client's *MsgSend()*. The client now unblocks.

You may have noticed that there are two sizes for every buffer transfer (in the client send case, there's *sbytes* on the client side and *rbytes* on the server side; in the server reply case, there's *sbytes* on the server side and *rbytes* on the client side.) The two sets of sizes are present so that the programmers of each component can specify the sizes of *their* buffers. This is done for added safety.

In our example, the *MsgSend()* buffer's size was the same as the message string's length. Let's look at the server and see how the size is used there.

Server framework

Here's the overall structure of a server:

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

...

void
server (void)
{
    int     rcvid;          // indicates who we should reply to
    int     chid;           // the channel ID
    char    message [512]; // big enough for our purposes
```


© 2020, QNX Software Systems Limited

```
    // create a channel
    chid = ChannelCreate (0);

    // this is typical of a server:  it runs forever
    while (1) {

        // get the message, and print it
        rcvid = MsgReceive (chid, message, sizeof (message),
                            NULL);
        printf ("Got a message, rcvid is %X\n", rcvid);
        printf ("Message was \"%s\".\n", message);

        // now, prepare the reply.  We reuse "message"
        strcpy (message, "This is the reply");
        MsgReply (rcvid, EOK, message, sizeof (message));
    }
}
```

As you can see, *MsgReceive()* tells the kernel that it can handle messages up to `sizeof (message)` (or 512 bytes). Our sample client (*above*) sent only 28 bytes (the length of the string). The following diagram illustrates:

Figure 26: Transferring less data than expected.

The kernel transfers the *minimum* specified by both sizes. In our case, the kernel would transfer 28 bytes. The server would be unblocked and display the client's message. The remaining 484 bytes (of the 512 byte buffer) will remain unaffected.

We run into the same situation again with *MsgReply()*. The *MsgReply()* function says that it wants to transfer 512 bytes, but our client's *MsgSend()* function has specified that a maximum of 200 bytes can be transferred. So the kernel once again transfers the minimum. In this case, the 200 bytes that the client can accept limits the transfer size. (One interesting aspect here is that once the server transfers the data, if the client doesn't receive *all* of it, as in our example, there's no way to get the data back—it's gone forever.)

Keep in mind that this “trimming” operation is normal and expected behavior.

When we discuss message passing over a network, you'll see that there's a tiny “gotcha” with the amount of data transferred. We'll see this in “*Networked message-passing differences,*” below.

Identifying messages

While the message-passing APIs transfer buffers of bytes without caring about the content, the client and server almost always care.

It's usually important to label the content, especially as most servers may support doing more than one operation (e.g., input and output). Data transmitted from a client to a server is generally identified with a *message type*, usually in the first few bytes in the message. The client includes the type in its messages, and the server examines the type to determine what action to take when it receives the message.

QNX Neutrino system messages (e.g., messages generated by standard C functions such as *fork()*, *read()*, or *open()*) start with a 16-bit unsigned integer (`uint16_t`) message type. You should use this message type to differentiate between your messages and system messages. System messages use values in the range from 0 through 511 (_IO_MAX, defined in **<sys/iomsg.h>**), so your messages should use types that are outside that range, that is, greater than 511. If different programs in your system define message types, you should use ranges that don't overlap.

A server should look at the message type in all incoming messages. If it recognizes the type, it should handle the message appropriately. If it doesn't recognize a message type or doesn't handle that type of message, it should reject the message by returning an error to the client with *MsgError()*. The expected *errno* value in this case is ENOSYS. It's also often a good idea to log a warning or error message in this case.

For example, your server might do this:

```
rcvid = MsgReceive(chid, &msg, sizeof(msg), NULL);

switch( msg.type ) {
case MY_MSG_1:
        handle_my_msg_1();
        break();
case MY_MSG_2:
        handle_my_msg_2();
        break;
default:
        printf("Unexpected message of type %d from client rcvid: %x\n",
               msg.type, rcvid );
        MsgError( rcvid, ENOSYS );
        break;
}
```


© 2020, QNX Software Systems Limited

Ignoring unrecognized messages instead of rejecting them may cause unexpected and inconvenient system failures, especially (but not limited to) causing debugging tools such as `qconn` or `pidin` to hang.

The send-hierarchy

One thing that's perhaps not obvious in a message-passing environment is the need to follow a strict *send-hierarchy*. What this means is that two threads should never send messages to each other; rather, they should be organized such that each thread occupies a *level*; all sends go from one level to a higher level, never to the same or lower level. The problem with having two threads send messages to each other is that eventually you'll run into the problem of deadlock; both threads are waiting for each other to reply to their respective messages. Since the threads are blocked, they'll never get a chance to run and perform the reply, so you end up with two (or more!) hung threads.

The way to assign the levels to the threads is to put the outermost clients at the highest level, and work down from there. For example, if you have a graphical user interface that relies on some database server, and the database server in turn relies on the filesystem, and the filesystem in turn relies on a block filesystem driver, then you've got a natural hierarchy of different processes. The sends will flow from the outermost client (the graphical user interface) down to the lower servers; the replies will flow in the opposite direction.

While this certainly works in the majority of cases, you *will* encounter situations where you need to "break" the send hierarchy. This is *never* done by simply violating the send hierarchy and sending a message "against the flow," but rather by using the *MsgDeliverEvent()* function, which we'll take a look at later.

Receive IDs, channels, and other parameters

We haven't talked about the various parameters in the examples above so that we could focus just on the message passing. Now let's take a look.

More about channels

In the server example above, we saw that the server created just one channel. It could certainly have created more, but generally, servers don't do that. (The most obvious example of a server with two channels is the Transparent Distributed Processing (TDP, also known as Qnet) native network manager—definitely an *odd* piece of software!)

As it turns out, there really isn't much need to create multiple channels in the real world. The main purpose of a channel is to give the server a well-defined place to "listen" for messages, and to give the clients a well-defined place to send their messages (via a connection). About the only time that you'd have multiple channels in a server is if the server wanted to provide either different services, or different classes of services, depending on which channel the message arrived on. The second channel could be used, for example, as a place to drop wake up pulses—this ensures that they're treated as a different "class" of service than messages arriving on the first channel.

In a previous paragraph, I had said that you could have a pool of threads running in a server, ready to accept messages from clients, and that it didn't really matter which thread got the request. This is

another aspect of the channel abstraction. Under previous versions of the QNX family of operating systems (notably QNX 4), a client would target messages at a server identified by a node ID and process ID. Since QNX 4 is single-threaded, this means that there cannot be confusion about “to whom” the message is being sent. However, once you introduce threads into the picture, the design decision had to be made as to how you would address the threads (really, the “service providers”). Since threads are ephemeral, it really didn't make sense to have the client connect to a particular node ID, process ID, and *thread* ID. Also, what if that particular thread was busy? We'd have to provide some method to allow a client to select a “non-busy thread within a defined pool of service-providing threads.”

Well, that's exactly what a channel is. It's the “address” of a “pool of service-providing threads.” The implication here is that a bunch of threads can issue a *MsgReceive()* function call on a particular channel, and block, with only one thread getting a message at a time.

Who sent the message?

Often a server will need to know who sent it a message. There are a number of reasons for this:

- accounting
- access control
- context association
- class of service
- compatibility
- etc.

It would be cumbersome (and a security hole) to have the client provide this information with each and every message sent. Therefore, there's a structure filled in by the kernel whenever the *MsgReceive()* function unblocks because it got a message. This structure is of type `struct _msg_info`, and contains the following:

```
struct _msg_info
{
    uint32_t   nd;
    uint32_t   srcnd;
    pid_t      pid;
    int32_t    tid;
    int32_t    chid;
    int32_t    scoid;
    int32_t    coid;
    int16_t    priority;
    int16_t    flags;
    ssize64_t  msglen;
    ssize64_t  srcmsglen;
    ssize64_t  dstmsglen;
};
```

You pass it to the *MsgReceive()* function as the last argument. If you pass a NULL, then nothing happens. (The information can be retrieved later via the *MsgInfo()* call, so it's not gone forever!)

© 2020, QNX Software Systems Limited

Let's look at the fields:

nd*, *srcnd*, *pid*, and *tid

Node Descriptors, process ID, and thread ID of the client. (Note that *nd* is the receiving node's node descriptor for the transmitting node; *srcnd* is the transmitting node's node descriptor for the receiving node. There's a very good reason for this `:-)`, which we'll see below in "*Some notes on NDs*.")

priority

The priority of the sending thread.

chid*, *coid

Channel ID that the message was sent to, and the connection ID used.

scoid

Server Connection ID. This is an internal identifier used by the kernel to route the message from the server back to the client. You don't need to know about it, except for the interesting fact that it will be a small integer that uniquely represents the client.

flags

Contains a variety of flag bits, including the following:

- _NTO_MI_BITS_64 and _NTO_MI_BITS_DIFF tell you that the sender is using a 64-bit architecture, or a different word-size architecture than you're using.
- _NTO_MI_ENDIAN_BIG and _NTO_MI_ENDIAN_DIFF tell you about the endian-ness of the sending machine (in case the message came over the network from a machine with a different endian-ness).
- _NTO_MI_NET_CRED_DIRTY is used internally.
- We'll look at _NTO_MI_UNBLOCK_REQ in the section *"Using the _NTO_MI_UNBLOCK_REQ,"* below.

If you determine that your program is incompatible with the sender, you can return an error such as ENOTSUP. See "*Replying with no data, or an errno*" later in this chapter.

msglen

Number of bytes received.

srcmsglen

The length of the source message, in bytes, as sent by the client. This may be greater than the value in *msglen*, as would be the case when receiving less data than what was sent.

dstmsglen

The length of the client's reply buffer, in bytes.

The receive ID (a.k.a. the client cookie)

In the code sample above, notice how we:

```
rcvid = MsgReceive (...);
...
MsgReply (rcvid, ...);
```

This is a key snippet of code, because it illustrates the binding between receiving a message from a client, and then being able to (sometime later) reply to that particular client. The *receive ID* is an integer that acts as a "magic cookie" that you'll need to hold onto if you want to interact with the client later. What if you lose it? It's gone. The client will not unblock from the *MsgSend()* until you (the server) die, or if the client has a timeout on the message-passing call (and even then it's tricky; see the *TimerTimeout()* function in the QNX Neutrino *C Library Reference*, and the discussion about its use in the *Clocks, Timers, and Getting A Kick Every So Often* chapter, under "Kernel timeouts").

Don't depend on the value of the receive ID to have any particular meaning—it may change in future versions of the operating system. You can assume that it will be unique, in that you'll never have two outstanding clients identified by the same receive IDs (in that case, the kernel couldn't tell them apart either when you do the *MsgReply()*).

Also, note that except in one special case (the *MsgDeliverEvent()* function which we'll look at later), once you've done the *MsgReply()*, that particular receive ID ceases to have meaning.

This brings us to the *MsgReply()* function.

Replying to the client

MsgReply() accepts a receive ID, a status, a message pointer, and a message size. We've just finished discussing the receive ID; it identifies who the reply message should be sent to. The status variable indicates the return status that should be passed to the client's *MsgSend()* function. Finally, the message pointer and size indicate the location and size of the *optional* reply message that should be sent.

The *MsgReply()* function may appear to be very simple (and it is), but its applications require some examination.

Not replying to the client

There's absolutely no requirement that you reply to a client *before* accepting new messages from other clients via *MsgReceive()*! This can be used in a number of different scenarios.

In a typical device driver, a client may make a request that won't be serviced for a long time. For example, the client may ask an Analog-to-Digital Converter (ADC) device driver to "Go out and collect 45 seconds worth of samples." In the meantime, the ADC driver shouldn't just close up shop for 45 seconds! Other clients might wish to have requests serviced (for example, there might be multiple analog channels, or there might be status information that should be available immediately, etc.).

Architecturally, the ADC driver will simply queue the receive ID that it got from the *MsgReceive()*, start up the 45-second accumulation process, and go off and handle other requests. When the 45 seconds

© 2020, QNX Software Systems Limited

are up and the samples have been accumulated, the ADC driver can find the receive ID associated with the request and *then* reply to the client.

You'd also want to hold off replying to a client in the case of the reply-driven server/subserver model (where some of the “clients” are the subservers). Since the subservers are looking for work, you'd simply make a note of their receive IDs and store those away. When actual work arrived, then and only then would you reply to the subserver, thus indicating that it should do some work.

Replying with no data, or an *errno*

When you finally reply to the client, there's no requirement that you transfer any data. This is used in two scenarios.

You may choose to reply with no data if the sole purpose of the reply is to unblock the client. Let's say the client just wants to be blocked until some particular event occurs, but it doesn't need to know which event. In this case, no data is required by the *MsgReply()* function; the receive ID is sufficient:

```
MsgReply (rcvid, EOK, NULL, 0);
```

This unblocks the client (but doesn't return any data) and returns the EOK “success” indication.

As a slight modification of that, you may wish to return an error status to the client. In this case, you can't do that with *MsgReply()*, but instead *must* use *MsgError()*:

```
MsgError (rcvid, EROFS);
```

In the above example, the server detects that the client is attempting to write to a read-only filesystem, and, instead of returning any actual data, simply returns an *errno* of EROFS back to the client.

Alternatively (and we'll look at the calls shortly), you may have already transferred the data (via *MsgWrite()*), and there's no additional data to transfer.

Why the two calls? They're subtly different. While both *MsgError()* and *MsgReply()* will unblock the client, *MsgError()* will *not* transfer any additional data, *will* cause the client's *MsgSend()* function to return -1, and *will* cause the client to have *errno* set to whatever was passed as the second argument to *MsgError()*.

On the other hand, *MsgReply() could* transfer data (as indicated by the third and fourth arguments), and *will* cause the client's *MsgSend()* function to return whatever was passed as the second argument to *MsgReply()*. *MsgReply()* has *no* effect on the client's *errno*.

Generally, if you're returning only a pass/fail indication (and no data), you'd use *MsgError()*, whereas if you're returning data, you'd use *MsgReply()*. Traditionally, when you *do* return data, the second argument to *MsgReply()* will be a positive integer indicating the number of bytes being returned.

Finding the server's ND/PID/CHID

You've noticed that in the *ConnectAttach()* function, we require a Node Descriptor (ND), a process ID (PID), and a channel ID (CHID) in order to be able to attach to a server. So far we haven't talked about how the client finds this ND/PID/CHID information.

If one process creates the other, then it's easy—the process creation call returns with the process ID of the newly created process. Either the creating process can pass its own PID and CHID on the

command line to the newly created process or the newly created process can issue the *getppid()* function call to get the PID of its parent and assume a "well-known" CHID.

What if we have two perfect strangers? This would be the case if, for example, a third party created a server and an application that you wrote wanted to talk to that server. The real issue is, "How does a server *advertise* its location?"

There are many ways of doing this; we'll look at four of them, in increasing order of programming "elegance":

1. Open a well-known filename and store the ND/PID/CHID there. This is the traditional approach taken by UNIX-style servers, where they open a file (for example, **/etc/httpd.pid**), write their process ID there as an ASCII string, and expect that clients will open the file and fetch the process ID.
2. Use global variables to advertise the ND/PID/CHID information. This is typically used in multithreaded servers that need to send themselves messages, and is, by its nature, a very limited case.
3. Use the name-location functions (*name_attach()* and *name_detach()*, and then the *name_open()* and *name_close()* functions on the client side).
4. Take over a portion of the pathname space and become a resource manager. We'll talk about this when we look at resource managers in the *Resource Managers chapter*.

The first approach is very simple, but can suffer from "pathname pollution," where the **/etc** directory has all kinds of ***.pid** files in it. Since files are persistent (meaning they survive after the creating process dies and the machine reboots), there's no obvious method of cleaning up these files, except perhaps to have a "grim reaper" task that runs around seeing if these things are still valid.

There's another related problem. Since the process that created the file can die without removing the file, there's no way of knowing whether or not the process is still alive until you try to send a message to it. Worse yet, the ND/PID/CHID specified in the file may be so stale that it would have been reused by another program! The message that you send to that program will at best be rejected, and at worst may cause damage. So that approach is out.

The second approach, where we use global variables to advertise the ND/PID/CHID values, is not a general solution, as it relies on the client's being able to access the global variables. And since this requires shared memory, it certainly won't work across a network! This generally gets used in either tiny test case programs or in very special cases, but always in the context of a multithreaded program. Effectively, all that happens is that one thread in the program is the client, and another thread is the server. The server thread creates the channel and then places the channel ID into a global variable (the node ID and process ID are the same for all threads in the process, so they don't need to be advertised.) The client thread then picks up the global channel ID and performs the *ConnectAttach()* to it.

The third approach, where we use the *name_attach()* and *name_detach()* functions, works well for simple client/server situations.

The last approach, where the server becomes a resource manager, is definitely the cleanest and is the recommended general-purpose solution. The mechanics of "how" will become clear in the *Resource Managers* chapter, but for now, all you need to know is that the server registers a particular pathname as its "domain of authority," and a client performs a simple *open()* of that pathname.

© 2020, QNX Software Systems Limited

I can't emphasize this enough:

POSIX file descriptors are implemented using connection IDs; that is, a file descriptor *is* a connection ID! The beauty of this scheme is that since the file descriptor that's returned from the *open()* is the connection ID, no further work is required on the client's end to be able to use that particular connection. For example, when the client calls *read()* later, passing it the file descriptor, this translates with very little overhead into a *MsgSend()* function.

What about priorities?

What if a low-priority process and a high-priority process send a message to a server at the same time?

Messages are *always* delivered in priority order.

If two processes send a message "simultaneously," the entire message from the higher-priority process is delivered to the server first.

If both processes are at the same priority, then the messages will be delivered in time order (since there's no such thing as absolutely simultaneous on a single-processor machine—even on an SMP box there will be some ordering as the CPUs arbitrate kernel access among themselves).

We'll come back to some of the other subtleties introduced by this question when we look at priority inversions later in this chapter.

Reading and writing data

So far you've seen the basic message-passing primitives. As I mentioned earlier, these are all that you *need*. However, there are a few extra functions that make life much easier.

Let's consider an example using a client and server where we might need other functions.

The client issues a *MsgSend()* to transfer some data to the server. After the client issues the *MsgSend()* it blocks; it's now waiting for the server to reply.

An interesting thing happens on the server side. The server has called *MsgReceive()* to receive the message from the client. Depending on the design that you choose for your messages, the server may or may not know how big the client's message is. Why on earth would the server *not* know how big the message is? Consider the filesystem example that we've been using. Suppose the client does:

```
write (fd, buf, 16);
```

This works as expected if the server does a *MsgReceive()* and specifies a buffer size of, say, 1024 bytes. Since our client sent only a tiny message (28 bytes), we have no problems.

However, what if the client sends something bigger than 1024 bytes, say 1 megabyte?

```
write (fd, buf, 1000000);
```

How is the server going to gracefully handle this? We could, arbitrarily, say that the client isn't allowed to write more than *n* bytes. Then, in the client-side C library code for *write()*, we could look at this requirement and split up the write request into several requests of *n* bytes each. This is awkward.

The other problem with this example would be, “How big should *n* be?”

You can see that this approach has major disadvantages:

- All functions that use message transfer with a limited size will have to be modified in the C library so that the function packetizes the requests. This in itself can be a fair amount of work. Also, it can have unexpected side effects for multithreaded functions—what if the first part of the message from one thread gets sent, and then another thread in the client preempts the current thread and sends its own message. Where does that leave the original thread?
- All servers must now be prepared to handle the largest possible message size that may arrive. This means that all servers will have to have a data area that's big, or the library will have to break up big requests into many smaller ones, thereby impacting speed.

Luckily, this problem has a fairly simple workaround that also gives us some advantages.

Two functions, *MsgRead()* and *MsgWrite()*, are especially useful here. The important fact to keep in mind is that *the client is blocked*. This means that the client isn't going to go and change data structures while the server is trying to examine them.

In a multithreaded client, the potential exists for another thread to mess around with the data area of a client thread that's blocked on a server. This is considered a bug (bad design)—the server thread assumes that it has exclusive access to a client's data area until the server thread unblocks the client.

The *MsgRead()* function looks like this:

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

ssize_t MsgRead( int rcvid,
                 void *msg,
                 size_t nbytes,
                 size_t offset );
```

MsgRead() lets your server read data from the blocked client's address space, starting *offset* bytes from the beginning of the client-specified “send” buffer, into the buffer specified by *msg* for *nbytes*. The server doesn't block, and the client doesn't unblock. *MsgRead()* returns the number of bytes it actually read, or `-1` if there was an error.

So let's think about how we'd use this in our *write()* example. The C Library *write()* function constructs a message with a header that it sends to the filesystem server, `fs-qnx6`. The server receives a small portion of the message via *MsgReceive()*, looks at it, and decides where it's going to put the rest of the message. The `fs-qnx6` server may decide that the best place to put the data is into some cache buffers it's already allocated.

Let's track an example:

© 2020, QNX Software Systems Limited

Figure 27: The `fs-qnx6` message example, showing contiguous data view.

So, the client has decided to send 4 KB to the filesystem. (Notice how the C Library stuck a tiny header in front of the data so that the filesystem could tell just what kind of request it actually was—we'll come back to this when we look at multipart messages, and in even more detail when we look at resource managers.) The filesystem reads just enough data (the header) to figure out what kind of a message it is:

```
// part of the headers, fictionalized for example purposes
struct _io_write {
    uint16_t    type;
    uint16_t    combine_len;
    uint32_t    nbytes;
    uint32_t    xtype;
};

typedef union {
    uint16_t           type;
    struct _io_read    io_read;
    struct _io_write   io_write;
    ...
} header_t;

header_t    header;    // declare the header

rcvid = MsgReceive (chid, &header, sizeof (header), NULL);

switch (header.type) {
...
case _IO_WRITE:
    number_of_bytes = header.io_write.nbytes;
    ...
```

At this point, `fs-qnx6` knows that 4 KB are sitting in the client's address space (because the message told it in the *nbytes* member of the structure) and that it should be transferred to a cache buffer. The `fs-qnx6` server could issue:

```
MsgRead (rcvid, cache_buffer [index].data,
         cache_buffer [index].size, sizeof (header.io_write));
```

Notice that the message transfer has specified an offset of `sizeof (header.io_write)` in order to skip the write header that was added by the client's C library. We're assuming here that `cache_buffer [index].size` is actually 4096 (or more) bytes.

Similarly, for writing data to the client's address space, we have:

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

ssize_t MsgWrite (int rcvid,
                  const void *msg,
                  size_t nbytes,
                  size_t offset);
```

MsgWrite() lets your server write data to the client's address space, starting *offset* bytes from the beginning of the client-specified "receive" buffer. This function is most useful in cases where the server has limited space but the client wishes to get a lot of information from the server.

For example, with a data acquisition driver, the client may specify a 4-megabyte data area and tell the driver to grab 4 megabytes of data. The driver really shouldn't need to have a big area like this lying around just in case someone asks for a huge data transfer.

The driver might have a 128 KB area for DMA data transfers, and then message-pass it piecemeal into the client's address space using *MsgWrite()* (incrementing the *offset* by 128 KB each time, of course). Then, when the last piece of data has been written, the driver will *MsgReply()* to the client.

Figure 28: Transferring several chunks with *MsgWrite()*.

Note that *MsgWrite()* lets you write the data components at various places, and then either just wake up the client using *MsgReply()*:

```
MsgReply (rcvid, EOK, NULL, 0);
```

or wake up the client after writing a header at the *start* of the client's buffer:

```
MsgReply (rcvid, EOK, &header, sizeof (header));
```

This is a fairly elegant trick for writing unknown quantities of data, where you know how much data you wrote only when you're done writing it. If you're using this method of writing the header after the

© 2020, QNX Software Systems Limited

data's been transferred, you must remember to leave room for the header at the beginning of the client's data area!

Multipart messages

Until now, we've shown only message transfers happening from one buffer in the client's address space into another buffer in the server's address space. (And one buffer in the server's space into another buffer in the client's space during the reply.)

While this approach is good enough for most applications, it can lead to inefficiencies. Recall that our *write()* C library code took the buffer that you passed to it, and stuck a small header on the front of it. Using what we've learned so far, you'd expect that the C library would implement *write()* something like this (this isn't the real source):

```
ssize_t write (int fd, const void *buf, size_t nbytes)
{
    char        *newbuf;
    io_write_t  *wptr;
    ssize_t     nwritten;

    newbuf = malloc (nbytes + sizeof (io_write_t));

    // fill in the write_header at the beginning
    wptr = (io_write_t *) newbuf;
    wptr -> type = _IO_WRITE;
    wptr -> nbytes = nbytes;

    // store the actual data from the client
    memcpy (newbuf + sizeof (io_write_t), buf, nbytes);

    // send the message to the server
    nwritten = MsgSend (fd,
                        newbuf,
                        nbytes + sizeof (io_write_t),
                        newbuf,
                        sizeof (io_write_t));
    free (newbuf);
    return (nwritten);
}
```

See what happened? A few bad things:

- The *write()* now has to be able to *malloc()* a buffer big enough for both the client data (which can be fairly big) and the header. The size of the header isn't the issue—in this case, it was 12 bytes.
- We had to *copy* the data twice: once via the *memcpy()*, and then again during the message transfer.

- We had to establish a pointer to the `io_write_t` type and point it to the beginning of the buffer, rather than access it natively (this is a minor annoyance).

Since the kernel is going to copy the data anyway, it would be nice if we could tell it that one part of the data (the header) is located at a certain address, and that the other part (the data itself) is located somewhere else, *without* the need for us to manually assemble the buffers and to copy the data.

As luck would have it, QNX Neutrino implements a mechanism that lets us do just that! The mechanism is something called an *IOV*, standing for "Input/Output Vector."

Let's look at some code first, then we'll discuss what happens:

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

ssize_t write (int fd, const void *buf, size_t nbytes)
{
    io_write_t  whdr;
    iov_t       iov [2];

    // set up the IOV to point to both parts:
    SETIOV (iov + 0, &whdr, sizeof (whdr));
    SETIOV (iov + 1, buf, nbytes);

    // fill in the io_write_t at the beginning
    whdr.type = _IO_WRITE;
    whdr.nbytes = nbytes;

    // send the message to the server
    return (MsgSendv (coid, iov, 2, iov, 1));
}
```

First of all, notice there's no *malloc()* and no *memcpy()*. Next, notice the use of the `iov_t` type. This is a structure that contains an address and length pair, and we've allocated two of them (named *iov*).

The `iov_t` type definition is automatically included by **<sys/neutrino.h>**, and is defined as:

```
typedef struct iovec
{
    void     *iov_base;
    size_t    iov_len;
} iov_t;
```

Given this structure, we fill the address and length pairs with the write header (for the first part) and the data from the client (in the second part). There's a convenience macro called *SETIOV()* that does the assignments for us. It's formally defined as:

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

#define SETIOV(_iov, _addr, _len) \
```


© 2020, QNX Software Systems Limited

```
                ((_iov)->iov_base = (void *)(_addr), \
                 (_iov)->iov_len = (_len))
```

SETIOV() accepts an `iov_t`, and the address and length data to be stuffed into the IOV.

Also notice that since we're creating an IOV to point to the header, we can allocate the header on the stack without using *malloc()*. This can be a blessing and a curse—it's a blessing when the header is quite small, because you avoid the headaches of dynamic memory allocation, but it can be a curse when the header is huge, because it can consume a fair chunk of stack space. Generally, the headers are quite small.

In any event, the important work is done by *MsgSendv()*, which takes almost the same arguments as the *MsgSend()* function that we used in the previous example:

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

long MsgSendv (int coid,
               const iov_t *siov,
               size_t sparts,
               const iov_t *riov,
               size_t rparts);
```

Let's examine the arguments:

coid

The connection ID that we're sending to, just as with *MsgSend()*.

sparts **and** ***rparts***

The number of send and receive parts specified by the `iov_t` parameters. In our example, we set *sparts* to 2 indicating that we're sending a 2-part message, and *rparts* to 1 indicating that we're receiving a 1-part reply.

siov **and** ***riov***

The `iov_t` arrays indicate the address and length pairs that we wish to send. In the above example, we set up the 2 part *siov* to point to the header and the client data, and the 1 part *riov* to point to just the header.

This is how the kernel views the data:

Figure 29: How the kernel sees a multipart message.

The kernel just copies the data seamlessly from each part of the IOV in the client's space into the server's space (and back, for the reply). Effectively, the kernel is performing a gather-scatter operation.

A few points to keep in mind:

- The number of parts is “limited” to 524288; however, our example of 2 is typical.
- The kernel simply copies the data specified in one IOV from one address space into another.
- *The source and the target IOVs don't have to be identical.*

Why is the last point so important? To answer that, let's take a look at the big picture. On the client side, let's say we issued:

```
write (fd, buf, 12000);
```

which generated a two-part IOV of:

- header (12 bytes)
- data (12000 bytes)

On the server side, (let's say it's the filesystem, `fs-qnx6`), we have a number of 4 KB cache blocks, and we'd like to efficiently receive the message directly into the cache blocks. Ideally, we'd like to write some code like this:

```
// set up the IOV structure to receive into:
SETIOV (iov + 0, &header, sizeof (header.io_write));
SETIOV (iov + 1, &cache_buffer [37], 4096);
SETIOV (iov + 2, &cache_buffer [16], 4096);
SETIOV (iov + 3, &cache_buffer [22], 4096);
rcvid = MsgReceivev (chid, iov, 4, NULL);
```

This code does pretty much what you'd expect: it sets up a 4-part IOV structure, sets the first part of the structure to point to the header, and the next three parts to point to cache blocks 37, 16, and 22. (These numbers represent cache blocks that just happened to be available at that particular time.) Here's a graphical representation:

Figure 30: Converting contiguous data to separate buffers.

Then the *MsgReceivev()* function is called, indicating that we'll receive a message from the specified channel (the *chid* parameter) and that we're supplying a 4-part IOV structure. This also shows the IOV structure itself.

(Apart from its IOV functionality, *MsgReceivev()* operates just like *MsgReceive()*.)

© 2020, QNX Software Systems Limited

Oops! We made the same mistake as we did before, when we introduced the *MsgReceive()* function. How do we know what kind of message we're receiving, and how much data is associated with it, until we actually receive the message?

We can solve this the same way as before:

```
rcvid = MsgReceive (chid, &header, sizeof (header), NULL);
switch (header.message_type) {
…
case    _IO_WRITE:
    number_of_bytes = header.io_write.nbytes;
    // allocate / find cache buffer entries
    // fill 3-part IOV with cache buffers
    MsgReadv (rcvid, iov, 3, sizeof (header.io_write));
```

This does the initial *MsgReceive()* (note that we didn't use the IOV form for this—there's really no need to do that with a one-part message), figures out what kind of message it is, and then continues reading the data out of the client's address space (starting at offset `sizeof (header.io_write)`) into the cache buffers specified by the 3-part IOV.

Notice that we switched from using a 4-part IOV (in the first example) to a 3-part IOV. That's because in the first example, the first part of the 4-part IOV was the header, which we read directly using *MsgReceive()*, and the last three parts of the 4-part IOV are the same as the 3-part IOV—they specify where we'd like the data to go.

You can imagine how we'd perform the reply for a read request:

1. Find the cache entries that correspond to the requested data.
2. Fill an IOV structure with those entries.
3. Use *MsgWritev()* (or *MsgReplyv()*) to transfer the data to the client.

Note that if the data doesn't start right at the beginning of a cache block (or other data structure), this isn't a problem. Simply offset the first IOV to point to where the data does start, and modify the size.

What about the other versions?

All the message-passing functions except the *MsgSend*()* family have the same general form: if the function has a `v` at the end of it, it takes an IOV and a number-of-parts; otherwise, it takes a pointer and a length.

The *MsgSend*()* family has four major variations in terms of the source and destinations for the message buffers, combined with two variations of the kernel call itself.

Look at the following table:

Function	Send buffer	Receive buffer
MsgSend()	Linear	Linear
MsgSendnc()	Linear	Linear
MsgSendsv()	Linear	IOV

Function	Send buffer	Receive buffer
MsgSendsvnc()	Linear	IOV
MsgSendvs()	IOV	Linear
MsgSendvsnc()	IOV	Linear
MsgSendv()	IOV	IOV
MsgSendvnc()	IOV	IOV

By "linear," I mean a single buffer of type `void *` is passed, along with its length. The easy way to remember this is that the "`v`" stands for "vector," and is in the same place as the appropriate parameter—first or second, referring to "send" or "receive," respectively.

Hmmm... looks like the *MsgSendsv()* and *MsgSendsvnc()* functions are identical, doesn't it? Well, yes, as far as their parameters go, they indeed are. The difference lies in whether or not they are cancellation points. The "`nc`" versions are not cancellation points, whereas the non-"`nc`" versions are. (For more information about cancellation points and cancelability in general, please consult the QNX Neutrino *C Library Reference,* under *pthread_cancel()*.)

Implementation

You've probably already suspected that all the variants of the *MsgRead()*, *MsgReceive()*, *MsgSend()*, and *MsgWrite()* functions are closely related. (The only exception is *MsgReceivePulse()*—we'll look at this one shortly.)

Which ones should you use? Well, that's a bit of a philosophical debate. My own personal preference is to mix and match.

If I'm sending or receiving only one-part messages, why bother with the complexity of setting up IOVs? The tiny amount of CPU overhead in setting them up is basically the same regardless of whether you set it up yourself or let the kernel/library do it. The single-part message approach saves the kernel from having to do address space manipulations and is a little bit faster.

Should you use the IOV functions? Absolutely! Use them any time you find yourself dealing with multipart messages. *Never* copy the data when you can use a multipart message transfer with only a few lines of code. This keeps the system screaming along by minimizing the number of times data gets copied around the system; passing the pointers is much faster than copying the data into a new buffer.

© 2020, QNX Software Systems Limited

Pulses

All the messaging we've talked about so far blocks the client. It's nap time for the client as soon as it calls *MsgSend()*. The client sleeps until the server gets around to replying.

However, there are instances where the sender of a message can't afford to block. We'll look at some examples in the *Interrupts* and *Clocks, Timers, and Getting a Kick Every So Often* chapters, but for now we should understand the concept.

The mechanism that implements a non-blocking send is called a *pulse*. A pulse is a tiny message that:

- can carry 40 bits of payload (an 8-bit code and 32 bits of data)
- is non-blocking for the sender
- can be received just like any other message
- is queued if the receiver isn't blocked waiting for it.

Receiving a pulse message

Receiving a pulse is very simple: a tiny, well-defined message is presented to the *MsgReceive()*, *as if a thread had sent a normal message*. The only difference is that you can't *MsgReply()* to this message—after all, the whole idea of a pulse is that it's asynchronous. In this section, we'll take a look at another function, *MsgReceivePulse()*, that's useful for dealing with pulses.

The only “funny” thing about a pulse is that the receive ID that comes back from the *MsgReceive()* function is zero. That's your indication that this is a pulse, rather than a regular message from a client. You'll often see code in servers that looks like this:

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

    rcvid = MsgReceive (chid, …);
    if (rcvid == 0) {   // it's a pulse
        // determine the type of pulse

        // handle it
    } else {            // it's a regular message
        // determine the type of message

        // handle it
    }
```

What's in a pulse?

Okay, so you receive this message with a receive ID of zero. What does it actually look like? From the **<sys/neutrino.h>** header file, here's the definition of the `_pulse` structure:

```
struct _pulse {
    uint16_t        type;
    uint16_t        subtype;
```

```
    int8_t          code;
    uint8_t         zero [3];
    union sigval    value;
    int32_t         scoid;
};
```

Both the *type* and *subtype* members are zero (a further indication that this is a pulse). The *code* and *value* members are set to whatever the sender of the pulse determined. Generally, the *code* will be an indication of why the pulse was sent; the *value* will be a 32-bit data value associated with the pulse. Those two fields are where the "40 bits" of content comes from; the other fields aren't user adjustable.

The kernel reserves negative values of *code*, leaving 127 values for programmers to use as they see fit.

The *value* member is actually a union:

```
union sigval {
    int     sival_int;
    void    *sival_ptr;
};
```

Therefore (expanding on the server example above), you often see code like:

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

    rcvid = MsgReceive (chid, …

    if (rcvid == 0) {   // it's a pulse

        // determine the type of pulse
        switch (msg.pulse.code) {

        case    MY_PULSE_TIMER:
            // One of your timers went off, do something
            // about it...

            break;

        case    MY_PULSE_HWINT:
            // A hardware interrupt service routine sent
            // you a pulse.  There's a value in the "value"
            // member that you need to examine:

            val = msg.pulse.value.sival_int;

            // Do something about it...
```


© 2020, QNX Software Systems Limited

```
            break;

        case    _PULSE_CODE_UNBLOCK:
            // A pulse from the kernel, indicating a client
            // unblock was received, do something about it...

            break;

        // etc...

    } else {            // it's a regular message

        // determine the type of message
        // handle it

    }
```

This code assumes, of course, that you've set up your *msg* structure to contain a `struct _pulse pulse;` member, and that the manifest constants MY_PULSE_TIMER and MY_PULSE_HWINT are defined. The pulse code _PULSE_CODE_UNBLOCK is one of those negative-numbered kernel pulses mentioned above. You can find a complete list of them in **<sys/neutrino.h>** along with a brief description of the value field.

The *MsgReceivePulse()* function

The *MsgReceive()* and *MsgReceivev()* functions will receive either a "regular" message or a pulse. There may be situations where you want to receive only pulses. The best example of this is in a server where you've received a request from a client to do something, but can't complete the request just yet (perhaps you have to do a long hardware operation). In such a design, you'd generally set up the hardware (or a timer, or whatever) to send you a pulse whenever a significant event occurs.

If you write your server using the classic "wait in an infinite loop for messages" design, you might run into a situation where one client sends you a request, and then, while you're waiting for the pulse to come in (to signal completion of the request), another client sends you another request. Generally, this is *exactly* what you want—after all, you want to be able to service multiple clients at the same time. However, there might be good reasons why this is not acceptable—servicing a client might be so resource-intensive that you want to limit the number of clients.

In that case, you now need to be able to "selectively" receive only a pulse, and *not* a regular message. This is where *MsgReceivePulse()* comes into play:

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

int MsgReceivePulse (int chid,
                     void *rmsg,
                     size_t rbytes,
                     struct _msg_info *info);
```

As you can see, you use the same parameters as *MsgReceive()*; the channel ID, the buffer (and its size), as well as the *info* parameter. (We discussed the *info* parameter above, in "*Who sent the message?*".) Note that the *info* parameter is not used in the case of a pulse; you might ask why it's present in the parameter list. Simple answer: it was easier to do it that way in the implementation. Just pass a NULL!

The *MsgReceivePulse()* function will receive *nothing but* pulses. So, if you had a channel with a number of threads blocked on it via *MsgReceivePulse()*, (and *no* threads blocked on it via *MsgReceive()*), and a client attempted to send your server a message, the client would remain SEND-blocked until a thread issued the *MsgReceive()* call. Pulses would be transferred via the *MsgReceivePulse()* functions in the meantime.

The only thing you can guarantee if you mix both *MsgReceivePulse()* and *MsgReceive()* is that the *MsgReceivePulse()* will get pulses only. The *MsgReceive()* could get pulses *or* messages! This is because, generally, the use of the *MsgReceivePulse()* function is reserved for the cases where you want to *exclude* regular message delivery to the server.

This does introduce a bit of confusion. Since the *MsgReceive()* function can receive *both* a message and a pulse, but the *MsgReceivePulse()* function can receive only a pulse, how do you deal with a server that makes use of both functions? Generally, the answer here is that you'd have a pool of threads that are performing *MsgReceive()*. This pool of threads (one or more threads; the number depends on how many clients you're prepared to service concurrently) is responsible for handling client calls (requests for service). Since you're trying to control the number of "service-providing threads," and since some of these threads may need to block, waiting for a pulse to arrive (for example, from some hardware or from another thread), you'd typically block the service-providing thread using *MsgReceivePulse()*. This ensures that a client request won't "sneak in" while you're waiting for the pulse (since *MsgReceivePulse()* will receive *only* a pulse).

The *MsgDeliverEvent()* function

As mentioned above in "The send-hierarchy," there are cases when you need to break the natural flow of sends.

Such a case might occur if you had a client that sent a message to the server, the result might not be available for a while, and the client didn't want to block. Of course, you could also partly solve this with threads, by having the client simply "use up" a thread on the blocking server call, but this may not scale well for larger systems (where you'd be using up lots of threads to wait for many different servers). Let's say you didn't want to use a thread, but instead wanted the server to reply immediately to the client, "I'll get around to your request shortly." At this point, since the server replied, the client is now free to continue processing. Once the server has completed whatever task the client gave it, the server now needs some way to tell the client, "Hey, wake up, I'm done." Obviously, as we saw in the send-hierarchy discussion above, you can't have the server send a message to the client, because this might cause deadlock if the client sent a message to the server at that exact same instant. So, how does the server "send" a message to a client *without* violating the send hierarchy?

It's actually a multistep operation. Here's how it works:

1. The client creates a `struct sigevent` structure, and fills it in.

© 2020, QNX Software Systems Limited

2. The client sends a message to the server, effectively stating, "Perform this specific task for me, reply right away, and by the way, here's a `struct sigevent` that you should use to notify me when the work is completed."
3. The server receives the message (which includes the `struct sigevent`), stores the `struct sigevent` and the receive ID away, and replies immediately to the client.
4. The client is now running, as is the server.
5. When the server completes the work, the server uses *MsgDeliverEvent()* to inform the client that the work is now complete.

We'll take a look in detail at the `struct sigevent` in the *Clocks, Timers, and Getting a Kick Every So Often* chapter, under "How to fill in the `struct sigevent`." For now, just think of the `struct sigevent` as a "black box" that somehow contains the event that the server uses to notify the client.

Since the server stored the `struct sigevent` and the receive ID from the client, the server can now call *MsgDeliverEvent()* to deliver the event, *as selected by the client*, to the client:

```
int
MsgDeliverEvent (int rcvid,
                 const struct sigevent *event);
```

Notice that the *MsgDeliverEvent()* function takes two parameters, the receive ID (in *rcvid*) and the event to deliver in *event*. The server *does not modify or examine the event in any way!* This point is important, because it allows the server to deliver *whatever* kind of event the client chose, without any specific processing on the server's part. (The server can, however, *verify* that the event is valid by using the *MsgVerifyEvent()* function.)

The *rcvid* is a receive ID that the server got from the client. Note that this is indeed a special case. Generally, after the server has replied to a client, the receive ID ceases to have any meaning (the reasoning being that the client is unblocked, and the server couldn't unblock it *again*, or read or write data from/to the client, etc.). But in this case, the receive ID contains just enough information for the kernel to be able to decide which client the event should be delivered to. When the server calls the *MsgDeliverEvent()* function, the server doesn't block—this is a non-blocking call for the server. The client has the event delivered to it (by the kernel), and may then perform whatever actions are appropriate.

Channel flags

When we introduced the server (in "The server"), we mentioned that the *ChannelCreate()* function takes a *flags* parameter and that we'd just leave it as zero.

Now it's time to explain the *flags*. We'll examine only a few of the possible *flags* values:

_NTO_CHF_FIXED_PRIORITY

The receiving thread will *not* change priority based on the priority of the sender. (We talk more about priority issues in the "*Priority inheritance*" section, below). Ordinarily (i.e., if you don't specify this flag), the receiving thread's priority is changed to that of the sender.

_NTO_CHF_UNBLOCK

The kernel delivers a pulse whenever a client thread attempts to unblock. The server *must* reply to the client in order to allow the client to unblock. We'll discuss this one below, because it has some very interesting consequences, for both the client and the server.

_NTO_CHF_THREAD_DEATH

The kernel delivers a pulse whenever a thread blocked on this channel dies. This is useful for servers that want to maintain a fixed *"pool of threads"* available to service requests at all times.

_NTO_CHF_DISCONNECT

The kernel delivers a pulse whenever *all* connections from a single client have been disconnected from the server.

_NTO_CHF_COID_DISCONNECT

The kernel delivers a pulse whenever any connection owned by this process is terminated due to the channel on the other end going away.

_NTO_CHF_UNBLOCK

Let's look at the _NTO_CHF_UNBLOCK flag; it has a few interesting wrinkles for both the client and the server.

Normally (i.e., where the server does *not* specify the _NTO_CHF_UNBLOCK flag) when a client wishes to unblock from a *MsgSend()* (and related *MsgSendv()*, *MsgSendvs()*, etc. family of functions), the client simply unblocks. The client could wish to unblock due to receiving a signal or a kernel timeout (see the *TimerTimeout()* function in the QNX Neutrino *C Library Reference*, and the *Clocks, Timers, and Getting a Kick Every So Often* chapter). The unfortunate aspect to this is that the server has *no idea* that the client has unblocked and is no longer waiting for a reply. Note that it isn't possible to write a reliable server with this flag off, except in very special situations which require cooperation between the server and all its clients.

Let's assume that you have a server with multiple threads, all blocked on the server's *MsgReceive()* function. The client sends a message to the server, and one of the server's threads receives it. At this point, the client is blocked, and a thread in the server is actively processing the request. Now, before the server thread has a chance to reply to the client, the client unblocks from the *MsgSend()* (let's assume it was because of a signal).

Remember, a server thread is *still* processing the request on behalf of the client. But since the client is now unblocked (the client's *MsgSend()* would have returned with EINTR), the client is free to send *another* request to the server. Thanks to the architecture of QNX Neutrino servers, another thread would receive another message from the client, with the *exact same receive ID*! The server has no way to tell these two requests apart! When the first thread completes and replies to the client, it's really replying to the *second* message that the client sent, not the first message (as the thread actually believes that it's doing). So, the server's first thread replies to the client's second message.

This is bad enough; but let's take this one step further. Now the server's second thread completes the request and tries to reply to the client. But since the server's first thread *already* replied to the client, the client is now unblocked and the server's second thread gets an error from its reply.

© 2020, QNX Software Systems Limited

This problem is limited to multithreaded servers, because in a single-threaded server, the server thread would still be busy working on the client's first request. This means that even though the client is now unblocked and sends again to the server, the client would now go into the SEND-blocked state (instead of the REPLY-blocked state), allowing the server to finish the processing, reply to the client (which would result in an error, because the client isn't REPLY-blocked any more), and *then* the server would receive the second message from the client. The real problem here is that the server is performing useless processing on behalf of the client (the client's first request). The processing is useless because the client is no longer waiting for the results of that work.

The solution (in the multithreaded server case) is to have the server specify the _NTO_CHF_UNBLOCK flag to its *ChannelCreate()* call. This says to the kernel, "Tell me when a client *tries* to unblock from me (by sending me a pulse), but *don't let the client unblock*! I'll unblock the client myself."

The key thing to keep in mind is that this server flag changes the behavior of the client by *not* allowing the client to unblock until the *server* says it's okay to do so.

In a single-threaded server, the following happens:

Action	Client	Server
Client sends to server	Blocked	Processing
Client gets hit with signal	Blocked	Processing
Kernel sends pulse to server	Blocked	Processing (first message)
Server completes the first request, replies to client	Unblocked with correct data	Processing (pulse)

This didn't help the client unblock when it should have, but it *did* ensure that the server didn't get confused. In this kind of example, the server would most likely simply ignore the pulse that it got from the kernel. This is okay to do—the assumption being made here is that it's safe to let the client block until the server is ready with the data.

If you want the server to act on the pulse that the kernel sent, there are two ways to do this:

- Create another thread in the server that listens for messages (specifically, listening for the pulse from the kernel). This second thread would be responsible for canceling the operation that's under way in the first thread. One of the two threads would reply to the client.
- Don't do the client's work in the thread itself, but rather queue up the work. This is typically done in applications where the server is going to store the client's work on a queue and the server is event driven. Usually, one of the messages arriving at the server indicates that the client's work is now complete, and that the server should reply. In this case, when the kernel pulse arrives, the server cancels the work being performed on behalf of the client and replies.

Which method you choose depends on the type of work the server does. In the first case, the server is actively performing the work on behalf of the client, so you really don't have a choice—you'll have to have a second thread that listens for unblock-pulses from the kernel (or you could poll periodically within the thread to see if a pulse has arrived, but polling is generally discouraged).

In the second case, the server has something else doing the work—perhaps a piece of hardware has been commanded to "go and collect data." In that case, the server's thread will be blocked on the

MsgReceive() function anyway, waiting for an indication from the hardware that the command has completed.

In either case, the server *must* reply to the client, otherwise the client will remain blocked.

Synchronization problem

Even if you use the _NTO_CHF_UNBLOCK flag as described above, there's still one more synchronization problem to deal with. Suppose that you have multiple server threads blocked on the *MsgReceive()* function, waiting for messages or pulses, and the client sends you a message. One thread goes off and begins the client's work. While that's happening, the client wishes to unblock, so the kernel generates the unblock pulse. Another thread in the server receives this pulse. At this point, there's a race condition—the first thread could be *just about* ready to reply to the client. If the second thread (that got the pulse) does the reply, then there's a chance that the client would unblock and send another message to the server, with the server's first thread now getting a chance to run and replying to the client's second request with the first request's data:

Figure 31: Confusion in a multithreaded server.

Or, if the thread that got the pulse is *just about* to reply to the client, and the first thread does the reply, then you have the same situation—the first thread unblocks the client, who sends another request, and the second thread (that got the pulse) now unblocks the client's second request.

The situation is that you have two parallel flows of execution (one caused by the message, and one caused by the pulse). Ordinarily, we'd immediately recognize this as a situation that requires a mutex.

© 2020, QNX Software Systems Limited

Unfortunately, this causes a problem—the mutex would have to be acquired immediately after the *MsgReceive()* and released before the *MsgReply()*. While this will indeed work, it defeats the whole purpose of the unblock pulse! (The server would either get the message and ignore the unblock pulse until after it had replied to the client, or the server would get the unblock pulse and cancel the client's second operation.)

A solution that looks promising (but is ultimately doomed to failure) would be to have a fine-grained mutex. What I mean by that is a mutex that gets locked and unlocked only around small portions of the control flow (the way that you're supposed to use a mutex, instead of blocking the entire processing section, as proposed above). You'd set up a "Have we replied yet?" flag in the server, and this flag would be cleared when you received a message and set when you replied to a message. Just before you replied to the message, you'd check the flag. If the flag indicates that the message has already been replied to, you'd skip the reply. The mutex would be locked and unlocked around the checking and setting of the flag.

Unfortunately, this won't work because we're not *always* dealing with two parallel flows of execution—the client won't always get hit with a signal during processing (causing an unblock pulse). Here's the scenario where it breaks:

- The client sends a message to the server; the client is now blocked, the server is now running.
- Since the server received a request from the client, the flag is reset to 0, indicating that we still need to reply to the client.
- The server replies normally to the client (because the flag was set to 0) and sets the flag to 1 indicating that, if an unblock-pulse arrives, it should be ignored.
- *(Problems begin here.)* The client sends a second message to the server, and almost immediately after sending it gets hit with a signal; the kernel sends an unblock-pulse to the server.
- The server thread that receives the message was about to acquire the mutex in order to check the flag, but didn't quite get there (it got preempted).
- Another server thread now gets the pulse and, because the flag is still set to a 1 from the last time, ignores the pulse.
- Now the server's first thread gets the mutex and clears the flag.
- At this point, the unblock event has been lost.

If you refine the flag to indicate more states (such as pulse received, pulse replied to, message received, message replied to), you'll still run into a synchronization race condition because there's no way for you to create an atomic binding between the flag and the receive and reply function calls. (Fundamentally, that's where the problem lies—the small timing windows after a *MsgReceive()* and before the flag is adjusted, and after the flag is adjusted just before the *MsgReply()*.) The only way to get around this is to have the kernel keep track of the flag for you.

Using the _NTO_MI_UNBLOCK_REQ

Luckily, the kernel keeps track of the flag for you as a single bit in the message info structure (the `struct _msg_info` that you pass as the last parameter to *MsgReceive()*, or that you can fetch later, given the receive ID, by calling *MsgInfo()*).

This flag is called _NTO_MI_UNBLOCK_REQ and is set if the client wishes to unblock (for example, after receiving a signal).

This means that in a multithreaded server, you'd typically have a “worker” thread that's performing the client's work, and another thread that's going to receive the unblock message (or some other message; we'll just focus on the unblock message for now). When you get the unblock message from the client, you'd set a flag to yourself, letting your program know that the worker thread wishes to unblock.

There are two cases to consider:

- the “worker” thread is blocked; or
- the “worker” thread is running.

If the worker thread is blocked, you'll need to have the thread that got the unblock message awaken it. It might be blocked if it's waiting for a resource, for example. When the worker thread wakes up, it should examine the _NTO_MI_UNBLOCK_REQ flag, and, if set, reply with an abort status. If the flag isn't set, then the thread can do whatever normal processing it does when it wakes up.

Alternatively, if the worker thread is running, it should periodically check the “flag to self” that the unblock thread may have set, and if the flag is set, it should reply to the client with an abort status. Note that this is just an optimization: in the unoptimized case, the worker thread would constantly call “MsgInfo” on the receive ID and check the _NTO_MI_UNBLOCK_REQ bit itself.

© 2020, QNX Software Systems Limited

Message passing over a network

To keep things clear, I've avoided talking about how you'd use message passing over a network, even though this is a crucial part of QNX Neutrino's flexibility!

Everything you've learned so far applies to message passing over the network.

Earlier in this chapter, I showed you an example:

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

int
main (void)
{
    int     fd;

    fd = open ("/net/wintermute/home/rk/filename", O_WRONLY);
    write (fd, "This is message passing\n", 24);
    close (fd);

    return (EXIT_SUCCESS);
}
```

At the time, I said that this was an example of “using message passing over a network.” The client creates a connection to a ND/PID/CHID (which just happens to be on a different node), and the server performs a *MsgReceive()* on its channel. The client and server are identical in this case to the local, single-node case. You could stop reading right here—there really isn't anything “tricky” about message passing over the network. But for those readers who are curious about the *how* of this, read on!

Now that we've seen some of the details of local message passing, we can discuss in a little more depth how message passing over a network works. While this discussion may seem complicated, it really boils down to two phases: name resolution, and once that's been taken care of, simple message passing.

Here's a diagram that illustrates the steps we'll be talking about:

Figure 32: Message passing over a network. Notice that Qnet is divided into two sections.

In the diagram, our node is called **magenta**, and, as implied by the example, the target node is called **wintermute**.

Let's analyze the interactions that occur when a client program uses Qnet to access a server over the network:

1. The client's *open()* function was told to open a filename that happened to have **/net** in front of it. (The name **/net** is the default name manifested by Qnet.) This client has no idea who is responsible for that particular pathname, so it connects to the process manager (step 1) in order to find out who actually owns the resource. This is done regardless of whether we're passing messages over a network and happens automatically. Since the native QNX Neutrino network manager, Qnet, “owns” all pathnames that begin with **/net**, the process manager returns information to the client telling it to ask Qnet about the pathname.
2. The client now sends a message to Qnet's resource manager thread, hoping that Qnet will be able to handle the request. However, Qnet on this node isn't responsible for providing the ultimate service that the client wants, so it tells the client that it should actually contact the process manager on node **wintermute**. (The way this is done is via a “redirect” response, which gives the client the ND/PID/CHID of a server that it should contact instead.) This redirect response is also handled automatically by the client's library.
3. The client now connects to the process manager on **wintermute**. This involves sending an off-node message through Qnet's network-handler thread. The Qnet process on the client's node gets the message and transports it over the medium to the remote Qnet, which delivers it to the process manager on **wintermute**. The process manager there resolves the rest of the pathname (in our example, that would be the “**/home/rk/filename**” part) and sends a redirect message back. This redirect message follows the reverse path (from the server's Qnet over the medium to the Qnet on the client's node, and finally back to the client). This redirect message now contains the location of the server that the client wanted to contact in the first place, that is, the ND/PID/CHID of the server that's going to service the client's requests. (In our example, the server was a filesystem.)
4. The client now sends the request to that server. The path followed here is identical to the path followed in step 3 above, except that the server is contacted directly instead of going through the process manager.

© 2020, QNX Software Systems Limited

Once steps 1 through 3 have been established, step 4 is the model for all future communications. In our client example above, the *open()*, *read()*, and *close()* messages all take path number 4. Note that the client's *open()* is what triggered this sequence of events to happen in the first place—but the actual open *message* flows as described (through path number 4).

For the *really* interested reader: I've left out one step. During step 2, when the client asks Qnet about **wintermute**, Qnet needs to figure out who **wintermute** is. This may result in Qnet performing one more network transaction to resolve the nodename. The diagram presented above is correct if we assume that Qnet already knew about **wintermute**.

We'll come back to the messages used for the *open()*, *read()*, and *close()* (and others) in the *Resource Managers* chapter.

Networked message passing differences

So, once the connection is established, all further messaging flows using step 4 in the diagram above. This may lead you to the erroneous belief that message passing over a network is identical to message passing in the local case. Unfortunately, this is *not true*. Here are the differences:

- longer delays
- *ConnectAttach()* returns success regardless of whether the node is alive or not—the real error indication happens on the first message pass
- *MsgDeliverEvent()* isn't guaranteed reliable
- *MsgReply()*, *MsgRead()*, *MsgWrite()* are now blocking calls, whereas in the local case they are not
- *MsgReceive()* might not receive all the data sent by the client; the server might need to call *MsgRead()* to get the rest.

Longer delays

Since message passing is now being done over some medium, rather than a direct kernel-controlled memory-to-memory copy, you can expect that the amount of time taken to transfer messages will be significantly higher (100 MB Ethernet versus 100 MHz 64-bit wide DRAM is going to be an order of magnitude or two slower). Plus, on top of this will be protocol overhead (minimal) and retries on lossy networks.

Impact on *ConnectAttach()*

When you call *ConnectAttach()*, you're specifying an ND, a PID, and a CHID. All that happens in QNX Neutrino is that the kernel returns a connection ID to the Qnet “network handler” thread pictured in the diagram above. Since no message has been sent, you're not informed as to whether the node that you've just attached to is still alive or not. In normal use, this isn't a problem, because most clients won't be doing their own *ConnectAttach()*—rather, they'll be using the services of the library call *open()*, which does the *ConnectAttach()* and then almost immediately sends out an “open” message. This has the effect of indicating almost immediately if the remote node is alive or not.

Impact on *MsgDeliverEvent()*

When a server calls *MsgDeliverEvent()* locally, it's the kernel's responsibility to deliver the event to the target thread. With the network, the server still calls *MsgDeliverEvent()*, but the kernel delivers a "proxy" of that event to Qnet, and it's up to Qnet to deliver the proxy to the other (client-side) Qnet, who'll then deliver the actual event to the client. Things can get screwed up on the server side, because the *MsgDeliverEvent()* function call is non-blocking—this means that once the server has called *MsgDeliverEvent()*, it's running. It's too late to turn around and say, "I hate to tell you this, but you know that *MsgDeliverEvent()* that I said succeeded? Well, it didn't!"

Impact on *MsgReply()*, *MsgRead()*, and *MsgWrite()*

To prevent the problem I just mentioned with *MsgDeliverEvent()* from happening with *MsgReply()*, *MsgRead()*, and *MsgWrite()*, these functions were transformed into blocking calls when used over the network. Locally they'd simply transfer the data and unblock immediately. On the network, we have to (in the case of *MsgReply()*) ensure that the data has been delivered to the client or (in the case of the other two) to actually transfer the data to or from the client over the network.

Impact on *MsgReceive()*

Finally, *MsgReceive()* is affected as well (in the networked case). Not all the client's data may have been transferred over the network by Qnet when the server's *MsgReceive()* unblocks. This is done for performance reasons.

There are two flags in the `struct _msg_info` that's passed as the last parameter to *MsgReceive()* (we've seen this structure in detail in "*Who sent the message?*" above):

msglen

Indicates how much data was actually transferred by the *MsgReceive()* (Qnet likes to transfer 8 KB).

srcmsglen

Indicates how much data the client wanted to transfer (determined by the client).

So, if the client wanted to transfer 1 megabyte of data over the network, the server's *MsgReceive()* would unblock and *msglen* would be set to 8192 (indicating that 8192 bytes were available in the buffer), while *srcmsglen* would be set to 1048576 (indicating that the client tried to send 1 megabyte).

The server then uses *MsgRead()* to get the rest of the data from the client's address space.

Some notes on NDs

The other "funny" thing that we haven't yet talked about when it comes to message passing is this whole business of a "node descriptor" or just "ND" for short.

Recall that we used symbolic node names, like **/net/wintermute** in our examples. Under QNX 4 (the previous version of the OS before QNX Neutrino), native networking was based on the concept of a node ID, a small integer that was unique on the network. Thus, we'd talk about "node 61," or "node 1," and this was reflected in the function calls.

© 2020, QNX Software Systems Limited

Under QNX Neutrino, all nodes are internally referred to by a 32-bit quantity, but it's *not* network unique! What I mean by that is that **wintermute** might think of **spud** as node descriptor number “7,” while **spud** might think of **magenta** as node descriptor number “7” as well. Let me expand that to give you a better picture. This table shows some sample node descriptors that might be used by three nodes, **wintermute**, **spud**, and **foobar**:

Node	wintermute	spud	foobar
wintermute	0	7	4
spud	4	0	6
foobar	5	7	0

Notice how each node's node descriptor for itself is zero. Also notice how **wintermute**'s node descriptor for **spud** is “7,” as is **foobar**'s node descriptor for **spud**. But **wintermute**'s node descriptor for **foobar** is “4” while **spud**'s node descriptor for **foobar** is “6.” As I said, they're not unique across the network, although they are unique on each node. You can effectively think of them as file descriptors—two processes might have the same file descriptor if they access the same file, but they might not; it just depends on who opened which file when.

Fortunately, you don't have to worry about node descriptors, for a number of reasons:

1. Most of the off-node message passing you'll typically be doing will be through higher-level function calls (such as *open()*, as shown in the example above).
2. Node descriptors are not to be cached—if you get one, you're supposed to use it immediately and then forget about it.
3. There are library calls to convert a pathname (like **/net/magenta**) to a node descriptor.

To work with node descriptors, you'll want to include the file **<sys/netmgr.h>** because it includes a bunch of *netmgr_*()* functions.

You'd use the function *netmgr_strtond()* to convert a string into a node descriptor. Once you have this node descriptor, you'd use it immediately in the *ConnectAttach()* function call. Specifically, you shouldn't ever cache it in a data structure! The reason is that the native networking manager may decide to reuse it once all connections to that particular node are disconnected. So, if you got a node descriptor of “7” for **/net/magenta**, and you connected to it, sent a message, and then disconnected, there's a possibility that the native networking manager will return a node descriptor of “7” again for a *different* node.

Since node descriptors aren't unique per network, the question that arises is, “How do you pass these things around the network?” Obviously, **magenta**'s view of what node descriptor “7” is will be radically different from **wintermute**'s. There are two solutions here:

- Don't pass around node descriptors; use the symbolic names (e.g., **/net/wintermute**) instead.
- Use the *netmgr_remote_nd()* function.

The first is a good general-purpose solution. The second solution is reasonably simple to use:

```
int
netmgr_remote_nd (int remote_nd, int local_nd);
```

This function takes two parameters: the *remote_nd* is the node descriptor of the target machine, and *local_nd* is the node descriptor (from the local machine's point of view) to be translated to the remote machine's point of view. The result is the node descriptor that is valid from the remote machine's point of view.

For example, let's say **wintermute** is our local machine. We have a node descriptor of "7" that is valid on our local machine and points to **magenta**. What we'd like to find out is what node descriptor **magenta** uses to talk to us:

```
int     remote_nd;
int     magenta_nd;

magenta_nd = netmgr_strtond ("/net/magenta", NULL);
printf ("Magenta's ND is %d\n", magenta_nd);
remote_nd = netmgr_remote_nd (magenta_nd, ND_LOCAL_NODE);
printf ("From magenta's point of view, we're ND %d\n",
        remote_nd);
```

This might print something similar to:

```
Magenta's ND is 7
From magenta's point of view, we're ND 4
```

This says that *on* **magenta**, the node descriptor "4" refers to our node. (Notice the use of the special constant ND_LOCAL_NODE, which is really zero, to indicate "this node.")

Now, recall that we said (in *"Who sent the message?"*) that the `struct _msg_info` contains, among other things, two node descriptors:

```
struct _msg_info
{
    uint32_t  nd;
    uint32_t  srcnd;
    ...
};
```

We stated in the description for those two fields that:

- *nd* is the receiving node's node descriptor for the transmitting node
- *srcnd* is the transmitting node's node descriptor for the receiving node

So, for our example above, where **wintermute** is the local node and **magenta** is the remote node, when **magenta** sends a message to us (**wintermute**), we'd expect that:

- *nd* would contain 7
- *srcnd* would contain 4

© 2020, QNX Software Systems Limited

Priority inheritance

One of the interesting issues in a realtime operating system is a phenomenon known as *priority inversion*.

Priority inversion manifests itself as, for example, a low-priority thread consuming all available CPU time, even though a higher-priority thread is ready to run.

Now you're probably thinking, “Wait a minute! You said that a higher-priority thread will *always* preempt a lower-priority thread! How can this be?”

This is true; a higher-priority thread will *always* preempt a lower-priority thread. But something interesting can happen. Let's look at a scenario where we have three threads (in three different processes, just to keep things simple), “L” is our low-priority thread, “H” is our high-priority thread, and “S” is a server. This diagram shows the three threads and their priorities:

Figure 33: Three threads at different priorities.

Currently, H is running. S, a higher-priority server thread, doesn't have anything to do right now so it's waiting for a message and is blocked in *MsgReceive()*. L would like to run but is at a lower priority than H, which is running. Everything is as you'd expect, right?

Now H has decided that it would like to go to sleep for 100 milliseconds—perhaps it needs to wait for some slow hardware. At this point, L is running.

This is where things get interesting.

As part of its normal operation, L sends a message to the server thread S, causing S to go READY and (because it's the highest-priority thread that's READY) to start running. Unfortunately, the message that L sent to S was “Compute pi to 5000 decimal places.”

Obviously, this takes more than 100 milliseconds. Therefore, when H's 100 milliseconds are up and H goes READY, guess what? It won't run, because S is READY and at a higher priority!

What happened is that a low-priority thread prevented a higher-priority thread from running by leveraging the CPU via an even higher-priority thread. This is *priority inversion*.

To fix it, we need to talk about *priority inheritance*. A simple fix is to have the server, S, *inherit* the priority of the client thread:

Figure 34: Blocked threads.

In this scenario, when H's 100-millisecond sleep has completed, it goes READY and, because it's the highest-priority READY thread, runs.

Not bad, but there's one more “gotcha.”

Suppose that H now decides that it too would like a computation performed. It wants to compute the 5,034th prime number, so it sends a message to S and blocks.

However, S is still computing pi, at a priority of 5! In our example system, there are lots of other threads running at priorities higher than 5 that are making use of the CPU, effectively ensuring that S *isn't* getting much time to calculate pi.

This is another form of priority inversion. In this case, a lower-priority thread has prevented a higher-priority thread from getting access to a resource. Contrast this with the first form of priority inversion, where the lower-priority thread was effectively *consuming* CPU—in this case it's only *preventing* a higher-priority thread from getting CPU—it's not consuming any CPU itself.

Luckily, the solution is fairly simple here too. Boost the server's priority to be the highest of all blocked clients:

Figure 35: Boosting the server's priority.

This way we take a minor hit by letting L's job run at a priority higher than L, but we do ensure that H gets a fair crack at the CPU.

© 2020, QNX Software Systems Limited

So what's the trick?

There's no trick! QNX Neutrino does this automatically for you. (You can turn off priority inheritance if you don't want it; see the _NTO_CHF_FIXED_PRIORITY flag in the *ChannelCreate()* function's documentation.)

There's a minor design issue here, however. How do you revert the priority to what it was before it got changed?

Your server is running along, servicing requests from clients, adjusting its priority automagically when it unblocks from the *MsgReceive()* call. But when should it adjust its priority *back* to what it was before the *MsgReceive()* call changed it?

There are two cases to consider:

- The server performs some additional processing *after* it properly services the client. This should be done at the *server's* priority, not the client's.
- The server immediately does another *MsgReceive()* to handle the next client request.

In the first case, it would be incorrect for the server to run at the client's priority when it's no longer doing work for that client! The solution is fairly simple. Use the *pthread_setschedparam()* or *pthread_setschedprio()* function (discussed in the *Processes and Threads* chapter) to revert the priority back to what it should be.

What about the other case? The answer is subtly simple: Who cares?

Think about it. What difference does it make if the server becomes RECEIVE-blocked when it was priority 29 versus when it was priority 2? The fact of the matter is it's RECEIVE-blocked! It isn't getting any CPU time, so its priority is irrelevant. As soon as the *MsgReceive()* function unblocks the server, the (new) client's priority is inherited by the server and everything works as expected.

Summary

Message passing is an extremely powerful concept and is one of the main features on which QNX Neutrino (and indeed, all past QNX operating systems) is built.

With message passing, a client and a server exchange messages (thread-to-thread in the same process, thread-to-thread in different processes on the same node, or thread-to-thread in different processes on different nodes in a network). The client sends a message and blocks until the server receives the message, processes it, and replies to the client.

The main advantages of message passing are:

- The content of a message doesn't change based on the location of the destination (local versus networked).
- A message provides a “clean” decoupling point for clients and servers.
- Implicit synchronization and serialization helps simplify the design of your applications.

Chapter 3
Clocks, Timers, and Getting a Kick Every So Often

It's time to take a look at everything related to time in the QNX Neutrino RTOS. We'll see how and why you'd use timers and the theory behind them. Then we'll take a look at getting and setting the realtime clock.

This chapter uses a ticksize of 10 ms, but QNX Neutrino now uses a 1 ms ticksize by default on most systems. This doesn't affect the substance of the issues being discussed.

Clocks and timers

Let's look at a typical system, say a car. In this car, we have a bunch of programs, most of which are running at different priorities. Some of these need to respond to actual external events (like the brakes or the radio tuner), while others need to operate periodically (such as the diagnostics system).

Operating periodically

So how does the diagnostics system “operate periodically?” You can imagine some process in the car's CPU that does something similar to the following:

```
// Diagnostics Process

int
main (void)     // ignore arguments here
{
    for (;;) {
        perform_diagnostics ();
        sleep (15);
    }

    // You'll never get here.
    return (EXIT_SUCCESS);
}
```

Here we see that the diagnostics process runs forever. It performs one round of diagnostics and then goes to sleep for 15 seconds, wakes up, goes through the loop again, and again, ...

Way back in the dim, dark days of single-tasking, where one CPU was dedicated to one user, these sorts of programs were implemented by having the `sleep (15);` code do a busy-wait loop. You'd calculate how fast your CPU was and then write your own *sleep()* function:

```
void
sleep (int nseconds)
{
    long    i;

    while (nseconds--) {
        for (i = 0; i < CALIBRATED_VALUE; i++) ;
    }
}
```

In those days, since nothing else was running on the machine, this didn't present much of a problem, because no other process cared that you were hogging 100% of the CPU in the *sleep()* function.

© 2020, QNX Software Systems Limited

Even today, we sometimes hog 100% of the CPU to do timing functions. Notably, the *nanospin()* function is used to obtain very fine-grained timing, but it does so at the expense of burning CPU at its priority. Use with caution!

If you *did* have to perform some form of "multitasking," it was usually done via an interrupt routine that would hang off the hardware timer or be performed within the "busy-wait" period, somewhat affecting the calibration of the timing. This usually wasn't a concern.

Luckily we've progressed far beyond that point. Recall, from *"Scheduling and the real world"* in the *Processes and Threads* chapter, what causes the kernel to reschedule threads:

- a hardware interrupt
- a kernel call
- a fault (exception)

In this chapter, we're concerned with the first two items on the list: the hardware interrupt and the kernel call.

When a thread calls *sleep()*, the C library contains code that eventually makes a kernel call. This call tells the kernel, "Put this thread on hold for a fixed amount of time." The call removes the thread from the running queue and starts a timer.

Meanwhile, the kernel has been receiving regular hardware interrupts from the computer's clock hardware. Let's say, for argument's sake, that these hardware interrupts occur at *exactly* 10-millisecond intervals.

Let's restate: every time one of these interrupts is handled by the kernel's clock interrupt service routine (ISR), it means that 10 ms have gone by. The kernel keeps track of the time of day by incrementing its time-of-day variable by an amount corresponding to 10 ms every time the ISR runs.

So when the kernel implements a 15-second timer, all it's really doing is:

1. Setting a variable to the current time plus 15 seconds.
2. In the clock ISR, comparing this variable against the time of day.
3. When the time of day is the same as (or greater than) the variable, putting the thread back onto the READY queue.

When multiple timers are outstanding, as would be the case if several threads all needed to be woken at different times, the kernel would simply queue the requests, sorting them by time order—the nearest one would be at the head of the queue, and so on. The variable that the ISR looks at is the one at the head of this queue.

At most 50 timer events are generated per clock tick, to limit the consumption of system resources. If the system has a large number of timers set to expire in the same clock period, then at most 50 of the timer events will actually trigger on time; the rest will be handled on the next clock tick (subject to the same limitation of at most 50 timer events).

That's the end of the timer five-cent tour.

Actually, there's a little bit more to it than first meets the eye.

Clock interrupt sources

So where does the clock interrupt come from?

Here's a diagram that shows the hardware components (and some typical values for a PC) responsible for generating these clock interrupts:

Figure 36: PC clock interrupt sources.

As you can see, there's a high-speed (MHz range) clock produced by the circuitry in the PC. This high-speed clock is then divided by a hardware counter (the 82C54 component in the diagram), which reduces the clock rate to the kHz or hundreds of Hz range (i.e., something that an ISR can actually handle). The clock ISR is a component of the kernel and interfaces directly with the data structures and code of the kernel itself. On non-x86 architectures, a similar sequence of events occurs; some chips have clocks built into the processor.

Note that the high-speed clock is being divided by an *integer* divisor. This means the rate isn't going to be *exactly* 10 ms, because the high-speed clock's rate *isn't* an integer multiple of 10 ms. Therefore, the kernel's ISR in our example above might actually be interrupted after 9.9999296004 ms.

Big deal, right? Well, sure, it's fine for our 15-second counter. 15 seconds is 1500 timer ticks—doing the math shows that it's approximately 106 µs off the mark:

```
15 s - 1500 × 9.9999296004 ms
= 15000 ms - 14999.8944006 ms
= 0.1055994 ms
= 105.5994 µs
```

Unfortunately, continuing with the math, that amounts to 608 ms per day, or about 18.5 seconds per month, or almost 3.7 minutes per year!

You can imagine that with other divisors, the error could be greater or smaller, depending on the rounding error introduced. Luckily, the kernel knows about this and corrects for it.

The point of this story is that regardless of the nice round value shown, the *real* value is selected to be the next *faster* value.

Base timing resolution

Let's say that the timer tick is operating at just slightly faster than 10 ms. Can I reliably sleep for 3 milliseconds?

Nope (or at least, not originally).

Consider what happens in the kernel. You issue the C-library *delay()* call to go to sleep for 3 milliseconds. The kernel has to set the variable in the ISR to some value. If it sets it to the current time, this means

© 2020, QNX Software Systems Limited

the timer has already expired and that you should wake up immediately. If it sets it to one tick more than the current time, this means that you should wake up on the next tick (*up to* 10 milliseconds away).

The moral of this story is: "Don't expect timing resolution any better than the input timer tick rate."

This has changed; in QNX Neutrino 7.0 and later, you can use a *high-resolution timer*. For details, see "Tolerant and high-resolution timers" in the "Understanding the Microkernel's Concept of Time" chapter of the *Programmer's Guide*.

Getting more precision

Under QNX Neutrino, a program can adjust the value of the hardware divisor component in conjunction with the kernel (so that the kernel knows what rate the timer tick ISR is being called at). We'll look at this below in the "*Getting and setting the realtime clock*" section.

Timing jitter

There's one more thing you have to worry about. Let's say the timing resolution is 10 ms and you want a 20 ms timeout.

Are you always going to get exactly 20 milliseconds worth of delay from the time that you issue the *delay()* call to the time that the function call returns?

Absolutely not.

There are two good reasons why. The first is fairly simple: when you block, you're taken off the running queue. This means that another thread at your priority may now be using the CPU. When your 20 milliseconds have expired, you'll be placed at the *end* of the READY queue for that priority so you'll be at the mercy of whatever thread happens to be running. This also applies to interrupt handlers running or higher-priority threads running—just because you are READY doesn't mean that you're consuming the CPU.

The second reason is a bit more subtle. The following diagram will help explain why:

Figure 37: Clock jitter.

The problem is that your request is *asynchronous* to the clock source. You have no way to synchronize the hardware clock with your request. Therefore, you'll get from just over 20 milliseconds to just under 30 milliseconds worth of delay, depending on where in the hardware's clock period you started your request.

This is a key point. Clock jitter is a sad fact of life. The way to get around it is to increase the system's timing resolution so your timing is within tolerance. (We'll see how to do this in the *"Getting and setting the realtime clock"* section, below.) Keep in mind that jitter takes place

only on the first tick—a 100-second delay with a 10-millisecond clock will delay for greater than 100 seconds and less than 100.01 seconds.

Types of timers

The type of timer that I showed you above is a *relative timer*. The timeout period selected is relative to the current time. If you want the timer to delay your thread until January 20, 2005 at 12:04:33 EDT, you'd have to calculate the number of seconds from "now" until then, and set up a relative timer for that number of seconds. Because this is a fairly common function, QNX Neutrino implements an *absolute timer* that will delay *until* the specified time (instead of *for* the specified time, like a relative timer).

What if you want to *do* something while you're waiting for that date to come around? Or, what if you want to do something and get a "kick" every 27 seconds? You certainly couldn't afford to be asleep!

As we discussed in the Processes and Threads chapter, you could simply start up another thread to do the work, and your thread could take the delay. However, since we're talking about timers, we'll look at another way of doing this.

You can do this with a periodic or one-shot timer, depending on your objectives. A *periodic timer* is one that goes off periodically, notifying the thread (over and over again) that a certain time interval has elapsed. A *one-shot timer* is one that goes off just once.

The implementation in the kernel is still based on the same principle as the delay timer that we used in our first example. The kernel takes the absolute time (if you specified it that way) and stores it. In the clock ISR, the stored time is compared against the time of day in the usual manner.

However, instead of your thread being removed from the running queue when you call the kernel, your thread continues to run. When the time of day reaches the stored time, the kernel notifies your thread that the designated time has been reached.

Notification schemes

How do you receive a timeout notification? With the delay timer, you received notification by virtue of being made READY again.

With periodic and one-shot timers, you have a choice:

- send a pulse
- send a signal
- create a thread

We've talked about pulses in the *Message Passing* chapter; signals are a standard UNIX-style mechanism, and we'll see the thread creation notification type shortly.

© 2020, QNX Software Systems Limited

How to fill in the `struct sigevent`

Let's take a quick look at how you fill in the `struct sigevent` structure.

Regardless of the notification scheme you choose, you'll need to fill in a `struct sigevent` structure. For the details, see the *C Library Reference*; here's a simplified version:

```
struct sigevent {
    int                             sigev_notify;
    union {
            int                     __sigev_signo;
            int                     __sigev_coid;
            int                     __sigev_id;
            void                    (*__sigev_notify_function)(union sigval);
            volatile unsigned       *__sigev_addr;
    } __sigev_un1;

    union sigval                    sigev_value;

    union {
            struct {
                    short           __sigev_code;
                    short           __sigev_priority;
            } __st;
            pthread_attr_t          *__sigev_notify_attributes;
            int                     __sigev_memop;
        } __sigev_un2;

};
```

Note that the above definition uses various unions and structures, but there are `#define` statements that make it easier for you to access the members. Check out **<sys/siginfo.h>** for details.

The first field you have to fill in is the *sigev_notify* member. This determines the notification type you've selected:

SIGEV_PULSE

A pulse will be sent.

SIGEV_SIGNAL, SIGEV_SIGNAL_CODE, or SIGEV_SIGNAL_THREAD

A signal will be sent.

SIGEV_UNBLOCK

Not used in this case; used with kernel timeouts (see *"Kernel timeouts"* below).

SIGEV_INTR

Not used in this case; used with interrupts (see the *Interrupts* chapter).

SIGEV_THREAD

Creates a thread.

Since we're going to be using the `struct sigevent` with timers, we're concerned only with the SIGEV_PULSE, SIGEV_SIGNAL* and SIGEV_THREAD values for *sigev_notify*; we'll see the other types as mentioned in the list above.

Pulse notification

To send a pulse when the timer fires, set the *sigev_notify* field to SIGEV_PULSE and provide some extra information:

Field	Value and meaning
sigev_coid	Send the pulse to the channel associated with this connection ID.
sigev_value	A 32-bit value that gets sent to the connection identified in the *sigev_coid* field.
sigev_code	An 8-bit value that gets sent to the connection identified in the *sigev_coid* field.
sigev_priority	The pulse's delivery priority. The value zero is not allowed (too many people were getting bitten by running at priority zero when they got a pulse—priority zero is what the idle task runs at, so effectively they were competing with QNX Neutrino's IDLE process and not getting much CPU time `:-)`).

Note that the *sigev_coid* could be a connection to *any* channel (usually, though not necessarily, the channel associated with the process that's initiating the event).

Signal notification

To send a signal, set the *sigev_notify* field to one of:

SIGEV_SIGNAL

Send a regular signal to the process.

SIGEV_SIGNAL_CODE

Send a signal containing an 8-bit code to the process.

SIGEV_SIGNAL_THREAD

Send a signal containing an 8-bit code to a specific thread.

For SIGEV_SIGNAL*, the additional fields you'll have to fill are:

Field	Value and meaning
sigev_signo	Signal number to send (from **<signal.h>**, e.g., SIGALRM).
sigev_code	An 8-bit code (if using SIGEV_SIGNAL_CODE or SIGEV_SIGNAL_THREAD).

© 2020, QNX Software Systems Limited

Thread notification

To create a thread whenever the timer fires, set the *sigev_notify* field to SIGEV_THREAD and fill these fields:

Field	Value and meaning
sigev_notify_function	Address of `void *` function that accepts a `void *` to be called when the event triggers.
sigev_value	Value passed as the parameter to the *sigev_notify_function()* function.
sigev_notify_attributes	Thread attributes structure (see the *Processes and Threads chapter, under "The thread attributes structure"* for details).

This notification type is a little scary! You could have a whole slew of threads created if the timer fires often enough and, if there are higher priority threads waiting to run, this could chew up all available resources on the system! Use with caution!

(QNX Neutrino 7.0.4 or later) In order to use an event of type SIGEV_THREAD, your process must have the PROCMGR_AID_SIGEV_THREAD ability enabled. For more information, see *procmgr_ability()*.

General tricks for notification

There are some convenience macros in **<sys/siginfo.h>** to make filling in the notification structures easier (see the entry for `sigevent` in the QNX Neutrino *C Library Reference*):

SIGEV_SIGNAL_INIT(eventp, signo)

Fill *eventp* with SIGEV_SIGNAL, and the appropriate signal number *signo*.

SIGEV_SIGNAL_CODE_INIT(eventp, signo, value, code)

Fill *eventp* with SIGEV_SIGNAL_CODE, the signal number *signo*, as well as the *value* and *code*.

SIGEV_SIGNAL_THREAD_INIT(eventp, signo, value, code)

Fill *eventp* with SIGEV_SIGNAL_THREAD, the signal number *signo*, as well as the *value* and *code*.

SIGEV_PULSE_INIT(eventp, coid, priority, code, value)

Fill *eventp* with SIGEV_PULSE, the connection to the channel in *coid* and a *priority*, *code*, and *value*. Note that there is a special value for *priority* of SIGEV_PULSE_PRIO_INHERIT that causes the receiving thread to run at the process's initial priority.

SIGEV_UNBLOCK_INIT(eventp)

Fill *eventp* with SIGEV_UNBLOCK.

SIGEV_INTR_INIT(eventp)

Fill *eventp* with SIGEV_INTR.

SIGEV_THREAD_INIT(eventp, func, val, attributes)

Fill *eventp* with the thread function (*func*) and the attributes structure (*attributes*). The value in *val* is passed to the function in *func* when the thread is executed.

Pulse notification

Suppose you're designing a server that spent most of its life RECEIVE blocked, waiting for a message. Wouldn't it be ideal to receive a special message, one that told you that the time you had been waiting for finally arrived?

This scenario is *exactly* where you should use pulses as the notification scheme. In the *"Using timers"* section below, I'll show you some sample code that can be used to get periodic pulse messages.

Signal notification

Suppose that, on the other hand, you're performing some kind of work, but don't want that work to go on forever. For example, you may be waiting for some function call to return, but you can't predict how long it takes.

In this case, using a signal as the notification scheme, with perhaps a signal handler, is a good choice (another choice we'll discuss later is to use kernel timeouts; see _NTO_CHF_UNBLOCK in the Message Passing chapter as well). In the *"Using timers"* section below, we'll see a sample that uses signals.

It isn't safe to use floating-point operations in signal handlers.

Alternatively, a signal with *sigwait()* is cheaper than creating a channel to receive a pulse on, if you're not going to be receiving messages in your application anyway.

© 2020, QNX Software Systems Limited

Using timers

Having looked at all this wonderful theory, let's turn our attention to some specific code samples to see what you can do with timers.

To work with a timer, you must:

1. Create the timer object.
2. Decide how you wish to be notified (signal, pulse, or thread creation), and create the notification structure (the `struct sigevent`).
3. Decide what kind of timer you wish to have (relative versus absolute, and one-shot versus periodic).
4. Start it.

Let's look at these in order.

Creating a timer

The first step is to create the timer with *timer_create()*:

```
#include <time.h>
#include <sys/siginfo.h>

int
timer_create (clockid_t clock_id,
              struct sigevent *event,
              timer_t *timerid);
```

The *clock_id* argument tells the *timer_create()* function *which* time base you're creating this timer for. This is a POSIX thing—POSIX says that on different platforms you can have multiple time bases, but that every platform must support at least the CLOCK_REALTIME time base. Under QNX Neutrino, the time bases to choose from include the following:

- CLOCK_REALTIME
- CLOCK_SOFTTIME
- CLOCK_MONOTONIC

For now, we'll ignore CLOCK_SOFTTIME and CLOCK_MONOTONIC (even though CLOCK_MONOTONIC is the best choice for timers), but we will come back to them in the *"Other clock sources"* section, below.

Signal, pulse, or thread?

The second parameter is a pointer to a `struct sigevent` data structure. This data structure is used to inform the kernel about what kind of event the timer should deliver whenever it "fires." We discussed how to fill in the `struct sigevent` above in the discussion of signals versus pulses versus thread creation.

So, you'd call *timer_create()* with CLOCK_REALTIME and a pointer to your `struct sigevent` data structure, and the kernel would create a timer object for you (which gets returned in the last

argument). This timer object is just a small integer that acts as an index into the kernel's timer tables; think of it as a "handle."

At this point, nothing else is going to happen. You've only just *created* the timer; you haven't triggered it yet.

What kind of timer?

Having created the timer, you now have to decide what *kind* of timer it is. This is done by a combination of arguments to *timer_settime()*, the function used to actually start the timer:

```
#include <time.h>

int
timer_settime (timer_t timerid,
               int flags,
               struct itimerspec *value,
               struct itimerspec *oldvalue);
```

The *timerid* argument is the value that you got back from the *timer_create()* function call—you can create a bunch of timers, and then call *timer_settime()* on them individually to set and start them at your convenience.

The *flags* argument is where you specify absolute versus relative.

If you pass the constant TIMER_ABSTIME, then it's absolute, pretty much as you'd expect. You then pass the actual date and time when you want the timer to go off.

If you pass a zero, then the timer is considered relative to the current time.

Let's look at how you specify the times. Here are key portions of two data structures (in **<time.h>**):

```
struct timespec {
    long    tv_sec,
            tv_nsec;
};

struct itimerspec {
    struct timespec it_value,
                    it_interval;
};
```

There are two members in `struct itimerspec`:

it_value

the one-shot value

it_interval

the reload value

© 2020, QNX Software Systems Limited

The *it_value* specifies either how long from now the timer should go off (in the case of a relative timer), or when the timer should go off (in the case of an absolute timer). Once the timer fires, the *it_interval* value specifies a relative value to reload the timer with so that it can trigger again. Note that specifying a value of zero for the *it_interval* makes it into a one-shot timer. You might expect that to create a "pure" periodic timer, you'd just set the *it_interval* to the reload value, and set *it_value* to zero. Unfortunately, the last part of that statement is false—setting the *it_value* to zero *disables* the timer. If you want to create a pure periodic timer, set *it_value* equal to *it_interval* and create the timer as a relative timer. This will fire once (for the *it_value* delay) and then keep reloading with the *it_interval* delay.

Both the *it_value* and *it_interval* members are actually structures of type `struct timespec`, another POSIX thing. The structure lets you specify sub-second resolutions. The first member, *tv_sec*, is the number of seconds; the second member, *tv_nsec*, is the number of nanoseconds in the current second. (What this means is that you should never set *tv_nsec* past the value 1 billion—this would imply more than a one-second offset.)

Here are some examples:

```
it_value.tv_sec = 5;
it_value.tv_nsec = 500000000;
it_interval.tv_sec = 0;
it_interval.tv_nsec = 0;
```

This creates a one-shot timer that goes off in 5.5 seconds. (We got the ".5" because of the 500,000,000 nanoseconds value.)

We're assuming that this is used as a relative timer, because if it weren't, then that time would have elapsed long ago (5.5 seconds past January 1, 1970, 00:00 GMT).

Here's another example:

```
it_value.tv_sec = 987654321;
it_value.tv_nsec = 0;
it_interval.tv_sec = 0;
it_interval.tv_nsec = 0;
```

This creates a one-shot timer that goes off Thursday, April 19, 2001 at 00:25:21 EDT. (There are a bunch of functions that help you convert between the human-readable date and the "number of seconds since January 1, 1970, 00:00:00 GMT" representation. Take a look in the C library at *time()*, *asctime()*, *ctime()*, *mktime()*, *strftime()*, etc.)

For this example, we're assuming that it's an absolute timer, because of the huge number of seconds that we'd be waiting if it were relative (987654321 seconds is about 31.3 years).

Note that in both examples, I've said, "We're assuming that..." There's nothing in the *code* for *timer_settime()* that checks those assumptions and does the "right" thing! *You* have to specify whether the timer is absolute or relative yourself. The kernel will happily schedule something 31.3 years into the future.

One last example:

```
it_value.tv_sec = 1;
it_value.tv_nsec = 0;
```

```
it_interval.tv_sec = 0;
it_interval.tv_nsec = 500000000;
```

Assuming it's relative, this timer will go off in one second, and then again every half second after that. There's absolutely no requirement that the reload values look anything like the one-shot values.

A server with periodic pulses

The first thing we should look at is a server that wants to get periodic messages.

The most typical uses for this are:

- server-maintained timeouts on client requests
- periodic server maintenance cycles

Of course there are other, specialized uses for these things, such as network “keep alive” messages that need to be sent periodically, retry requests, etc.

Server-maintained timeouts

In this scenario, a server is providing some kind of service to a client, and the client has the ability to specify a timeout. There are lots of places where this is used. For example, you may wish to tell a server, “Get me 15 seconds' worth of data,” or “Let me know when 10 seconds are up,” or “Wait for data to show up, but if it doesn't show up within 2 minutes, time out.”

These are all examples of server-maintained timeouts. The client sends a message to the server, and blocks. The server receives periodic messages from a timer (perhaps once per second, perhaps more or less often), and counts how many of those messages it's received. When the number of timeout messages exceeds the timeout specified by the client, the server replies to the client with some kind of timeout indication or perhaps with the data accumulated so far—it really depends on how the client/server relationship is structured.

Here's a complete example of a server that accepts one of two messages from clients and a timeout message from a pulse. The first client message type says, “Let me know if there's any data available, but don't block me for more than 5 seconds.” The second client message type says, “Here's some data.” The server should allow multiple clients to be blocked on it, waiting for data, and must therefore associate a timeout with the clients. This is where the pulse message comes in; it says, “One second has elapsed.”

In order to keep the code sample from being one overwhelming mass, I've included some text before each of the major sections. You can find the complete version of *time1.c* in the Sample Programs appendix.

Declarations

The first section of code here sets up the various manifest constants that we'll be using, the data structures, and includes all the header files required. We'll present this without comment. :-)

```
/*
 *  time1.c
 *
 *  Example of a server that receives periodic messages from
```


© 2020, QNX Software Systems Limited

```
 *  a timer, and regular messages from a client.
 *
 *  Illustrates using the timer functions with a pulse.
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <signal.h>
#include <errno.h>
#include <unistd.h>
#include <sys/siginfo.h>
#include <sys/neutrino.h>

// message send definitions

// messages
#define MT_WAIT_DATA        2       // message from client
#define MT_SEND_DATA        3       // message from client

// pulses
#define CODE_TIMER          1       // pulse from timer

// message reply definitions
#define MT_OK               0       // message to client
#define MT_TIMEDOUT         1       // message to client

// message structure
typedef struct
{
    // contains both message to and from client
    int messageType;
    // optional data, depending upon message
    int messageData;
} ClientMessageT;

typedef union
{
    // a message can be either from a client, or a pulse
    ClientMessageT  msg;
    struct _pulse   pulse;
} MessageT;
```

```
// client table
#define MAX_CLIENT 16       // max # of simultaneous clients

struct
{
    int in_use;             // is this client entry in use?
    int rcvid;              // receive ID of client
    int timeout;            // timeout left for client
}   clients [MAX_CLIENT];   // client table

int     chid;               // channel ID (global)
int     debug = 1;          // set debug value, 1=on, 0=off
char    *progname = "time1.c";

// forward prototypes
static  void setupPulseAndTimer (void);
static  void gotAPulse (void);
static  void gotAMessage (int rcvid, ClientMessageT *msg);
```

main()

This next section of code is the mainline. It's responsible for:

- creating the channel (via *ChannelCreate()*),
- calling the *setupPulseAndTimer()* routine (to set up a once-per-second timer, with a pulse as the event delivery method), and then
- sitting in a “do-forever” loop waiting for pulses or messages and processing them.

Notice the check against the return value from *MsgReceive()*; a zero indicates it's a pulse (and we don't do any strong checking to ensure that it's *our* pulse), a nonzero indicates it's a message. The processing of the pulse or message is done by *gotAPulse()* and *gotAMessage()*.

```
int
main (void)                 // ignore command-line arguments
{
    int rcvid;              // process ID of the sender
    MessageT msg;           // the message itself

    if ((chid = ChannelCreate (0)) == -1) {
        fprintf (stderr, "%s:  couldn't create channel!\n",
                 progname);
        perror (NULL);
        exit (EXIT_FAILURE);
    }

    // set up the pulse and timer
```


© 2020, QNX Software Systems Limited

```
    setupPulseAndTimer ();

    // receive messages
    for (;;) {
        rcvid = MsgReceive (chid, &msg, sizeof (msg), NULL);

        // determine who the message came from
        if (rcvid == 0) {
            // production code should check "code" field...
            gotAPulse ();
        } else {
            gotAMessage (rcvid, &msg.msg);
        }
    }

    // you'll never get here
    return (EXIT_SUCCESS);
}
```

setupPulseAndTimer()

In *setupPulseAndTimer()* you see the code where we define the type of timer and notification scheme. When we talked about the timer function calls in the text above, I said that the timer could deliver a signal, a pulse, or cause a thread to be created. That decision is made here (in *setupPulseAndTimer()*).

Notice that we used the macro *SIGEV_PULSE_INIT()*. By using this macro, we're effectively assigning the value SIGEV_PULSE to the *sigev_notify* member. (Had we used one of the *SIGEV_SIGNAL*_INIT()* macros instead, it would have delivered the specified signal.) Notice that, for the pulse, we set the connection back to ourselves via the *ConnectAttach()* call, and give it a code that uniquely identifies it (we chose the manifest constant CODE_TIMER, something that *we* defined). The final parameter in the initialization of the event structure is the priority of the pulse; we chose SIGEV_PULSE_PRIO_INHERIT (the constant -1). This tells the kernel not to change the priority of the receiving thread when the pulse arrives.

Near the bottom of this function, we call *timer_create()* to create a timer object within the kernel, and then we fill it in with data saying that it should go off in one second (the *it_value* member) and that it should reload with one-second repeats (the *it_interval* member). Note that the timer is activated only when we call *timer_settime()*, *not* when we create it.

The SIGEV_PULSE notification scheme is a QNX Neutrino extension; POSIX has no concept of pulses.

```
/*
 *  setupPulseAndTimer
 *
 *  This routine is responsible for setting up a pulse so it
 *  sends a message with code MT_TIMER.  It then sets up a
```

```
 *  periodic timer that fires once per second.
*/

void
setupPulseAndTimer (void)
{
    timer_t             timerid;    // timer ID for timer
    struct sigevent     event;      // event to deliver
    struct itimerspec   timer;      // the timer data structure
    int                 coid;       // connection back to ourselves

    // create a connection back to ourselves
    coid = ConnectAttach (0, 0, chid, 0, 0);
    if (coid == -1) {
        fprintf (stderr, "%s:  couldn't ConnectAttach to self!\n",
                 progname);
        perror (NULL);
        exit (EXIT_FAILURE);
    }

    // set up the kind of event that we want to deliver -- a pulse
    SIGEV_PULSE_INIT (&event, coid,
                      SIGEV_PULSE_PRIO_INHERIT, CODE_TIMER, 0);

    // create the timer, binding it to the event
    if (timer_create (CLOCK_REALTIME, &event, &timerid) == -1) {
        fprintf (stderr, "%s:  couldn't create a timer, errno %d\n",
                 progname, errno);
        perror (NULL);
        exit (EXIT_FAILURE);
    }

    // setup the timer (1s delay, 1s reload)
    timer.it_value.tv_sec = 1;
    timer.it_value.tv_nsec = 0;
    timer.it_interval.tv_sec = 1;
    timer.it_interval.tv_nsec = 0;

    // and start it!
    timer_settime (timerid, 0, &timer, NULL);
}
```


© 2020, QNX Software Systems Limited

gotAPulse()

In *gotAPulse()*, you can see how we've implemented the server's ability to time out a client. We walk down the list of clients, and since we know that the pulse is being triggered once per second, we simply decrement the number of seconds that the client has left before a timeout. If this value reaches zero, we reply back to that client with a message saying, Sorry, timed out (the MT_TIMEDOUT message type). You'll notice that we prepare this message ahead of time (outside the `for` loop), and then send it as needed. This is just a style/usage issue—if you expect to be doing a lot of replies, then it might make sense to incur the setup overhead once. If you don't expect to do a lot of replies, then it might make more sense to set it up as needed.

If the timeout value hasn't yet reached zero, we don't do anything about it—the client is still blocked, waiting for a message to show up.

```
/*
 *  gotAPulse
 *
 *  This routine is responsible for handling the fact that a
 *  timeout has occurred.  It runs through the list of clients
 *  to see which client has timed out, and replies to it with
 *  a timed-out response.
 */

void
gotAPulse (void)
{
    ClientMessageT  msg;
    int             i;

    if (debug) {
        time_t  now;

        time (&now);
        printf ("Got a Pulse at %s", ctime (&now));
    }

    // prepare a response message
    msg.messageType = MT_TIMEDOUT;

    // walk down list of clients
    for (i = 0; i < MAX_CLIENT; i++) {

        // is this entry in use?
        if (clients [i].in_use) {
```

```
            // is it about to time out?
            if (--clients [i].timeout == 0) {

                // send a reply
                MsgReply (clients [i].rcvid, EOK, &msg,
                          sizeof (msg));

                // entry no longer used
                clients [i].in_use = 0;
            }
        }
    }
}
```

gotAMessage()

In *gotAMessage()*, you see the other half of the functionality, where we add a client to the list of clients waiting for data (if it's a MT_WAIT_DATA message), or we match up a client with the message that just arrived (if it's a MT_SEND_DATA message). Note that for simplicity we didn't add a queue of clients that are waiting to send data, but for which no receiver is yet available—that's a queue management issue left as an exercise for the reader!

```
/*
 *  gotAMessage
 *
 *  This routine is called whenever a message arrives.  We
 *  look at the type of message (either a "wait for data"
 *  message, or a "here's some data" message), and act
 *  accordingly.  For simplicity, we'll assume that there is
 *  never any data waiting.  See the text for more discussion
 *  about this.
*/

void
gotAMessage (int rcvid, ClientMessageT *msg)
{
    int i;

    // determine the kind of message that it is
    switch (msg -> messageType) {

    // client wants to wait for data
    case    MT_WAIT_DATA:

        // see if we can find a blank spot in the client table
```


© 2020, QNX Software Systems Limited

```
        for (i = 0; i < MAX_CLIENT; i++) {

            if (!clients [i].in_use) {

                // found one -- mark as in use, save rcvid, set timeout
                clients [i].in_use = 1;
                clients [i].rcvid = rcvid;
                clients [i].timeout = 5;
                return;
            }
        }

        fprintf (stderr, "Table full, message from rcvid %d ignored, "
                         "client blocked\n", rcvid);
        break;

    // client with data
    case    MT_SEND_DATA:

        // see if we can find another client to reply to with
        // this client's data
        for (i = 0; i < MAX_CLIENT; i++) {

            if (clients [i].in_use) {

                // found one -- reuse the incoming message
                // as an outgoing message
                msg -> messageType = MT_OK;

                // reply to BOTH CLIENTS!
                MsgReply (clients [i].rcvid, EOK, msg,
                          sizeof (*msg));
                MsgReply (rcvid, EOK, msg, sizeof (*msg));

                clients [i].in_use = 0;
                return;
            }
        }

        fprintf (stderr, "Table empty, message from rcvid %d ignored, "
                         "client blocked\n", rcvid);
        break;
```

```
        }
    }
```

Notes

Some general notes about the code:

- If there's no one waiting and a data message arrives, or there's no room in the list for a new waiter client, we print a message to standard error, but never reply to the client. This means that some clients could be sitting there, REPLY-blocked forever—we've lost their receive ID, so we have no way to reply to them later.

  This is intentional in the design. You could modify this to add MT_NO_WAITERS and MT_NO_SPACE messages, respectively, which can be returned whenever these errors were detected.
- When a waiter client is waiting, and a data-supplying client sends to it, we reply to *both* clients. This is crucial, because we want *both* clients to unblock.
- We reused the data-supplying client's buffer for both replies. This again is a style issue—in a larger application you'd probably have to have multiple types of return values, in which case you may not want to reuse the same buffer.
- The implementation shown here uses a "cheesy" fixed-length array with an "in use" flag (`clients[i].in_use`). Since my goal here isn't to demonstrate owner-list tricks and techniques for singly linked list management, I've shown the version that's the easiest to understand. Of course, in your production code, you'd probably use a linked list of dynamically managed storage blocks.
- When the message arrives in the *MsgReceive()*, our decision as to whether it was in fact "our" pulse is done on weak checking—we assume (as per the comments) that *all* pulses are the CODE_TIMER pulse. Again, in your production code you'd want to check the pulse's code value and report on any anomalies.

Note that the example above shows just *one* way of implementing timeouts for clients. Later in this chapter (in "*Kernel timeouts*"), we'll talk about kernel timeouts, which are another way of implementing almost the exact same thing, except that it's driven by the client, rather than a timer.

Periodic server maintenance cycles

Here we have a slightly different use for the periodic timeout messages. The messages are purely for the internal use of the server and generally have nothing to do with the client at all.

For example, some hardware might require that the server poll it periodically, as might be the case with a network connection—the server should see if the connection is still "up," regardless of any instructions from clients.

Another case could occur if the hardware has some kind of "inactivity shutdown" timer. For example, since keeping a piece of hardware powered up for long periods of time may waste power, if no one has used that hardware for, say, 10 seconds, the hardware could be powered down. Again, this has nothing to do with the client (except that a client request will cancel this inactivity powerdown)—it's just something that the server has to be able to provide for its hardware.

Code-wise, this would be very similar to the example above, except that instead of having a list of clients that are waiting, you'd have only one timeout variable. Whenever a timer event arrives, this

 © 2020, QNX Software Systems Limited

variable would be decremented; if zero, it would cause the hardware to shut down (or whatever other activity you wish to perform at that point). If it's still greater than zero, nothing would happen.

The only “twist” in the design would be that whenever a message comes in from a client that uses the hardware, you'd have to reset that timeout variable back to its full value—having someone use that resource resets the “countdown.” Conversely, the hardware may take a certain “warm-up” time in order to recover from being powered down. In this case, once the hardware has been powered down, you would have to set a different timer once a request arrived from a client. The purpose of this timer would be to delay the client's request from going to the hardware until the hardware has been powered up again.

Timers delivering signals

So far, we've seen just about all there is to see with timers, except for one small thing. We've been delivering messages (via a pulse), but you can also deliver POSIX signals. Let's see how this is done:

```
timer_create (CLOCK_REALTIME, NULL, &timerid);
```

This is the simplest way to create a timer that sends you a signal. This method raises SIGALRM when the timer fires. If we had actually supplied a `struct sigevent`, we could specify which signal we actually want to get:

```
struct sigevent event;

SIGEV_SIGNAL_INIT (&event, SIGUSR1);
timer_create (CLOCK_REALTIME, &event, &timerid);
```

This hits us with SIGUSR1 instead of SIGALRM.

You catch timer signals with normal signal handlers; there's nothing special about them.

Timers creating threads

If you'd like to create a new thread every time a timer fires, then you can do so with the `struct sigevent` and all the other timer stuff we just discussed:

```
struct sigevent event;

SIGEV_THREAD_INIT (&event, maintenance_func, NULL);
```

You'll want to be particularly careful with this one, because if you specify too short an interval, you'll be flooded with new threads! This could eat up all your CPU and memory resources!

Getting and setting the realtime clock and more

Apart from using timers, you can also get and set the current realtime clock, and adjust it gradually.

The following functions can be used for these purposes:

Function	Type?	Description
ClockAdjust()	QNX Neutrino	Gradually adjust the time
ClockCycles()	QNX Neutrino	High-resolution snapshot
clock_getres()	POSIX	Fetch the base timing resolution
clock_gettime()	POSIX	Get the current time of day
ClockPeriod()	QNX Neutrino	Get or set the base timing resolution
clock_settime()	POSIX	Set the current time of day
ClockTime()	QNX Neutrino	Get or set the current time of day

Getting and setting

The functions *clock_gettime()* and *clock_settime()* are the POSIX functions based on the kernel function *ClockTime()*. These functions can be used to get or set the current time of day. Unfortunately, setting this is a “hard” adjustment, meaning that whatever time you specify in the buffer is immediately taken as the *current* time. This can have startling consequences, especially when time appears to move “backwards” because the time was ahead of the “real” time. Generally, setting a clock using this method should be done only during power up or when the time is very much out of synchronization with the real time. That said, to effect a gradual change in the current time, the function *ClockAdjust()* can be used:

```
int
ClockAdjust (clockid_t id,
             const struct _clockadjust *new,
             const struct _clockadjust *old);
```

The parameters are the clock source (always use CLOCK_REALTIME), and a *new* and *old* parameter. Both the *new* and *old* parameters are optional, and can be NULL. The *old* parameter simply returns the current adjustment. The operation of the clock adjustment is controlled through the *new* parameter, which is a pointer to a structure that contains two elements, *tick_nsec_inc* and *tick_count*. Basically, the operation of *ClockAdjust()* is very simple. Over the next *tick_count* clock ticks, the adjustment contained in *tick_nsec_inc* is added to the current system clock. This means that to move the time forward (to “catch up” with the real time), you'd specify a positive value for *tick_nsec_inc*. Note that you'd never move the time backwards! Instead, if your clock was too fast, you'd specify a small negative number to *tick_nsec_inc*, which would cause the current time to not advance as fast as it would. So effectively, you've slowed down the clock until it matches reality. A rule of thumb is that you shouldn't adjust the clock by more than 10% of the base timing resolution of your system (as indicated by the functions we'll talk about next, *ClockPeriod()* and friends).

© 2020, QNX Software Systems Limited

Adjusting the timebase

As we've been saying throughout this chapter, the timing resolution of everything in the system is going to be *no more accurate than* the base timing resolution coming into the system. So the obvious question is, how do you set the base timing resolution? You can use the *ClockPeriod()* function for this:

```
int
ClockPeriod (clockid_t id,
             const struct _clockperiod *new,
             struct _clockperiod *old,
             int reserved);
```

As with the *ClockAdjust()* function described above, the *new* and the *old* parameters are how you get and/or set the values of the base timing resolution.

The *new* and *old* parameters are pointers to structures of `struct _clockperiod`, which contains two members, *nsec* and *fract*. Currently, the *fract* member must be set to zero (it's the number of femtoseconds; we probably won't use this kind of resolution for a little while yet!) The *nsec* member indicates how many nanoseconds elapse between ticks of the base timing clock. The default is 10 milliseconds (1 millisecond on machines with CPU speeds of greater than 40 MHz), so the *nsec* member (if you use the "get" form of the call by specifying the *old* parameter) will show approximately 10 million nanoseconds. (As we discussed above, in *"Clock interrupt sources,"* it's not going to be *exactly* 10 millisecond.)

While you can certainly feel free to try to set the base timing resolution on your system to something ridiculously small, the kernel will step in and prevent you from doing that. Generally, you can set most systems in the 1 millisecond to hundreds of microseconds range.

QNX Neutrino 7.0 and later supports *high-resolution timers*, which give you more precision without requiring that you adjust the clock period. For details, see "Tolerant and high-resolution timers" in the "Understanding the Microkernel's Concept of Time" chapter of the *Programmer's Guide*.

An accurate timestamp

There is one timebase that might be available on your processor that doesn't obey the rules of "base timing resolution" we just described. Some processors have a high-frequency (high-accuracy) counter built right into them, which QNX Neutrino can let you have access to via the *ClockCycles()* call. For example, on a Pentium processor running at 200 MHz, this counter increments at 200 MHz as well, so it can give you timing samples right down to 5 nanoseconds. This is particularly useful if you want to figure out exactly how long a piece of code takes to execute (assuming of course, that you don't get preempted). You'd call *ClockCycles()* before your code and after your code, and then compute the delta. See the QNX Neutrino *C Library Reference* for more details.

In QNX Neutrino 7.0 or later the hardware underlying *ClockCycles()* must be synchronized across all processors on an SMP system. If it isn't, you might encounter some unexpected behavior, such as drifting times and timers. Using synchronized hardware means that you no longer have to use a runmask for threads to prevent them from migrating to other processors between calls to *ClockCycles()*.

Advanced topics

Now that we've seen the basics of timers, we'll look at a few advanced topics:

1. the CLOCK_SOFTTIME and CLOCK_MONOTONIC timer types, and
2. *kernel timeouts*

Other clock sources

We've seen the clock source CLOCK_REALTIME, and mentioned that a POSIX conforming implementation may supply as many different clock sources as it feels like, provided that it at least provides CLOCK_REALTIME.

What is a clock source? Simply put, it's an abstract source of timing information. If you want to put it into real life concepts, your personal watch is a clock source; it measures how fast time goes by. Your watch will have a different level of accuracy than someone else's watch. You may forget to wind your watch, or get it new batteries, and time may seem to "freeze" for a while. Or, you may adjust your watch, and all of a sudden time seems to "jump." These are all characteristics of a clock source.

(QNX Neutrino provides some special clock sources that you can use to track how much CPU time a process or thread uses, but they're beyond the scope of this chapter. See "Monitoring execution times" in the "Understanding the Microkernel's Concept of Time" chapter of the QNX Neutrino *Programmer's Guide*.)

Under QNX Neutrino, CLOCK_REALTIME is based off of the "current time of day" clock that the OS provides. (In the examples below, we refer to this as "QNX Neutrino time.") This means that if the system is running, and suddenly someone adjusts the time forward by 5 seconds, the change may or may not adversely affect your programs (depending on what you're doing). Let's look at a `sleep (30);` call:

Real time	QNX Neutrino time	Activity
11:22:05	11:22:00	`sleep (30);`
11:22:15	11:22:15	Clock gets adjusted to 11:22:15; it was 5 seconds too slow!
11:22:35	11:22:35	`sleep (30);` wakes up

Beautiful! The thread did exactly what you expected: at 11:22:00 it went to sleep for thirty seconds, and at 11:22:35 (thirty elapsed seconds later) it woke up. Notice how the *sleep()* "appeared" to sleep for 35 seconds, instead of 30; in real, elapsed time, though, only 30 seconds went by because QNX Neutrino's clock got adjusted ahead by five seconds (at 11:22:15).

The kernel knows that the *sleep()* call is a relative timer, so it takes care to ensure that the specified amount of "real time" elapses.

Now, what if, on the other hand, we had used an absolute timer, and at 11:22:00 in "QNX Neutrino time" told the kernel to wake us up at 11:22:30?

© 2020, QNX Software Systems Limited

Real time	QNX Neutrino time	Activity
11:22:05	11:22:00	Wake up at 11:22:30
11:22:15	11:22:15	Clock gets adjusted as before
11:22:30	11:22:30	Wakes up

This too is just like what you'd expect—you wanted to be woken up at 11:22:30, and (in spite of adjusting the time) you were. However, in real time, the timer expired after 25 seconds, instead of the full 30. Depending on the application, this could be a problem.

CLOCK_MONOTONIC

The POSIX people thought about this, and the solution they came up with was another clock source called CLOCK_MONOTONIC.

When you create a timer object with *timer_create()*, you can tell it which clock source to use.

The way CLOCK_MONOTONIC works is that its timebase is *never* adjusted. The impact of that is that regardless of what time it is in the real world, if you base a timer on CLOCK_MONOTONIC and add 30 seconds to it (and then do whatever adjustments you want to the time), the timer will expire in 30 elapsed seconds.

The clock source CLOCK_MONOTONIC has the following characteristics:

- always increasing count
- based on *real* time
- starts at zero

The important thing about the clock starting at zero is that this is a different “epoch” (or “base”) than CLOCK_REALTIME's epoch of Jan 1 1970, 00:00:00 GMT. So, even though both clocks run at the same rate, their values are *not* interchangeable.

Functions that don't let you choose the clock source—such as *pthread_mutex_timedlock()*—use CLOCK_REALTIME. This is built in to the function and isn't something that you can change.

So what does CLOCK_SOFTTIME do?

If we wanted to sort our clock sources by “hardness” we'd have the following ordering. You can think of CLOCK_MONOTONIC as being a freight train—it doesn't stop for anyone. Next on the list is CLOCK_REALTIME, because it can be pushed around a bit (as we saw with the time adjustment). Finally, we have CLOCK_SOFTTIME, which we can push around a *lot*.

The main use of CLOCK_SOFTTIME (a QNX Neutrino extension to POSIX) is for things that are “soft”—things that aren't going to cause a critical failure if they don't get done. CLOCK_SOFTTIME is “active” only when the CPU is running. (Yes, this does sound obvious `:-)` but wait!) When the CPU is powered down because power management has detected that nothing is going to happen for a little while, CLOCK_SOFTTIME gets powered down as well!

Here's a timing chart showing the three clock sources:

Real time	QNX Neutrino time	Activity
11:22:05	11:22:00	Wake up at “now” + 00:00:30 (see below)
11:22:15	11:22:15	Clock gets adjusted as before
11:22:20	11:22:20	Power management turns off CPU
11:22:30	11:22:30	CLOCK_REALTIME wakes up
11:22:35	11:22:35	CLOCK_MONOTONIC wakes up
11:45:07	11:45:07	Power management turns on CPU, and CLOCK_SOFTTIME wakes up

There are a few things to note here:

- We precomputed our wakeup time as “now” plus 30 seconds and used an absolute timer to wake us up at the computed time. This is *different* from waking up *in* 30 seconds using a relative timer.
- Note that for convenience of putting the example on one time-line, we've lied a little bit. If the CLOCK_REALTIME thread did indeed wake up, (and later the same for CLOCK_MONOTONIC) it would have caused us to exit out of power management mode at that time, which would then cause CLOCK_SOFTTIME to wake up.

When CLOCK_SOFTTIME “over-sleeps,” it wakes up as soon as it's able—it doesn't stop “timing” while the CPU is powered down, it's just not in a position to wake up until *after* the CPU powers up. Other than that, CLOCK_SOFTTIME is just like CLOCK_REALTIME.

Using different clock sources

To specify one of the different clock source, use a POSIX timing function that accepts a clock ID. For example:

```
#include <time.h>

int
clock_nanosleep (clockid_t clock_id,
                 int flags,
                 const struct timespec *rqtp,
                 struct timespec *rmtp);
```

The *clock_nanosleep()* function accepts the *clock_id* parameter (telling it which clock source to use), a *flag* (which determines if the time is relative or absolute), a “requested sleep time” parameter (*rqtp*), as well as a pointer to an area where the function can fill in the amount of time remaining (in the *rmtp* parameter, which can be NULL if you don't care).

Kernel timeouts

QNX Neutrino lets you have a timeout associated with all kernel blocking states. We talked about the blocking states in the Processes and Threads chapter, in the section “Kernel states.” Most often, you'll

© 2020, QNX Software Systems Limited

want to use this with message passing; a client will send a message to a server, but the client won't want to wait "forever" for the server to respond. In that case, a kernel timeout is suitable. Kernel timeouts are also useful with the *pthread_join()* function. You might want to wait for a thread to finish, but you might not want to wait too long.

Here's the definition for the *TimerTimeout()* function call, which is the kernel function responsible for kernel timeouts:

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

int
TimerTimeout (clockid_t id,
              int flags,
              const struct sigevent *notify,
              const uint64_t *ntime,
              uint64_t *otime);
```

This says that *TimerTimeout()* returns an integer (a pass/fail indication, with -1 meaning the call failed and set *errno*, and zero indicating success). The time source (CLOCK_REALTIME, etc.) is passed in *id*, and the *flags* parameter gives the relevant kernel state or states. The *notify* should always be a notification event of type SIGEV_UNBLOCK, and the *ntime* is the relative time when the kernel call should timeout. The *otime* parameter indicates the previous value of the timeout—it's not used in the vast majority of cases (you can pass NULL).

It's important to note that the timeout is *armed* by *TimerTimeout()*, and *activated* on entry into one of the kernel states specified by *flags*. It is *cleared* upon return from any kernel call. This means that you must re-arm the timeout before *each and every* kernel call that you want to be timeout-aware. You don't have to clear the timeout *after* the kernel call; this is done automagically.

Kernel timeouts with *pthread_join()*

The simplest case to consider is a kernel timeout used with the *pthread_join()* call. Here's how you'd set it up:

```
/*
 * part of tt1.c
*/

#include <sys/neutrino.h>

// 1 billion nanoseconds in a second
#define SEC_NSEC 1000000000LL

int
main (void) // ignore arguments
{
```

```
uint64_t        timeout;
struct sigevent event;
int             rval;

…
// set up the event -- this can be done once

// This or event.sigev_notify = SIGEV_UNBLOCK:
SIGEV_UNBLOCK_INIT (&event);

// set up for 10 second timeout
timeout = 10LL * SEC_NSEC;

TimerTimeout (CLOCK_MONOTONIC, _NTO_TIMEOUT_JOIN,
              &event, &timeout, NULL);

rval = pthread_join (thread_id, NULL);
if (rval == ETIMEDOUT) {
    printf ("Thread %d still running after 10 seconds!\n",
            thread_id);
}
…
```

(You'll find the complete version of *tt1.c* in the Sample Programs appendix.)

We used the *SIGEV_UNBLOCK_INIT()* macro to initialize the event structure, but we could have set the *sigev_notify* member to SIGEV_UNBLOCK ourselves. Even more elegantly, we could pass NULL as the `struct sigevent`—*TimerTimeout()* understands this to mean that it should use a SIGEV_UNBLOCK.

If the thread (specified in *thread_id*) is still running after 10 seconds, then the kernel call will be timed out—*pthread_join()* will return with an *errno* of ETIMEDOUT.

You can use another shortcut—by specifying a NULL for the timeout value (*ntime* in the formal declaration above), this tells the kernel not to block in the given state. This can be used for polling. (While polling is generally discouraged, you could use it quite effectively in the case of the *pthread_join()*—you'd periodically poll to see if the thread you're interested in was finished yet. If not, you could perform other work.)

Here's a code sample showing a non-blocking *pthread_join()*:

```
int
pthread_join_nb (int tid, void **rval)
{
    TimerTimeout (CLOCK_MONOTONIC, _NTO_TIMEOUT_JOIN,
                  NULL, NULL, NULL);
    return (pthread_join (tid, rval));
}
```


© 2020, QNX Software Systems Limited

Kernel timeouts with message passing

Things get a little trickier when you're using kernel timeouts with message passing. Recall from the Message Passing chapter (in the "Message passing and client/server" part) that the server may or may not be waiting for a message when the client sends it. This means that the client could be blocked in either the SEND-blocked state (if the server hasn't received the message yet), or the REPLY-blocked state (if the server has received the message, and hasn't yet replied). The implication here is that you should specify *both* blocking states for the *flags* argument to *TimerTimeout()*, because the client might get blocked in either state.

To specify multiple states, you simply OR them together:

```
TimerTimeout (… _NTO_TIMEOUT_SEND | _NTO_TIMEOUT_REPLY, …);
```

This causes the timeout to be active whenever the kernel enters either the SEND-blocked state or the REPLY-blocked state. There's nothing special about entering the SEND-blocked state and timing out—the server hasn't received the message yet, so the server isn't actively doing anything on behalf of the client. This means that if the kernel times out a SEND-blocked client, the server doesn't have to be informed. The client's *MsgSend()* function returns an ETIMEDOUT indication, and processing has completed for the timeout.

However, as was mentioned in the Message Passing chapter (under "_NTO_CHF_UNBLOCK"), if the server has already received the client's message, and the client wishes to unblock, there are two choices for the server. If the server has *not* specified _NTO_CHF_UNBLOCK on the channel it received the message on, then the client will be unblocked immediately, and the server won't receive any indication that an unblock has occurred. Most of the servers I've seen *always* have the _NTO_CHF_UNBLOCK flag enabled. In that case, the kernel delivers a pulse to the server, *but the client remains blocked until the server replies!* As mentioned in the above-referenced section of the Message Passing chapter, this is done so that the server has an indication that it should *do* something about the client's unblock request.

Summary

We've looked at QNX Neutrino's time-based functions, including timers and how they can be used, as well as kernel timeouts. Relative timers provide some form of event "in a certain number of seconds," while absolute timers provide this event "at a certain time." Timers (and, generally speaking, the `struct sigevent`) can cause the delivery of a pulse, a signal, or a thread to start.

The kernel implements timers by storing the absolute time that represents the next "event" on a sorted queue, and comparing the current time (as derived by the timer tick interrupt service routine) against the head of the sorted queue. When the current time is greater than or equal to the first member of the queue, the queue is processed (for all matching entries) and the kernel dispatches events or threads (depending on the type of queue entry) and (possibly) reschedules.

To provide support for power-saving features, you should disable periodic timers when they're not needed—otherwise, the power-saving feature won't implement power saving, because it believes that there's something to "do" periodically. You could also use the CLOCK_SOFTTIME clock source, unless of course you actually *wanted* the timer to defeat the power saving feature.

Given the different types of clock sources, you have flexibility in determining the basis of your clocks and timer; from “real, elapsed” time through to time sources that are based on power management activities.

© 2020, QNX Software Systems Limited

Chapter 4
Interrupts

In this section, we'll take a look at interrupts, how we deal with them under QNX Neutrino, their impact on scheduling and realtime, and some interrupt-management strategies.

QNX Neutrino and interrupts

The first thing we need to ask is, “What's an interrupt?”

An interrupt is exactly what it sounds like—an interruption of whatever was going on and a diversion to another task.

For example, suppose you're sitting at your desk working on job “A.” Suddenly, the phone rings. A Very Important Customer (VIC) needs you to immediately answer some skill-testing question. When you've answered the question, you may go back to working on job “A,” or the VIC may have changed your priorities so that you push job “A” off to the side and immediately start on job “B.”

Now let's put that into perspective under QNX Neutrino.

At any moment in time, the processor is busy processing the work for the highest-priority READY thread (this will be a thread that's in the RUNNING state). To cause an interrupt, a piece of hardware on the computer's bus asserts an interrupt line (in our analogy, this was the phone ringing).

As soon as the interrupt line is asserted, the kernel jumps to a piece of code that sets up the environment to run an *interrupt service routine* (ISR), a piece of software that determines what should happen when that interrupt is detected.

The amount of time that elapses between the time that the interrupt line is asserted by the hardware and the first instruction of the ISR being executed is called the *interrupt latency*. Interrupt latency is measured in microseconds. Different processors have different interrupt latency times; it's a function of the processor speed, cache architecture, memory speed, and, of course, the efficiency of the operating system.

In our analogy, if you're listening to some music in your headphones and ignoring the ringing phone, it will take you longer to notice this phone “interrupt.” Under QNX Neutrino, the same thing can happen; there's a processor instruction that disables interrupts (`cli` on the x86, for example). The processor won't notice any interrupts until it reenables interrupts (on the x86, this is the `sti` opcode).

To avoid CPU-specific assembly language calls, QNX Neutrino provides the following calls: *InterruptEnable()* and *InterruptDisable()*, and *InterruptLock()* and *InterruptUnlock()*. These take care of all the low-level details on all supported platforms.

The ISR usually performs the minimum amount of work possible, and then ends (in our analogy, this was the conversation on the telephone with the VIC—we usually don't put the customer on hold and do several hours of work; we just tell the customer, “Okay, I'll get right on that!”). When the ISR ends, it can tell the kernel either that nothing should happen (meaning the ISR has completely handled the event and nothing else needs to be done about it) or that the kernel should perform some action that might cause a thread to become READY.

In our analogy, telling the kernel that the interrupt was handled would be like telling the customer the answer—we can return to whatever we were doing, knowing that the customer has had their question answered.

Telling the kernel that some action needs to be performed is like telling the customer that you'll get back to them—the telephone has been hung up, but it could ring again.

© 2020, QNX Software Systems Limited

Interrupt service routine

The ISR is a piece of code that's responsible for clearing the source of the interrupt.

This is a key point, especially in conjunction with this fact: the interrupt runs at a priority *higher than any software priority*. This means that the amount of time spent in the ISR can have a serious impact on thread scheduling. You should spend as little time as possible in the ISR. Let's examine this in a little more depth.

It isn't safe to use floating-point operations in Interrupt Service Routines.

Clearing the interrupt source

The hardware device that generated the interrupt will keep the interrupt line asserted until it's sure the software handled the interrupt. Since the hardware can't read minds, the software must tell it when it has responded to the cause of the interrupt. Generally, this is done by reading a status register from a specific hardware port or a block of data from a specific memory location.

In any event, there's usually some form of positive acknowledgment between the hardware and the software to “de-assert” the interrupt line. (Sometimes there isn't an acknowledgment; for example, a piece of hardware may generate an interrupt and assume that the software will handle it.)

Because the interrupt runs at a higher priority than any software thread, we should spend as little time as possible in the ISR itself to minimize the impact on scheduling. If we clear the source of the interrupt simply by reading a register, and perhaps stuffing that value into a global variable, then our job is simple.

This is the kind of processing done by the ISR for the serial port. The serial port hardware generates an interrupt when a character has arrived. The ISR handler reads a status register containing the character, and stuffs that character into a circular buffer. Done. Total processing time: a few microseconds. And, it *must* be fast. Consider what would happen if you were receiving characters at 115 Kbaud (a character about every 100 µs); if you spent anywhere near 100 µs *handling* the interrupt, you wouldn't have time to do anything else!

Don't let me mislead you though—the serial port's interrupt service routine could take longer to complete. This is because there's a tail-end poll that looks to see if more characters are waiting in the device.

Clearly, minimizing the amount of time spent in the interrupt can be perceived as “Good customer service” in our analogy—by keeping the amount of time that we're on the phone to a minimum, we avoid giving other customers a busy signal.

What if the handler needs to do a significant amount of work? Here are a couple of possibilities:

- The amount of time required to clear the source of the interrupt is short, but the amount of work required to talk to the hardware is long (the customer asked us a short question that takes a long time to answer).
- The amount of time required to clear the source of the interrupt is long (the customer's description of the problem is long and involved).

In the first case, we'd like to clear the source of the interrupt as fast as possible and then tell the kernel to have a thread do the actual work of talking to the slow hardware. The advantage here is that the ISR spends just a tiny amount of time at the super-high priority, and then the rest of the work is done based on regular thread priorities. This is similar to your answering the phone (the super-high priority), and delegating the real work to one of your assistants. We'll look at how the ISR tells the kernel to schedule someone else later in this chapter.

In the second case, things get ugly. If an ISR doesn't clear the source of the interrupt when it exits, the kernel will immediately be re-interrupted by the Programmable Interrupt Controller (PIC—on the x86, this is the 8259 or equivalent) chip.

For PIC fans: we'll talk about edge-sensitive and level-sensitive interrupts shortly.

We'll continuously be running the ISR, without ever getting a chance to run the thread-level code we need to properly handle the interrupt.

What kind of brain-damaged hardware requires a long time to clear the source of the interrupt? Your basic PC floppy disk controller keeps the interrupt asserted until you've read a number of status register values. Unfortunately, the data in the registers isn't available immediately, and you have to poll for this status data. This could take milliseconds (a long time in computer terms)!

The solution to this is to temporarily *mask interrupts*—literally tell the PIC to ignore interrupts from this particular source until you tell it otherwise. In this case, even though the interrupt line is asserted from the hardware, the PIC ignores it and doesn't tell the processor about it. This lets your ISR schedule a thread to handle this hardware outside the ISR. When your thread is finished transferring data from the hardware, it can tell the PIC to *unmask* that interrupt. This lets interrupts from that piece of hardware be recognized again. In our analogy, this is like transferring the VIC's call to your assistant.

Telling a thread to do something

How does an ISR tell the kernel that it should now schedule a thread to do some work? (And conversely, how does it tell the kernel that it *shouldn't* do that?)

Here's some pseudo-code for a typical ISR:

```
FUNCTION ISR BEGIN
    determine source of interrupt
    clear source of interrupt
    IF thread required to do some work THEN
        RETURN (event);
    ELSE
        RETURN (NULL);
    ENDIF
END
```

The trick is to return an event (of type `struct sigevent`, which we talked about in the Clocks, Timers, and Getting a Kick Every So Often chapter) instead of NULL. Note that the event that you return *must* be persistent after the stack frame of the ISR has been destroyed. This means that the event must be declared outside of the ISR, or be passed in from a persistent data area using the *area*

 © 2020, QNX Software Systems Limited

parameter to the ISR, or declared as a `static` within the ISR itself. Your choice. If you return an event, the kernel delivers it to a thread when your ISR returns. Because the event "alerts" a thread (via a pulse, as we talked about in the *Message Passing* chapter, or via a signal), this can cause the kernel to reschedule the thread that gets the CPU next. If you return NULL from the ISR, then the kernel knows that nothing special needs to be done at thread time, so it won't reschedule any threads—the thread that was running at the time that the ISR preempted it resumes running.

Level-sensitivity versus edge-sensitivity

There's one more piece of the puzzle we've been missing. Most PICs can be programmed to operate in *level-sensitive* or *edge-sensitive* mode.

In level-sensitive mode, the interrupt line is deemed to be asserted by the PIC while it's in the "on" state. (This corresponds to label "1" in the diagram below.)

Figure 38: Level-sensitive interrupt assertion.

We can see that this would cause the problem described above with the floppy controller example. Whenever the ISR finishes, the kernel tells the PIC, "Okay, I've handled this interrupt. Tell me the next time that it gets activated" (step 2 in the diagram). In technical terms, the kernel sends an *End Of Interrupt* (EOI) to the PIC. The PIC looks at the interrupt line and if it's still active would immediately re-interrupt the kernel (step 3).

We could get around this by programming the PIC into edge-sensitive mode. In this mode, the interrupt is noticed by the PIC only *on an active-going edge*.

Figure 39: Edge-sensitive interrupt assertion.

Even if the ISR fails to clear the source of the interrupt, when the kernel sends the EOI to the PIC (step 2 in the diagram), the PIC wouldn't re-interrupt the kernel, because there isn't another active-going edge transition after the EOI. In order to recognize another interrupt on that line, the line must first go inactive (step 4), and then active (step 1).

Well, it seems all our problems have been solved! Simply use edge-sensitive for all interrupts.

Unfortunately, edge-sensitive mode has a problem of its own.

Suppose your ISR fails to clear the cause of the interrupt. The hardware would still have the interrupt line asserted when the kernel issues the EOI to the PIC. However, because the PIC is operating in edge-sensitive mode, it never sees another interrupt from that device.

Now what kind of bozo would write an ISR that forgot to clear the source of the interrupt? Unfortunately it isn't that cut-and-dried. Consider a case where two devices (let's say a SCSI bus adapter and an Ethernet card) are sharing the same interrupt line, on a hardware bus architecture that allows that. (Now you're asking, “Who'd set up a machine like that?!?” Well, it happens, especially if the number of interrupt sources on the PIC is in short supply!) In this case, the two ISR routines would be attached to the same interrupt vector (this is legal, by the way), and the kernel would call them in turn whenever it got an interrupt from the PIC for that hardware interrupt level.

Figure 40: Sharing interrupts: one at a time.

In this case, because only one of the hardware devices was active when its associated ISR ran (the SCSI device), it correctly cleared the source of the interrupt (step 2). Note that the kernel runs the ISR for the Ethernet device (in step 3) regardless—it doesn't know whether the Ethernet hardware requires servicing or not as well, so it *always* runs the whole chain.

But consider this case:

Figure 41: Sharing interrupts: several at once.

Here's where the problem lies.

The Ethernet device interrupted first. This caused the interrupt line to be asserted (active-going edge was noted by the PIC), and the kernel called the first interrupt handler in the chain (the SCSI disk driver; step 1 in the diagram). The SCSI disk driver's ISR looked at its hardware and said, “Nope, wasn't me. Oh well, ignore it” (step 2). Then the kernel called the next ISR in the chain, the Ethernet ISR (step 3). The Ethernet ISR looked at the hardware and said, “Hey! That's my hardware that triggered

© 2020, QNX Software Systems Limited

the interrupt. I'm going to clear it.” Unfortunately, while it was clearing it, the SCSI device generated an interrupt (step 4).

When the Ethernet ISR finished clearing the source of the interrupt (step 5), the interrupt line is *still asserted*, thanks to the SCSI hardware device. However, the PIC, being programmed in edge-sensitive mode, is looking for an inactive-to-active transition (on the composite line) before recognizing another interrupt. That isn't going to happen because the kernel has already called both interrupt service routines and is now waiting for another interrupt from the PIC.

In this case, a level-sensitive solution would be appropriate because when the Ethernet ISR finishes and the kernel issues the EOI to the PIC, the PIC would pick up the fact that an interrupt is still active on the bus and re-interrupt the kernel. The kernel would then run through the chain of ISRs, and this time the SCSI driver would get a chance to run and clear the source of the interrupt.

The selection of edge-sensitive versus level-sensitive is something that will depend on the hardware and the startup code. Some hardware will support only one or the other; hardware that supports either mode will be programmed by the startup code to one or the other. You'll have to consult the BSP (Board Support Package) documentation that came with your system to get a definitive answer.

Writing interrupt handlers

Let's see how to set up interrupt handlers—the calls, the characteristics, and some strategies.

Attaching an interrupt handler

To attach to an interrupt source, you'd use either *InterruptAttach()* or *InterruptAttachEvent()*.

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

int
InterruptAttachEvent (int intr,
                      const struct sigevent *event,
                      unsigned flags);

int
InterruptAttach (int intr,
                 const struct sigevent *
                     (*handler) (void *area, int id),
                 const void *area,
                 int size,
                 unsigned flags);
```

The *intr* argument specifies *which* interrupt you wish to attach the specified handler to. The values passed are defined by the startup code that initialized the PIC (amongst other things) just before QNX Neutrino was started. (There's more information on the startup code in your QNX Neutrino documentation; look in the `startup-*` in the *Utilities Reference*.)

Since attaching an interrupt isn't something you want everyone to be able to do, QNX Neutrino allows only processes that have the PROCMGR_AID_INTERRUPT or PROCMGR_AID_INTERRUPTEVENT ability enabled (see *procmgr_ability()* in the *C Library Reference*) to do so.

At this point, the two functions *InterruptAttach()* and *InterruptAttachEvent()* differ. Let's look at *InterruptAttachEvent()* as it's simpler, first. Then we'll come back to *InterruptAttach()*.

Attaching with *InterruptAttachEvent()*

The *InterruptAttachEvent()* function takes two additional arguments: the argument *event*, which is a pointer to the `struct sigevent` that should be delivered, and a *flags* parameter. *InterruptAttachEvent()* tells the kernel that the *event* should be returned whenever the interrupt is detected, and that the interrupt level should be masked off. Note that it's the kernel that interprets the event and figures out which thread should be made READY.

© 2020, QNX Software Systems Limited

Attaching with *InterruptAttach()*

With *InterruptAttach()*, we're specifying a different set of parameters. The *handler* parameter is the address of a function to call. As you can see from the prototype, *handler()* returns a `struct sigevent`, which indicates what kind of an event to return, and takes two parameters. The first passed parameter is the *area*, which is simply the *area* parameter that's passed to *InterruptAttach()* to begin with. The second parameter, *id*, is the identification of the interrupt, which is also the return value from *InterruptAttach()*. This is used to identify the interrupt and to mask, unmask, lock, or unlock the interrupt. The fourth parameter to *InterruptAttach()* is the *size*, which indicates how big (in bytes) the data area that you passed in *area* is. Finally, the *flags* parameter is the same as that passed for the *InterruptAttachEvent()*; we'll discuss that shortly.

Now that you've attached an interrupt

At this point, you've called either *InterruptAttachEvent()* or *InterruptAttach()*.

Here's a code snippet that attaches an ISR to the hardware interrupt vector, which we've identified in our code sample by the constant HW_SERIAL_IRQ:

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

int interruptID;

const struct sigevent *
intHandler (void *arg, int id)
{
    ...
}

int
main (int argc, char **argv)
{
    ...
    interruptID = InterruptAttach (HW_SERIAL_IRQ,
                                   intHandler,
                                   &event,
                                   sizeof (event),
                                   0);
    if (interruptID == -1) {
        fprintf (stderr, "%s:  can't attach to IRQ %d\n",
                 progname, HW_SERIAL_IRQ);
        perror (NULL);
        exit (EXIT_FAILURE);
    }
    ...
```

```
    return (EXIT_SUCCESS);
}
```

This creates the association between the ISR (the routine called *intHandler()*; see below for details) and the hardware interrupt vector HW_SERIAL_IRQ.

At this point, if an interrupt occurs on that interrupt vector, our ISR will be dispatched. When we call *InterruptAttach()*, the kernel unmasks the interrupt source at the PIC level (unless it's already unmasked, which would be the case if multiple ISRs were sharing the same interrupt).

Detaching an interrupt handler

When done with the ISR, we *may* wish to break the association between the ISR and the interrupt vector:

```
int
InterruptDetach (int id);
```

I said “may” because threads that handle interrupts are generally found in servers, and servers generally hang around forever. It's therefore conceivable that a well-constructed server wouldn't ever issue the *InterruptDetach()* function call. Also, the OS will remove any interrupt handlers that a thread or process may have associated with it when the thread or process dies. So, simply falling off the end of *main()*, calling *exit()*, or exiting due to a SIGSEGV, will dissociate your ISR from the interrupt vector, automagically. (Of course, you'll probably want to handle this a little better, and stop your device from generating interrupts. If another device is sharing the interrupt, then there are no two ways about it—you *must* clean up, otherwise you won't get any more interrupts if running edge-sensitive mode, or you'll get a constant flood of ISR dispatches if running in level-sensitive mode.)

Continuing the above example, if we want to detach, we'd use the following code:

```
void
terminateInterrupts (void)
{
    InterruptDetach (interruptID);
}
```

If this was the last ISR associated with that interrupt vector, the kernel would automatically mask the interrupt source at the PIC level so that it doesn't generate interrupts.

The *flags* parameter

The last parameter, *flags*, controls all kinds of things:

_NTO_INTR_FLAGS_END

Indicates that this handler should go *after* other handlers that may be attached to the same interrupt source.

_NTO_INTR_FLAGS_NO_UNMASK

(QNX Neutrino 6.6 or later) Indicates that the kernel should leave the interrupt masked. Normally, *InterruptAttach()* and *InterruptAttachEvent()* automatically unmask an interrupt

 © 2020, QNX Software Systems Limited

the first time something is attached to it. If you specify _NTO_INTR_FLAGS_NO_UNMASK, the kernel leaves the interrupt masked, and you must specifically call *InterruptUnmask()* to enable it.

_NTO_INTR_FLAGS_PROCESS

Indicates that this handler is associated with the process rather than the thread. What this boils down to is that if you specify this flag, the interrupt handler will be automatically dissociated from the interrupt source when the process exits. If you don't specify this flag, the interrupt handler will be dissociated from the interrupt source when the thread that created the association in the first place exits.

_NTO_INTR_FLAGS_TRK_MSK

Indicates that the kernel should track the number of times the interrupt has been masked. This causes a little more work for the kernel, but is required to ensure an orderly unmasking of the interrupt source should the process or thread exit.

_NTO_INTR_FLAGS_EXCLUSIVE

(QNX Neutrino 7.0.4 or later) Indicates that the interrupt handler or event currently attached to the interrupt vector is the only one that can be attached.

The interrupt service routine

Let's look at the ISR itself. In the first example, we'll look at using the *InterruptAttach()* function. Then, we'll see the exact same thing, except with *InterruptAttachEvent()*.

Using *InterruptAttach()*

Continuing our example, here's the ISR *intHandler()*. It looks at the 8250 serial port chip that we assume is attached to HW_SERIAL_IRQ:

```
/*
 * int1.c
*/

#include <stdio.h>
#include <sys/neutrino.h>

#define REG_RX              0
#define REG_II              2
#define REG_LS              5
#define REG_MS              6
#define IIR_MASK            0x07
#define IIR_MSR             0x00
#define IIR_THE             0x02
#define IIR_RX              0x04
#define IIR_LSR             0x06
#define IIR_MASK            0x07

volatile int serial_msr;   // saved contents of Modem Status Reg
volatile int serial_rx;    // saved contents of RX register
```

```
volatile int serial_lsr;   // saved contents of Line Status Reg
static int base_reg = 0x2f8;

const struct sigevent *
intHandler (void *arg, int id)
{
    int  iir;
    struct sigevent *event = (struct sigevent *)arg;

    /*
     * determine the source of the interrupt
     * by reading the Interrupt Identification Register
    */

    iir = in8 (base_reg + REG_II) & IIR_MASK;

    /* no interrupt? */
    if (iir & 1) {
        /* then no event */
        return (NULL);
    }

    /*
     * figure out which interrupt source caused the interrupt,
     * and determine if a thread needs to do something about it.
     * (The constants are based on the 8250 serial port's interrupt
     * identification register.)
    */

    switch (iir) {
    case    IIR_MSR:
        serial_msr = in8 (base_reg + REG_MS);

        /* wake up thread */
        return (event);
        break;

    case    IIR_THE:
        /* do nothing */
        break;

    case    IIR_RX:
        /* note the character */
        serial_rx = in8 (base_reg + REG_RX);
        break;

    case    IIR_LSR:
        /* note the line status reg. */
        serial_lsr = in8 (base_reg + REG_LS);
        break;

    default:
        break;
    }
```


© 2020, QNX Software Systems Limited

```
    /* don't bother anyone */
    return (NULL);
}
```

The first thing we notice is that any variable that the ISR touches must be declared `volatile`. On a single-processor box, this isn't for the ISR's benefit, but rather for the benefit of the thread-level code, which can be interrupted at any point by the ISR. Of course, on an SMP box, we *could* have the ISR running concurrently with the thread-level code, in which case we have to be *very* careful about these sorts of things.

With the `volatile` keyword, we're telling the compiler not to cache the value of any of these variables, because they can change at any point during execution.

The next thing we notice is the prototype for the interrupt service routine itself. It's marked as `const struct sigevent *`. This says that the routine *intHandler()* returns a `struct sigevent` pointer. This is standard for all interrupt service routines.

Finally, notice that the ISR decides if the thread will or won't be sent an event. Only in the case of a Modem Status Register (MSR) interrupt do we want the event to be delivered (the event is identified by the variable *event*, which was conveniently passed to the ISR when we attached it). In all other cases, we ignore the interrupt (and update some global variables). In *all* cases, however, we clear the source of the interrupt. This is done by reading the I/O port via *in8()*.

Using *InterruptAttachEvent()*

If we were to recode the example above to use *InterruptAttachEvent()*, it would look like this:

```
/*
 * part of int2.c
*/

#include <stdio.h>
#include <sys/neutrino.h>

#define HW_SERIAL_IRQ      3
#define REG_RX             0
#define REG_II             2
#define REG_LS             5
#define REG_MS             6
#define IIR_MASK           0x07
#define IIR_MSR            0x00
#define IIR_THE            0x02
#define IIR_RX             0x04
#define IIR_LSR            0x06
#define IIR_MASK           0x07

static int base_reg = 0x2f8;

int
main (int argc, char **argv)
{
    int  intId;         // interrupt id
    int  iir;           // interrupt identification register
```

```
int  serial_msr;    // saved contents of Modem Status Reg
int  serial_rx;     // saved contents of RX register
int  serial_lsr;    // saved contents of Line Status Reg
struct sigevent event;

// usual main() setup stuff...

// set up the event
intId = InterruptAttachEvent (HW_SERIAL_IRQ, &event, 0);

for (;;) {

    // wait for an interrupt event (could use MsgReceive instead)
    InterruptWait (0, NULL);

    /*
     * determine the source of the interrupt (and clear it)
     * by reading the Interrupt Identification Register
    */

    iir = in8 (base_reg + REG_II) & IIR_MASK;

    // unmask the interrupt, so we can get the next event
    InterruptUnmask (HW_SERIAL_IRQ, intId);

    /* no interrupt? */
    if (iir & 1) {
        /* then wait again for next */
        continue;
    }

    /*
     * figure out which interrupt source caused the interrupt,
     * and determine if we need to do something about it
    */

    switch (iir) {
    case    IIR_MSR:
        serial_msr = in8 (base_reg + REG_MS);

        /*
         * perform whatever processing you would've done in
         * the other example...
        */
        break;

    case    IIR_THE:
        /* do nothing */
        break;

    case    IIR_RX:
        /* note the character */
        serial_rx = in8 (base_reg + REG_RX);
        break;
```


© 2020, QNX Software Systems Limited

```
        case    IIR_LSR:
            /* note the line status reg. */
            serial_lsr = in8 (base_reg + REG_LS);
            break;
        }
    }

    /* You won't get here. */
    return (0);
}
```

Notice that the *InterruptAttachEvent()* function returns an interrupt identifier (a small integer). We've saved this into the variable *intId* so that we can use it later when we go to unmask the interrupt.

After we've attached the interrupt, we then need to wait for the interrupt to hit. Since we're using *InterruptAttachEvent()*, we'll get the event that we created earlier dropped on us *for every interrupt*. Contrast this with what happened when we used *InterruptAttach()*—in that case, our ISR determined whether or not to drop an event on us. With *InterruptAttachEvent()*, the kernel has no idea whether or not the hardware event that caused the interrupt was "significant" for us, so it drops the event on us every time it occurs, masks the interrupt, and lets us decide if the interrupt was significant or not.

We handled the decision in the code example for *InterruptAttach()* (above) by returning either a `struct sigevent` to indicate that something should happen, or by returning the constant NULL. Notice the changes that we did to our code when we modified it for *InterruptAttachEvent()*:

- The "ISR" work is now done at thread time in *main()*.
- We must *always* unmask the interrupt source after receiving our event (because the kernel masks it for us).
- If the interrupt is not significant to us, we don't do anything and simply loop around again in the `for` statement, waiting for another interrupt.
- If the interrupt *is* significant to us, we handle it directly (in the `case IIR_MSR` part).

Where you decide to clear the source of the interrupt depends on your hardware and the notification scheme you've chosen. With the combination of SIGEV_INTR and *InterruptWait()*, the kernel doesn't "queue" more than one notification; with SIGEV_PULSE and *MsgReceive()*, the kernel will queue all the notifications. If you're using signals (and SIGEV_SIGNAL, for example), you define whether the signals are queued or not. With some hardware schemes, you may need to clear the source of the interrupt *before* you can read more data out of the device; with other pieces of hardware, you don't have to and can read data while the interrupt is asserted.

An ISR returning SIGEV_THREAD is one scenario that fills me with absolute fear! I'd recommend avoiding this "feature" if at all possible.

(QNX Neutrino 7.0.4 or later) In order to register or use an event of type SIGEV_THREAD, your process must have the PROCMGR_AID_SIGEV_THREAD ability enabled. For more information, see *procmgr_ability()*.

In the serial port example above, we've decided to use *InterruptWait()*, which will queue one entry. The serial port hardware may assert another interrupt immediately after we've read the interrupt

identification register, but that's fine, because at most one SIGEV_INTR will get queued. We'll pick up this notification on our next iteration of the `for` loop.

InterruptAttach() versus *InterruptAttachEvent()*

This naturally brings us to the question, Why would I use one over the other?

The most obvious advantage of *InterruptAttachEvent()* is that it's simpler to use than *InterruptAttach()*—there's no ISR routine (hence no need to debug it). Another advantage is that since there's nothing running in kernel space (as an ISR routine would be) there's no danger of crashing the entire system. If you do encounter a programming error, then the process will crash, rather than the whole system. However, it may be more or less efficient than *InterruptAttach()* depending on what you're trying to achieve. This issue is complex enough that reducing it to a few words (like “faster” or “better”) probably won't suffice. We'll need to look at a few pictures and scenarios.

Here's what happens when we use *InterruptAttach()*:

Figure 42: Control flow with *InterruptAttach()*.

The thread that's currently running (“thread1”) gets interrupted, and we go into the kernel. The kernel saves the context of “thread1.” The kernel then does a lookup to see who's responsible for handling the interrupt and decides that “ISR1” is responsible. At this point, the kernel sets up the context for “ISR1” and transfers control. “ISR1” looks at the hardware and decides to return a `struct sigevent`. The kernel notices the return value, figures out who needs to handle it, and makes them READY. This may cause the kernel to schedule a different thread to run, “thread2.”

Now, let's contrast that with what happens when we use *InterruptAttachEvent()*:

Figure 43: Control flow with *InterruptAttachEvent()*.

© 2020, QNX Software Systems Limited

In this case, the servicing path is much shorter. We made one context switch from the currently running thread ("thread1") into the kernel. Instead of doing another context switch into the ISR, the kernel simply "pretended" that the ISR returned a `struct sigevent` and acted on it, rescheduling "thread2" to run.

Now you're thinking, "Great! I'm going to forget all about *InterruptAttach()* and just use the easier *InterruptAttachEvent()*."

That's not such a great idea, because you may not *need* to wake up for every interrupt that the hardware generates! Go back and look at the source example above—it returned an event only when the modem status register on the serial port changed state, not when a character arrived, not when a line status register changed, and not when the transmit holding buffer was empty.

In that case, especially if the serial port was receiving characters (that you wanted to ignore), you'd be wasting a lot of time rescheduling your thread to run, only to have it look at the serial port and decide that it didn't want to do anything about it anyway. In that case, things would look like this:

Figure 44: Control flow with *InterruptAttachEvent()* and unnecessary rescheduling.

All that happens is that you incur a thread-to-thread context switch to get into "thread2" which looks at the hardware and decides that it doesn't need to do anything about it, costing you another thread-to-thread context switch to get back to "thread1."

Here's how things would look if you used *InterruptAttach()* but *didn't* want to schedule a different thread (i.e., you returned):

Figure 45: Control flow with *InterruptAttach()* with no thread rescheduling.

The kernel knows that “thread1” was running, and the ISR didn't tell it to do anything, so it can just go right ahead and let “thread1” continue after the interrupt.

Just for reference, here's what the *InterruptAttachEvent()* function call does (note that this isn't the real source, because *InterruptAttachEvent()* actually binds a data structure to the kernel—it *isn't* implemented as a discrete function that gets called!):

```
// the "internal" handler
static const struct sigevent *
internalHandler (void *arg, int id)
{
    struct sigevent *event = arg;

    InterruptMask (intr, id);
    return (arg);
}

int
InterruptAttachEvent (int intr,
    const struct sigevent *event, unsigned flags)
{
    static struct sigevent static_event;

    memcpy (&static_event, event, sizeof (static_event));

    return (InterruptAttach (intr, internalHandler,
            &static_event, sizeof (*event), flags));
}
```

The tradeoffs

So, which function should you use? For low-frequency interrupts, you can almost always get away with *InterruptAttachEvent()*. Since the interrupts occur infrequently, there won't be a significant impact on

© 2020, QNX Software Systems Limited

overall system performance, even if you do schedule threads unnecessarily. The only time that this can come back to haunt you is if another device is chained off the same interrupt—in this case, because *InterruptAttachEvent()* masks the source of the interrupt, it'll effectively disable interrupts from the other device until the interrupt source is unmasked. This is a concern only if the first device takes a long time to be serviced. In the bigger picture, this is a hardware system design issue—you shouldn't chain slow-to-respond devices on the same line as high-speed devices.

For higher-frequency interrupts, it's a toss up, and there are many factors:

- Unnecessary interrupts—if there will be a significant number of these, you're better off using *InterruptAttach()* and filtering them out in the ISR. For example, consider the case of a serial device. A thread may issue a command saying "Get me 64 bytes." If the ISR is programmed with the knowledge that nothing useful will happen until 64 bytes are received from the hardware, the ISR has effectively filtered the interrupts. The ISR will then return an event only *after* 64 bytes have been accumulated.
- Latency—if your hardware is sensitive to the amount of time that passes between asserting the interrupt request and the execution of the ISR, you should use *InterruptAttach()* to minimize this interrupt latency. This is because the kernel is very fast at dispatching the ISR.
- Buffering—if your hardware has buffering in it, you may be able to get away with *InterruptAttachEvent()* and a single-entry queueing mechanism like SIGEV_INTR and *InterruptWait()*. This method lets the hardware interrupt as often as it wants, while letting your thread pick the values out of the hardware's buffer when it can. Since the hardware is buffering the data, there's no problem with interrupt latencies.

ISR functions

The next issue we should tackle is the list of functions an ISR is allowed to call.

Let me digress just a little at this point. Historically, the reason that ISRs were so difficult to write (and still are in most other operating systems) is that the ISR runs in a special environment.

One particular thing that complicates writing ISRs is that the ISR isn't actually a "proper" thread as far as the kernel is concerned. It's this weird "hardware" thread, if you want to call it that. This means that the ISR isn't allowed to do any "thread-level" things, like messaging, synchronization, kernel calls, disk I/O, etc.

But doesn't that make it much *harder* to write ISR routines? Yes it does. The solution, therefore, is to do as little work as possible in the ISR, and do the rest of the work at thread-level, where you have access to all the services.

Your goals in the ISR should be:

- Fetch information that is transitory.
- Clear the source of the ISR.
- Optionally dispatch a thread to get the "real" work done.

This "architecture" hinges on the fact that QNX Neutrino has very fast context-switch times. You know that you can get into your ISR quickly to do the time-critical work. You also know that when the ISR returns an event to trigger thread-level work, that thread will start quickly as well. It's this "don't do anything in the ISR" philosophy that makes QNX Neutrino ISRs so simple!

So, what calls can you use in the ISR? Here's a summary (for the official list, see the Full Safety Information appendix in the QNX Neutrino *C Library Reference*):

- *atomic_*()* functions (such as *atomic_set()*)
- *mem*()* functions (such as *memcpy()*)
- most *str*()* functions (such as *strcmp()*). Beware, though, that not all these are safe, such as *strdup()*—it calls *malloc()*, which uses a mutex, and that's not allowed. For the string functions, you should really consult the individual entries in the QNX Neutrino *C Library Reference* before using.
- *InterruptMask()*
- *InterruptUnmask()*
- *InterruptLock()*
- *InterruptUnlock()*
- *InterruptDisable()*
- *InterruptEnable()*
- *in*()* and *out*()*

Basically, the rule of thumb is, "Don't use anything that's going to take a huge amount of stack space or time, and don't use anything that issues kernel calls." The stack space requirement stems from the fact that ISRs have very limited stacks.

The list of interrupt-safe functions makes sense—you might want to move some memory around, in which case the *mem*()* and *str*()* functions are a good choice. You'll most likely want to read data registers from the hardware (in order to save transitory data variables and/or clear the source of the interrupt), so you'll want to use the *in*()* and *out*()* functions.

What about the bewildering choice of *Interrupt*()* functions? Let's examine them in pairs:

InterruptMask()* and *InterruptUnmask()

These functions are responsible for masking the interrupt source at the PIC level; this keeps them from being passed on to the CPU. Generally, you'd use this if you want to perform further work in the thread and can't clear the source of the interrupt in the ISR itself. In this case, the ISR would issue *InterruptMask()*, and the thread would issue *InterruptUnmask()* when it had completed whatever operation it was invoked to do.

Keep in mind that *InterruptMask()* and *InterruptUnmask()* are *counting*—you must "unmask" the same number of times that you've "masked" in order for the interrupt source to be able to interrupt you again.

By the way, note that the *InterruptAttachEvent()* performs the *InterruptMask()* for you (in the kernel)—therefore you must call *InterruptUnmask()* from your interrupt-handling thread.

InterruptLock()* and *InterruptUnlock()

These functions are used to disable (*InterruptLock()*) and enable (*InterruptUnlock()*) interrupts on a single- or multiprocessor system. You'd want to disable interrupts if you needed to protect the thread from the ISR (or additionally, on an SMP system, the ISR from a thread). Once you've done your critical data manipulation, you'd then enable interrupts. Note that these functions are recommended over the "old" *InterruptDisable()* and

© 2020, QNX Software Systems Limited

InterruptEnable() functions as they will operate properly on an SMP system. There's an additional cost over the “old” functions to perform the check on an SMP system, but in a single-processor system it's negligible, which is why I'm recommending that you always use *InterruptLock()* and *InterruptUnlock()*.

InterruptDisable()* and *InterruptEnable()

These functions shouldn't be used in new designs. Historically, they were used to invoke the x86 processor instructions `cli` and `sti` when QNX Neutrino was x86-only. They've since been upgraded to handle all supported processors, but you should use *InterruptLock()* and *InterruptUnlock()* (to keep SMP systems happy).

The one thing that bears repeating is that on an SMP system, *it is possible* to have *both* the interrupt service routine *and* another thread running *at the same time*.

Summary

Keep the following things in mind when dealing with interrupts:

- Don't take too long in an ISR—perform the minimum amount of work you can get away with. This helps minimize interrupt latency and debugging.
- Use *InterruptAttach()* when you *need* to access the hardware as soon as the interrupt occurs; otherwise, avoid it.
- Use *InterruptAttachEvent()* at all other times. The kernel will schedule a thread (based on the event that you passed) to handle the interrupt.
- Protect variables used by both the interrupt service routine (if using *InterruptAttach()*) and threads by calling *InterruptLock()* and *InterruptUnlock()*.
- Declare variables that are going to be used between the thread and the ISR as `volatile` so that the compiler isn't caching “stale” values that have been changed by the ISR.

© 2020, QNX Software Systems Limited

Chapter 5
Resource Managers

In this chapter, we'll take a look at what you need to understand in order to write a *resource manager*.

What is a resource manager?

A resource manager is simply a program with some well-defined characteristics.

This program is called different things on different operating systems—some call them “device drivers,” “I/O managers,” “filesystems,” “drivers,” “devices,” and so on. In all cases, however, the goal of this program (which we'll just call a resource manager) is to present an abstract view of some service.

Also, since QNX Neutrino is a POSIX-conforming operating system, it turns out that the abstraction is based on the POSIX specification.

Examples of resource managers

Before we get carried away, let's take a look at a couple of examples and see how they “abstract” some “service.” We'll look at an actual piece of hardware (a serial port) and something much more abstract (a filesystem).

Serial port

On a typical system, there usually exists some way for a program to transmit output and receive input from a serial, RS-232-style hardware interface. This hardware interface consists of a bunch of hardware devices, including a *UART* (*U*niversal *A*synchronous *R*eceiver *T*ransmitter) chip which knows how to convert the CPU's parallel data stream into a serial data stream and vice versa.

In this case, the “service” being provided by the serial resource manager is the capability for a program to send and receive characters on a serial port.

We say that an “abstraction” occurs, because the client program (the one ultimately using the service) doesn't know (nor does it care about) the details of the UART chip and its implementation. All the client program knows is that to send some characters it should call the *fprintf()* function, and to receive some characters it should call the *fgets()* function. Notice that we used standard, POSIX function calls to interact with the serial port.

Filesystem

As another example of a resource manager, let's examine the filesystem. This consists of a number of cooperating modules: the filesystem itself, the block I/O driver, and the disk driver.

The “service” being offered here is the capability for a program to read and write characters on some medium. The “abstraction” that occurs is the same as with the serial port example above—the client program can still use *the exact same function calls* (e.g., the *fprintf()* and *fgets()* functions) to interact with a storage medium instead of a serial port. In fact, the client really doesn't know or need to know which resource manager it's interacting with.

Characteristics of resource managers

As we saw in our examples (above), the key to the flexibility of the resource managers is that all the functionality of the resource manager is accessed using standard POSIX function calls—we didn't use

 © 2020, QNX Software Systems Limited

“special” functions when talking to the serial port. But what if you *need* to do something “special,” something *very* device-specific?

For example, setting the baud rate on a serial port is an operation that's very specific to the serial port resource manager—it's totally meaningless to the filesystem resource manager. Likewise, setting the file position via *lseek()* is useful in a filesystem, but meaningless in a serial port. The solution POSIX chose for this is simple. Some functions, like *lseek()*, simply return an error code on a device that doesn't support them. Then there's the “catch-all” (and non-POSIX) device control function, called *devctl()*, that allows device-specific functionality to be provided. Devices that don't understand the particular *devctl()* command simply return an error, just as devices that don't understand the *lseek()* command would.

Since we've mentioned *lseek()* and *devctl()* as two common commands, it's worthwhile to note that pretty much *all* file-descriptor (or `FILE *` stream) function calls are supported by resource managers.

This naturally leads us to the conclusion that resource managers will be dealing almost exclusively with file-descriptor based function calls. Since QNX Neutrino is a message-passing operating system, it follows that the POSIX functions get translated into messages, which are then sent to resource managers. It is this “POSIX-function to message-passing” translation trick that lets us decouple clients from resource managers. All a resource manager has to do is handle certain well-defined messages. All a client has to do is generate the same well-defined messages that the resource manager is expecting to receive and handle.

Since the interaction between clients and resource managers is based on message passing, it makes sense to make this “translation layer” as thin as possible. For example, when a client does an *open()* and gets back a file descriptor, the *file descriptor is in fact the connection ID*! This connection ID (file descriptor) gets used in the client's C library functions (such as *read()*) where a message is created and sent to the resource manager.

The client's view

We've already seen a hint of what the client expects. It expects a file-descriptor-based interface, using standard POSIX functions.

In reality, though, there are a few more things going on “under the hood.”

For example, how does the client actually connect to the appropriate resource manager? What happens in the case of union filesystems (where multiple filesystems are responsible for the same “namespace”)? How are directories handled?

Finding the server

The first thing that a client does is call *open()* to get a file descriptor. (Note that if the client calls the higher-level function *fopen()* instead, the same discussion applies—*fopen()* eventually calls *open()*).

Inside the C library implementation of *open()*, a message is constructed, and sent to the process manager (`procnto`) component. The process manager is responsible for maintaining information about the *pathname space*. This information consists of a tree structure that contains pathnames and node descriptor, process ID, channel ID, and handle associations:

Figure 46: QNX Neutrino's namespace.

Note that in the diagram above and in the descriptions that follow, I've used the designation `fs-qnx6` as the name of the resource manager that implements the Power-Safe filesystem—in reality, it's a bit more complicated, because the filesystem drivers are based on a series of DLLs that get bundled together. So, there's actually no executable called `fs-qnx6`; we're just using it as a placeholder for the filesystem component.

Let's say that the client calls *open()*:

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

In the client's C library implementation of *open()*, a message is constructed and sent to the process manager. This message states, “I want to open **/dev/ser1**; who should I talk to?”

© 2020, QNX Software Systems Limited

Figure 47: First stage of name resolution.

The process manager receives the request and looks through its tree structure to see if there's a match (let's assume for now that we need an exact match). Sure enough, the pathname "**/dev/ser1**" matches the request, and the process manager is able to reply to the client: "I found **/dev/ser1**. It's being handled by node descriptor 0, process ID 44, channel ID 1, handle 1. Send them your request!"

Remember, we're still in the client's *open()* code!

So, the *open()* function creates another message, and a connection to the specified node descriptor (0, meaning our node), process ID (44), channel ID (1), stuffing the handle into the message itself. This message is really the "connect" message—it's the message that the client's *open()* library uses to establish a connection to a resource manager (step 3 in the picture below). When the resource manager gets the connect message, it looks at it and performs validation. For example, you may have tried to open-for-write a resource manager that implements a read-only filesystem, in which case you'd get back an error (in this case, EROFS). In our example, however, the serial port resource manager looks at the request (we specified O_WRONLY; perfectly legal for a serial port) and replies back with an EOK (step 4 in the picture below).

Figure 48: The _IO_CONNECT message.

Finally, the client's *open()* returns to the client with a valid file descriptor.

Really, this file descriptor is the connection ID we just used to send a connect message to the resource manager! Had the resource manager *not* given us an EOK, we would have passed this error back to the client (via *errno* and a -1 return from *open()*). (It's worthwhile to note that the process manager can return the node ID, process ID and channel ID of *more than one* resource manager in response to a name resolution request. In that case, the client will try each of them in turn until one succeeds, returns an error that's not ENOSYS, ENOENT, or EROFS, or the client exhausts the list, in which case the *open()* fails. We'll discuss this further when we look at the "before" and "after" flags, later on.)

Finding the process manager

Now that we understand the basic steps used to find a particular resource manager, we need to solve the mystery of, "How did we find the process manager to begin with?"

Actually, this one's easy. By definition, the process manager has a node descriptor of 0 (meaning this node), a process ID of 1, and a channel ID of 1. So, the ND/PID/CHID triplet 0/1/1 always identifies the process manager.

Handling directories

The example we used above was that of a serial port resource manager. We also stated an assumption: “Let's assume for now that we need an exact match.” The assumption is only half-true—all the pathname matching we'll be talking about in this chapter has to *completely* match a *component* of the pathname, but may not have to match the *entire* pathname. We'll clear this up shortly.

Suppose I had code that does this:

```
fp = fopen ("/etc/passwd", "r");
```

Recall that *fopen()* eventually calls *open()*, so we have *open()* asking about the pathname **/etc/passwd**. But there isn't one in the diagram:

Figure 49: QNX Neutrino's namespace.

We do notice, however, that `fs-qnx6` has registered its association of ND/PID/CHID at the pathname “**/**.” Although it's not shown on the diagram, `fs-qnx6` registered itself as a *directory resource manager*—it told the process manager that it'll be responsible for “**/**” *and below*. This is something that the other, “device” resource managers (e.g., the serial port resource manager) didn't do. By setting the “directory” flag, `fs-qnx6` is able to handle the request for “**/etc/passwd**” because the first part of the request is “**/**”—a matching component!

What if we tried to do the following?

```
fd = open ("/dev/ser1/9600.8.1.n", O_WRONLY);
```

Well, since the serial port resource manager doesn't have the directory flag set, the process manager will look at it and say “Nope, sorry, the pathname **/dev/ser1** is not a directory. I'm going to have to fail this request.” The request fails right then and there—the process manager doesn't even return a ND/PID/CHID/handle that the *open()* function should try.

Obviously, as hinted at in my choice of parameters for the *open()* call above, it *may* be a clever idea to allow some “traditional” drivers to be opened with additional parameters past the “usual” name. However, the rule of thumb here is, “If you can get away with it in a design review meeting, knock yourself out.” Some of my students, upon hearing me say that, pipe up with “But I *am* the design review committee!” To which I usually reply, “You are given a gun. Shoot yourself in the foot. `:-)`”

© 2020, QNX Software Systems Limited

Unioned filesystems

Take a closer look at the diagram we've been using:

Figure 50: QNX Neutrino's namespace.

Notice how *both* `fs-qnx6` *and* the process manager have registered themselves as being responsible for "**/**"? This is fine, and nothing to worry about. In fact, there are times when it's a very good idea. Let's consider one such case.

Suppose you have a very slow network connection and you've mounted a networked filesystem over it. You notice that you often use certain files and wish that they were somehow magically "cached" on your system, but alas, the designers of the network filesystem didn't provide a way for you to do that. So, you write yourself a pass-through filesystem (called `fs-cache`) that sits on top of the network filesystem. Here's how it looks from the client's point of view:

Figure 51: Overlaid filesystems.

Both `fs-nfs` (the network filesystem) and your caching filesystem (`fs-cache`) have registered themselves for the same prefix, namely "**/nfs**." As we mentioned above, this is fine, normal, and legal under QNX Neutrino.

Let's say that the system just started up and your caching filesystem doesn't have anything in it yet. A client program tries to open a file, let's say **/nfs/home/rk/abc.txt**. Your caching filesystem is "in front of" the network filesystem (I'll show you how to do that later, when we discuss resource manager implementation).

At this point, the client's *open()* code does the usual steps:

1. Message to the process manager: "Whom should I talk to about the filename **/nfs/home/rk/abc.txt**?"
2. Response from the process manager: "Talk to `fs-cache` first, and then `fs-nfs`."

Notice here that the process manager returned *two* sets of ND/PID/CHID/handle; one for `fs-cache` and one for `fs-nfs`. This is critical.

Now, the client's *open()* continues:

1. Message to `fs-cache`: “I'd like to open the file **/nfs/home/rk/abc.txt** for read, please.”
2. Response from `fs-cache`: “Sorry, I've never heard of this file.”

At this point, the client's *open()* function is out of luck as far as the `fs-cache` resource manager is concerned. The file doesn't exist! However, the *open()* function knows that it got a list of *two* ND/PID/CHID/handle tuples, so it tries the second one next:

1. Message to `fs-nfs`: “I'd like to open the file **/nfs/home/rk/abc.txt** for read, please.”
2. Response from `fs-nfs`: “Sure, no problem!”

Now that the *open()* function has an EOK (the “no problem”), it returns the file descriptor. The client then performs *all* further interactions with the `fs-nfs` resource manager.

The *only* time that we “resolve” to a resource manager is during the *open()* call. This means that once we've successfully opened a particular resource manager, we will continue to use that resource manager for all file descriptor calls.

So how does our `fs-cache` caching filesystem come into play? Well, eventually, let's say that the user is done reading the file (they've loaded it into a text editor). Now they want to write it out. The same set of steps happen, with an interesting twist:

1. Message to the process manager: “Whom should I talk to about the filename **/nfs/home/rk/abc.txt**?”
2. Response from the process manager: “Talk to `fs-cache` first, and then `fs-nfs`.”
3. Message to `fs-cache`: “I'd like to open the file **/nfs/home/rk/abc.txt** for *write*, please.”
4. Response from `fs-cache`: “Sure, no problem.”

Notice that this time, in step 3, we opened the file for *write* and not *read* as we did previously. It's not surprising, therefore, that `fs-cache` allowed the operation this time (in step 4).

Even more interesting, observe what happens the *next* time we go to read the file:

1. Message to the process manager: “Whom should I talk to about the filename **/nfs/home/rk/abc.txt**?”
2. Response from the process manager: “Talk to `fs-cache` first, and then `fs-nfs`.”
3. Message to `fs-cache`: “I'd like to open the file **/nfs/home/rk/abc.txt** for *read*, please.”
4. Response from `fs-cache`: “Sure, no problem.”

Sure enough, the caching filesystem handled the request for the read this time (in step 4)!

Now, we've left out a few details, but these aren't important to getting across the basic ideas. Obviously, the caching filesystem will need some way of sending the data across the network to the “real” storage medium. It should also have some way of verifying that no one else modified the file just before it returns the file contents to the client (so that the client doesn't get stale data). The caching filesystem *could* handle the first read request itself, by loading the data from the network filesystem on the first read into its cache. And so on.

Running more than one pass-through filesystem or resource manager on overlapping pathname spaces might cause deadlocks.

© 2020, QNX Software Systems Limited

UFS versus UMP

A slight terminology digression is in order. The primary difference between a *U*nioned *F*ile *S*ystem (UFS) and a *U*nioned *M*ount *P*oint (UMP) is that the UFS is based on a per-file organization, while the UMP is based on a per-mountpoint organization.

In the above cache filesystem example, we showed a UFS, because no matter how deep the file was in the tree structure, either resource manager was able to service it. In our example, consider another resource manager (let's call it "foobar") taking over "**/nfs/other**." In a UFS system, the `fs-cache` process would be able to cache files from that as well, just by attaching to "**/nfs**." In a UMP implementation, which is the default in QNX Neutrino since it does longest prefix match, only the `foobar` resource manager would get the open requests.

Client summary

We're done with the client side of things. The following are key points to remember:

- The client usually triggers communication with the resource manager via *open()* (or *fopen()*).
- Once the client's request has "resolved" to a particular resource manager, we never change resource managers.
- All further messages for the client's session are based on the file descriptor (or `FILE *` stream), (e.g., *read()*, *lseek()*, *fgets()*).
- The session is terminated (or "dissociated") when the client closes the file descriptor or stream (or terminates for any reason).
- All client file-descriptor-based function calls are translated into messages.

The resource manager's view

Let's look at things from the resource manager's perspective. Basically, the resource manager needs to tell the process manager that it'll be responsible for a certain part of the pathname space (it needs to *register* itself). Then, the resource manager needs to receive messages from clients and handle them. Obviously, things aren't quite *that* simple.

Let's take a quick overview look at the functions that the resource manager provides, and then we'll look at the details.

Registering a pathname

The resource manager needs to tell the process manager that one or more pathnames are now under its *domain of authority*—effectively, that this particular resource manager is prepared to handle client requests for those pathnames.

The serial port resource manager might handle (let's say) four serial ports. In this case, it would register four different pathnames with the process manager: **/dev/ser1**, **/dev/ser2**, **/dev/ser3**, and **/dev/ser4**. The impact of this is that there are now four distinct entries in the process manager's pathname tree, one for each of the serial ports. Four entries isn't too bad. But what if the serial port resource manager handled one of those fancy multiport cards, with 256 ports on it? Registering 256 *individual* pathnames (i.e., **/dev/ser1** through **/dev/ser256**) would result in 256 different entries in the process manager's pathname tree! The process manager isn't optimized for searching this tree; it assumes that there will be a few entries in the tree, not hundreds.

As a rule, you shouldn't *discretely* register more than a few dozen pathnames at each level—this is because a linear search is performed. The 256 port registration is certainly beyond that. In that case, what the multiport serial resource manager should do is register a *directory*-style pathname, for example **/dev/multiport**. This occupies only one entry in the process manager's pathname tree. When a client opens a serial port, let's say port 57:

```
fp = fopen ("/dev/multiport/57", "w");
```

The process manager resolves this to the ND/PID/CHID/handle for the multiport serial resource manager; it's up to that resource manager to decide if the rest of the pathname (in our case, the “57”) is valid. In this example, assuming that the variable *path* contains the rest of the pathname past the mountpoint, this means that the resource manager could do checking in a *very* simple manner:

```
devnum = atoi (path);
if ((devnum <= 0) || (devnum >= 256)) {
    // bad device number specified
} else {
    // good device number specified
}
```

This search would certainly be faster than anything the process manager could do, because the process manager must, by design, be much more general-purpose than our resource manager.

© 2020, QNX Software Systems Limited

Handling messages

Once we've registered one or more pathnames, we should then be prepared to receive messages from clients. This is done in the “usual” way, with the *MsgReceive()* function call. There are fewer than 30 well-defined message types that the resource manager handles. To simplify the discussion and implementation, however, they're broken into two groups:

Connect messages

Always contain a pathname; these are either one-shot messages or they establish a context for further I/O messages.

I/O messages

Always based on a connect message; these perform further work.

Connect messages

Connect messages always contain a pathname. The *open()* function that we've been using throughout our discussion is a perfect example of a function that generates a connect message. In this case, the handler for the connect message establishes a context for further I/O messages. (After all, we expect to be performing things like *read()* after we've done an *open()*).

An example of a “one-shot” connect message is the message generated as a result of the *rename()* function call. No further “context” is established—the handler in the resource manager is expected to change the name of the specified file to the new name, and that's it.

I/O messages

An I/O message is expected only after a connect message and refers to the context created by that connect message. As mentioned above in the connect message discussion, *open()* followed by *read()* is a perfect example of this.

Three groups, really

Apart from connect and I/O messages, there are also “other” messages that can be received (and handled) by a resource manager. Since they aren't “resource manager” messages proper, we'll defer discussion of them until later.

The resource manager library

Before we get too far into all the issues surrounding resource managers, we have to get acquainted with the QNX Neutrino resource manager library. Note that this "library" actually consists of several distinct pieces:

- thread pool functions (which we discussed in the *Processes and Threads chapter under "Pools of threads"*)
- dispatch interface
- resource manager functions
- POSIX library helper functions

While you certainly *could* write resource managers "from scratch" (as was done in the QNX 4 world), that's far more hassle than it's worth.

Just to show you the utility of the library approach, here's the source for a single-threaded version of "**/dev/null**":

```
/*
 *  resmgr1.c
 *
 *  /dev/null using the resource manager library
*/

#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int
main (int argc, char **argv)
{
    dispatch_t              *dpp;
    resmgr_attr_t           resmgr_attr;
    dispatch_context_t      *ctp;
    resmgr_connect_funcs_t  connect_func;
    resmgr_io_funcs_t       io_func;
    iofunc_attr_t           attr;

    // create the dispatch structure
    if ((dpp = dispatch_create ()) == NULL) {
        perror ("Unable to dispatch_create");
        exit (EXIT_FAILURE);
    }

    // initialize the various data structures
    memset (&resmgr_attr, 0, sizeof (resmgr_attr));
    resmgr_attr.nparts_max = 1;
    resmgr_attr.msg_max_size = 2048;

    // bind default functions into the outcall tables
```


© 2020, QNX Software Systems Limited

```
    iofunc_func_init (_RESMGR_CONNECT_NFUNCS, &connect_func,
                      _RESMGR_IO_NFUNCS, &io_func);
    iofunc_attr_init (&attr, S_IFNAM | 0666, 0, 0);

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

    ctp = dispatch_context_alloc (dpp);

    // wait here forever, handling messages
    while (1) {
        if ((ctp = dispatch_block (ctp)) == NULL) {
            perror ("Unable to dispatch_block");
            exit (EXIT_FAILURE);
        }
        dispatch_handler (ctp);
    }
}
```

There you have it! A complete **/dev/null** resource manager implemented in a few function calls!

If you were to write this from scratch, and have it support all the functionality that this one does (e.g., *stat()* works, *chown()* and *chmod()* work, and so on), you'd be looking at many hundreds if not thousands of lines of C code.

The library really does what we just talked about

By way of introduction to the library, let's see (briefly) what the calls do in the **/dev/null** resource manager.

dispatch_create()

Creates a dispatch structure; this will be used for blocking on the message reception.

iofunc_attr_init()

Initializes the attributes structure used by the device. We'll discuss attributes structures in more depth later, but for now, the short story is that there's one of these per device name, and they contain information about a particular device.

iofunc_func_init()

Initializes the two data structures *cfuncs* and *ifuncs*, which contain pointers to the connect and I/O functions, respectively. You might argue that this call has the most “magic” in it, as this is where the actual “worker” routines for handling all the messages got bound into a data structure. We didn't actually see any *code* to handle the connect message, or the I/O messages resulting from a client *read()* or *stat()* function etc. That's because the library is supplying default POSIX versions of those functions for us, and it's the *iofunc_func_init()* function that binds those same default handler functions into the two supplied tables.

resmgr_attach()

Creates the channel that the resource manager will use for receiving messages, and talks to the process manager to tell it that we're going to be responsible for "**/dev/null**." While there are a lot of parameters, we'll see them all in painful detail later. For now, it's important to note that this is where the dispatch handle (*dpp*), pathname (the string **/dev/null**), and the connect (*cfuncs*) and I/O (*ifuncs*) message handlers all get bound together.

dispatch_context_alloc()

Allocates a dispatch internal context block. It contains information relevant to the message being processed.

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

---

dispatch_block()

This is the dispatch layer's blocking call; it's where we wait for a message to arrive from a client.

dispatch_handler()

Once the message arrives from the client, this function is called to process it.

Behind the scenes at the library

You've seen that your code is responsible for providing the main message receiving loop:

```
while (1) {
    // wait here for a message
    if ((ctp = dispatch_block (ctp)) == NULL) {
        perror ("Unable to dispatch_block");
        exit (EXIT_FAILURE);
    }
    // handle the message
    dispatch_handler (ctp);
}
```

This is very convenient, for it lets you place breakpoints on the receiving function and to intercept messages (perhaps with a debugger) during operation.

The library implements the "magic" inside of the *dispatch_handler()* function, because that's where the message is analyzed and disposed of through the connect and I/O functions tables we mentioned earlier.

© 2020, QNX Software Systems Limited

In reality, the library consists of two cooperating layers: a *base layer* that provides "raw" resource manager functionality, and a *POSIX layer* that provides POSIX helper and default functions. We'll briefly define the two layers, and then in *"Resource manager structure,"* below, we'll pick up the details.

The base layer

The bottom-most layer consists of functions that begin with *resmgr_*()* in their names. This class of function is concerned with the mechanics of making a resource manager work.

I'll just briefly mention the functions that are available and where we'd use them. I'll then refer you to the QNX documentation for additional details on these functions.

The base layer functions consist of:

resmgr_msgreadv() **and** ***resmgr_msgread()***

Reads data from the client's address space using message passing.

resmgr_msgwritev() **and** ***resmgr_msgwrite()***

Writes data to the client's address space using message passing.

resmgr_open_bind()

Associates the context from a connect function, so that it can be used later by an I/O function.

resmgr_attach()

Creates a channel, associates a pathname, dispatch handle, connect functions, I/O functions, and other parameters together. Sends a message to the process manager to register the pathname.

resmgr_detach()

Opposite of *resmgr_attach()*; dissociates the binding of the pathname and the resource manager.

pulse_attach()

Associates a pulse code with a function. Since the library implements the message receive loop, this is a convenient way of "gaining control" for handling pulses.

pulse_detach()

Dissociates a pulse code from the function.

In addition to the functions listed above, there are also numerous functions dealing with the dispatch interface.

One function from the above list that deserves special mention is *resmgr_open_bind()*. It associates some form of context data when the connect message (typically as a result of the client calling *open()* or *fopen()*) arrives, so that this data block is around when the I/O messages are being handled. Why didn't we see this in the **/dev/null** handler? Because the POSIX-layer default functions call this function for us. If we're handling all the messages ourselves, we'd certainly call this function.

The *resmgr_open_bind()* function not only sets up the context block for further I/O messages, but *also* initializes other data structures used by the resource manager library itself.

The rest of the functions from the above list are somewhat intuitive—we'll defer their discussion until we use them.

The POSIX layer

The second layer provided by the resource manager library is the POSIX layer. As with the base layer, you *could* code a resource manager without using it, but it would be a *lot* of work! Before we can talk about the POSIX-layer functions in detail, we need to look at some of the base layer data structures, the messages that arrive from the clients, and the overall structure and responsibilities of a resource manager.

© 2020, QNX Software Systems Limited

Writing a resource manager

Now that we've introduced the basics—how the client looks at the world, how the resource manager looks at the world, and an overview of the two cooperating layers in the library, it's time to focus on the details.

In this section, we'll take a look at the following topics:

- *data structures*
- *resource manager structure*
- *POSIX-layer data structure*
- *handler routines*
- and of course, *lots of examples*

Keep in mind the following "big picture," which contains almost everything related to a resource manager:

Figure 52: Architecture of a resource manager: the big picture.

Data structures

The first thing we need to understand is the data structures used to control the operation of the library:

- `resmgr_attr_t` *control structure*
- `resmgr_connect_funcs_t` *connect table*
- `resmgr_io_funcs_t` *I/O table*

And one data structure that's used internally by the library:

- `resmgr_context_t` *internal context block*

Later, we'll see the *OCB*, *attributes structure*, and *mount structure* data types that are used with the POSIX-layer libraries.

`resmgr_attr_t` control structure

The control structure (type `resmgr_attr_t`) is passed to the *resmgr_attach()* function, which puts the resource manager's path into the general pathname space and binds requests on this path to a dispatch handle.

The control structure (from **<sys/dispatch.h>**) has the following contents:

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

The *other_func* message handler

In general, you should avoid using this member. This member, if non-NULL, represents a routine that will get called with the current message received by the resource manager library when the library doesn't recognize the message. While you *could* use this to implement "private" or "custom" messages, this practice is discouraged (use either the _IO_DEVCTL or _IO_MSG handlers, see below). If you wish to handle pulses that come in, I recommend that you use the *pulse_attach()* function instead.

You should leave this member with the value NULL.

The data structure sizing parameters

These two parameters are used to control various sizes of messaging areas.

The *nparts_max* parameter controls the size of the dynamically allocated *iov* member in the resource manager library context block (of type `resmgr_context_t`, see below). You'd typically adjust this member if you were returning more than a one-part IOV from some of your handling functions. Note that it has no effect on the *incoming* messages—this is only used on *outgoing* messages.

The *msg_max_size* parameter controls how much buffer space the resource manager library should set aside as a receive buffer for the message. The resource manager library will set this value to be at least as big as the header for the biggest message it will be receiving. This ensures that when your handler function gets called, it will be passed the entire header of the message. Note, however, that the data (if any) beyond the current header is *not* guaranteed to be present in the buffer, even if the *msg_max_size* parameter is "large enough." An example of this is when messages are transferred over a network using Qnet. (For more details about the buffer sizes, see "The `resmgr_context_t` internal context block," below.)

The *flags* parameter

This parameter gives additional information to the resource manager library.

For our purposes, we'll just pass a 0. You can read up about the other values in the QNX Neutrino *C Library Reference* under the *resmgr_attach()* function.

© 2020, QNX Software Systems Limited

`resmgr_connect_funcs_t` connect table

When the resource manager library receives a message, it looks at the type of message and sees if it can do anything with it. In the base layer, there are two tables that affect this behavior. The `resmgr_connect_funcs_t` table, which contains a list of connect message handlers, and the `resmgr_io_funcs_t` table, which contains a similar list of I/O message handlers. We'll see the I/O version below.

When it comes time to fill in the connect and I/O tables, we recommend that you use the *iofunc_func_init()* function to load up the tables with the POSIX-layer default handler routines. Then, if you need to override some of the functionality of particular message handlers, you'd simply assign your own handler function instead of the POSIX default routine. We'll see this in the section *"Putting in your own functions."* Right now, let's look at the connect functions table itself (this is from **<sys/resmgr.h>**):

```
typedef struct _resmgr_connect_funcs {
  unsigned nfuncs;

  int (*open)
      (ctp, io_open_t *msg, handle, void *extra);
  int (*unlink)
      (ctp, io_unlink_t *msg, handle, void *reserved);
  int (*rename)
      (ctp, io_rename_t *msg, handle, io_rename_extra_t *extra);
  int (*mknod)
      (ctp, io_mknod_t *msg, handle, void *reserved);
  int (*readlink)
      (ctp, io_readlink_t *msg, handle, void *reserved);
  int (*link)
      (ctp, io_link_t *msg, handle, io_link_extra_t *extra);
  int (*unblock)
      (ctp, io_pulse_t *msg, handle, void *reserved);
  int (*mount)
      (ctp, io_mount_t *msg, handle, io_mount_extra_t *extra);

} resmgr_connect_funcs_t;
```

Note that I've shortened the prototype down by omitting the `resmgr_context_t *` type for the first member (the *ctp*), and the `RESMGR_HANDLE_T *` type for the third member (the *handle*). For example, the full prototype for *open* is really:

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

The very first member of the structure (*nfuncs*) indicates how big the structure is (how many members it contains). In the above structure, it should contain the value "8," for there are 8 members (*open* through to *mount*). This member is mainly in place to allow QNX Software Systems to upgrade this library without any ill effects on your code. For example, suppose you had compiled in a value of 8, and then QNX Software Systems upgraded the library to have 9. Because the member only had a value of 8, the library could say to itself, "Aha! The user of this library was compiled when we had only 8 functions, and now we have 9. I'll provide a useful default for the ninth function." There's a manifest constant in **<sys/resmgr.h>** called _RESMGR_CONNECT_NFUNCS that has the current number. Use this constant if manually filling in the connect functions table (although it's *best* to use *iofunc_func_init()*).

Notice that the function prototypes all share a common format. The first parameter, *ctp*, is a pointer to a `resmgr_context_t` structure. This is an internal context block used by the resource manager library, and which you should treat as read-only (except for one field, which we'll come back to).

The second parameter is always a pointer to the message. Because the functions in the table are there to handle different types of messages, the prototypes match the kind of message that each function will handle.

The third parameter is a `RESMGR_HANDLE_T` structure called a *handle*—it's used to identify the device that this message was targeted at. We'll see this later as well, when we look at the attributes structure.

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

Finally, the last parameter is either "reserved" or an "extra" parameter for functions that need some extra data. We'll show the *extra* parameter as appropriate during our discussions of the handler functions.

`resmgr_io_funcs_t` I/O table

The I/O table is very similar in spirit to the connect functions table just shown above. Here it is, from **<sys/resmgr.h>**:

```
typedef struct _resmgr_io_funcs {
   unsigned   nfuncs;
   int (*read)       (ctp, io_read_t *msg, ocb);
   int (*write)      (ctp, io_write_t *msg, ocb);
   int (*close_ocb)  (ctp, void *reserved, ocb);
   int (*stat)       (ctp, io_stat_t *msg, ocb);
   int (*notify)     (ctp, io_notify_t *msg, ocb);
   int (*devctl)     (ctp, io_devctl_t *msg, ocb);
   int (*unblock)    (ctp, io_pulse_t *msg, ocb);
   int (*pathconf)   (ctp, io_pathconf_t *msg, ocb);
   int (*lseek)      (ctp, io_lseek_t *msg, ocb);
   int (*chmod)      (ctp, io_chmod_t *msg, ocb);
   int (*chown)      (ctp, io_chown_t *msg, ocb);
   int (*utime)      (ctp, io_utime_t *msg, ocb);
   int (*openfd)     (ctp, io_openfd_t *msg, ocb);
   int (*fdinfo)     (ctp, io_fdinfo_t *msg, ocb);
   int (*lock)       (ctp, io_lock_t *msg, ocb);
   int (*space)      (ctp, io_space_t *msg, ocb);
```


© 2020, QNX Software Systems Limited

```
    int (*shutdown)   (ctp, io_shutdown_t *msg, ocb);
    int (*mmap)       (ctp, io_mmap_t *msg, ocb);
    int (*msg)        (ctp, io_msg_t *msg, ocb);
    int (*reserved)   (ctp, void *msg, ocb);
    int (*dup)        (ctp, io_dup_t *msg, ocb);
    int (*close_dup)  (ctp, io_close_t *msg, ocb);
    int (*lock_ocb)   (ctp, void *reserved, ocb);
    int (*unlock_ocb) (ctp, void *reserved, ocb);
    int (*sync)       (ctp, io_sync_t *msg, ocb);
    int (*power)      (ctp, io_power_t *msg, ocb);
    int (*acl)        (ctp, io_acl_t *msg, ocb);
    int (*pause)      (ctp, void *reserved, ocb);
    int (*unpause)    (ctp, io_pulse_t *msg, ocb);
    int (*read64)     (ctp, io_read_t *msg, ocb);
    int (*write64)    (ctp, io_write_t *msg, ocb);
    int (*notify64)   (ctp, io_notify_t *msg, ocb);
    int (*utime64)    (ctp, io_utime_t *msg, ocb);
} resmgr_io_funcs_t;
```

For this structure as well, I've shortened the prototype by removing the type of the *ctp* member (`resmgr_context_t *`) and the last member (*ocb*, of type `RESMGR_OCB_T *`). For example, the full prototype for *read* is really:

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

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

The very first member of the structure (*nfuncs*) indicates how big the structure is (how many members it contains). The proper manifest constant for initialization is _RESMGR_IO_NFUNCS.

Note that the parameter list in the I/O table is also very regular. The first parameter is the *ctp*, and the second parameter is the *msg*, just as they were in the connect table handlers.

The third parameter is different, however. It's an *ocb*, which stands for “Open Context Block.” It holds the context that was bound by the connect message handler (e.g., as a result of the client's *open()* call), and is available to the I/O functions.

As discussed above, when it comes time to fill in the two tables, we recommend that you use the *iofunc_func_init()* function to load up the tables with the POSIX-layer default handler routines. Then, if you need to override some of the functionality of particular message handlers, you'd simply assign your own handler function instead of the default routine. We'll see this in the section *“Putting in your own functions.”*

The `resmgr_context_t` internal context block

Finally, one data structure is used by the lowest layer of the library to keep track of information that *it* needs to know about. You should view the contents of this data structure as "read-only," (except for the *iov* member).

Here's the data structure (from **<sys/resmgr.h>**):

```
typedef struct _resmgr_context {
  int                 rcvid;
  struct _msg_info    info;
  resmgr_iomsgs_t     *msg;
  dispatch_t          *dpp;
  int                 id;
  size_t              msg_max_size;
  long                status;
  size_t              offset;
  size_t              size;
  iov_t               iov [1];
} resmgr_context_t;
```

As with the other data structure examples, I've taken the liberty of deleting reserved fields.

Let's look at the contents:

rcvid

The receive ID from the resource manager library's *MsgReceivev()* function call. Indicates who you should reply to (if you're going to do the reply yourself).

info

Contains the information structure returned by *MsgReceivev()* in the resource manager library's receive loop. Useful for getting information about the client, including things like the node descriptor, process ID, thread ID, and so on. See the documentation for *MsgReceivev()* for more details.

msg

A pointer to a union of all possible message types. This isn't very useful to you, because each of your handler functions get passed the appropriate union member as their second parameter.

dpp

A pointer to the dispatch structure that you passed in to begin with. Again, not very useful to you, but obviously useful to the resource manager library.

id

The identifier for the mountpoint this message was meant for. When you did the *resmgr_attach()*, it returned a small integer ID. This ID is the value of the *id* member. Note

© 2020, QNX Software Systems Limited

that you'd most likely never use this parameter yourself, but would instead rely on the *attributes structure* passed to you in your open connect function handler.

msg_max_size

This contains the *msg_max_size* that was passed in as the *msg_max_size* member of `resmgr_attr_t` (given to the *resmgr_attach()* function) so that the *size*, *offset*, and *msg_max_size* are all contained in one handy structure/location.

status

This is where your handler function places the result of the operation. Note that you should always use the macro *_RESMGR_STATUS()* to write this field. For example, if you're handling the connect message from an *open()*, and you're a read-only resource manager but the client wanted to open you for write, you'd return an EROFS *errno* via (typically) `_RESMGR_STATUS (ctp, EROFS)`.

offset

The current number of bytes into the client's message buffer. Only relevant to the base layer library when used with *resmgr_msgreadv()* with *combine messages* (see below).

size

This tells you how many bytes are valid in the message area that gets passed to your handler function. This number is important because it indicates if more data needs to be read from the client (for example, if not all of the client's data was read by the resource manager base library), or if storage needs to be allocated for a reply to the client (for example, to reply to the client's *read()* request).

iov

The I/O Vector table where you can write your return values, if returning data. For example, when a client calls *read()* and your read-handling code is invoked, you may need to return data. This data can be set up in the *iov* array, and your read-handling code can then return something like `_RESMGR_NPARTS (2)` to indicate (in this example) that both `iov [0]` and `iov [1]` contain data to return to the client. Note that the *iov* member is defined as only having one element. However, you'll also notice that it's conveniently at the end of the structure. The actual number of elements in the *iov* array is defined by you when you set the *nparts_max* member of the *control structure* above (in the section "*`resmgr_attr_t` control structure,*" above).

Resource manager structure

Now that we've seen the data structures, we can discuss interactions between the parts that you'd supply to actually make your resource manager *do* something.

We'll look at:

- *The resmgr_attach() function and its parameters*
- *Putting in your own functions*
- *The general flow of a resource manager*

- *Messages that should be connect messages but aren't*
- *Combine messages*

The *resmgr_attach()* function and its parameters

As you saw in the **/dev/null** example above, the first thing you'll want to do is register your chosen "mountpoint" with the process manager. This is done via *resmgr_attach()*.

This function has the following prototype:

```
int
resmgr_attach (void *dpp,
               resmgr_attr_t *resmgr_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);
```

Let's examine these arguments, in order, and see what they're used for.

dpp

The dispatch handle. This lets the dispatch interface manage the message receive for your resource manager.

resmgr_attr

Controls the resource manager characteristics, as discussed above.

path

The mountpoint that you're registering. If you're registering a discrete mountpoint (such as would be the case, for example, with **/dev/null**, or **/dev/ser1**), then this mountpoint must be matched *exactly* by the client, with no further pathname components past the mountpoint. If you're registering a directory mountpoint (such as would be the case, for example, with a network filesystem mounted as **/nfs**), then the match must be exact as well, with the added feature that pathnames past the mountpoint are allowed; they get passed to the connect functions stripped of the mountpoint (for example, the pathname **/nfs/etc/passwd** would match the network filesystem resource manager, and it would get **etc/passwd** as the rest of the pathname).

file_type

The class of resource manager. See below.

flags

Additional flags to control the behavior of your resource manager. These flags are defined below.

© 2020, QNX Software Systems Limited

connect_funcs **and** ***io_funcs***

These are simply the list of connect functions and I/O functions that you wish to bind to the mountpoint.

handle

This is an "extendable" data structure (*aka* "attributes structure") that identifies the resource being mounted. For example, for a serial port, you'd extend the standard POSIX-layer attributes structure by adding information about the base address of the serial port, the baud rate, etc. Note that it *does not* have to be an attributes structure—if you're providing your own "open" handler, then you can choose to interpret this field any way you wish. It's only if you're using the default *iofunc_open_default()* handler as your "open" handler that this field *must* be an attributes structure.

The *flags* member can contain any of the following flags (or the constant 0 if none are specified):

_RESMGR_FLAG_BEFORE or _RESMGR_FLAG_AFTER

These flags indicate that your resource manager wishes to be placed before or after (respectively) other resource managers with the same mountpoint. These two flags would be useful with unioned (overlaid) filesystems. We'll discuss the interactions of these flags shortly.

_RESMGR_FLAG_DIR

This flag indicates that your resource manager is taking over the specified mountpoint *and below*—it's effectively a filesystem style of resource manager, as opposed to a discretely-manifested resource manager.

_RESMGR_FLAG_OPAQUE

If set, prevents resolving to any other manager below your mount point *except* for the path manager. This effectively eliminates unioning on a path.

_RESMGR_FLAG_FTYPEONLY

This ensures that only requests that have the same _FTYPE_* as the *file_type* passed to *resmgr_attach()* are matched.

_RESMGR_FLAG_FTYPEALL

This flag is used when a resource manager wants to catch *all* client requests, even those with a different _FTYPE_* specification than the one passed to *resmgr_attach()* in the *file_type* argument. This can only be used in conjunction with a registration file type of _FTYPE_ALL.

_RESMGR_FLAG_SELF

Allow this resource manager to talk to itself. This really is a "Don't try this at home, kids" kind of flag, because allowing a resource manager to talk to itself can break the send-hierarchy and lead to deadlock (as was discussed in the Message Passing chapter).

You can call *resmgr_attach()* as many times as you wish to mount different mountpoints. You can also call *resmgr_attach() from within the connect or I/O functions*—this is kind of a neat feature that allows you to "create" devices on the fly.

Your resource manager needs certain abilities enabled when it calls *resmgr_attach()*:

- PROCMGR_AID_PATHSPACE, to add a name to the pathname space
- PROCMGR_AID_PUBLIC_CHANNEL, to create a public channel (i.e., *without* _NTO_CHF_PRIVATE set)

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

When you've decided on the mountpoint, and want to create it, you'll need to tell the process manager if this resource manager can handle requests from just anyone, or if it's limited to handling requests only from clients who identify their connect messages with special tags. For example, consider the POSIX message queue (`mqueue`) driver. It's not going to allow (and certainly wouldn't know what to do with) "regular" *open()* messages from any old client. It will allow messages only from clients that use the POSIX *mq_open()*, *mq_receive()*, and so on, function calls. To prevent the process manager from even allowing regular requests to arrive at the `mqueue` resource manager, `mqueue` specified _FTYPE_MQUEUE as the *file_type* parameter. This means that when a client requests a name resolution from the process manager, the process manager won't even bother considering the resource manager during the search unless the client has specified that it wants to talk to a resource manager that has identified itself as _FTYPE_MQUEUE.

Unless you're doing something very special, you'll use a *file_type* of _FTYPE_ANY, which means that your resource manager is prepared to handle requests from anyone. For the full list of _FTYPE_* manifest constants, take a look in **<sys/ftype.h>**.

With respect to the "before" and "after" flags, things get a little bit more interesting. You can specify only one of these flags or the constant 0.

Let's see how this works. A number of resource managers have started, in the order given in the table. We also see the flags they passed for the *flags* member. Observe the positions they're given:

Resmgr	Flag	Order
1	_RESMGR_FLAG_BEFORE	1
2	_RESMGR_FLAG_AFTER	1, 2
3	0	1, 3, 2
4	_RESMGR_FLAG_BEFORE	1, 4, 3, 2
5	_RESMGR_FLAG_AFTER	1, 4, 3, 5, 2
6	0	1, 4, 6, 3, 5, 2

As you can see, the first resource manager to actually specify a flag always ends up in that position. (From the table, resource manager number 1 was the first to specify the "before" flag; no matter who registers, resource manager 1 is always first in the list. Likewise, resource manager 2 was the first to specify the "after" flag; again, no matter who else registers, it's always last.) If no flag is specified, it effectively acts as a "middle" flag. When resource manager 3 started with a flag of zero, it got put into

© 2020, QNX Software Systems Limited

the middle. As with the “before” and “after” flags, there's a preferential ordering given to all the “middle” resource managers, whereby newer ones are placed in front of other, existing “middle” ones.

However, in reality, there are very few cases where you'd actually mount more than one, and even fewer cases where you'd mount more than two resource managers at the same mountpoint. Here's a design tip: expose the ability to set the flags at the command line of the resource manager so that the end-user of your resource manager is able to specify, for example, `-b` to use the “before” flag, and `-a` to use the “after” flag, with no command-line option specified to indicate that a zero should be passed as the flag.

Keep in mind that this discussion applies *only* to resource managers mounted with the same mountpoint. Mounting “**/nfs**” with a “before” flag and “**/disk2**” with an “after” flag will have no effect on each other; only if you were to *then* mount *another* “**/nfs**” or “**/disk2**” would these flags (and rules) come into play.

Finally, the *resmgr_attach()* function returns a small integer handle on success (or -1 for failure). This handle can then be used subsequently to detach the pathname from the process manager's internal pathname tables.

Putting in your own functions

When designing your very first resource manager, you'll most likely want to take an incremental design approach. It can be very frustrating to write thousands of lines of code only to run into a fundamental misunderstanding and then having to make the ugly decision of whether to try to kludge (er, I mean “fix”) all that code, or scrap it and start from scratch.

The recommended approach for getting things running is to use the *iofunc_func_init()* POSIX-layer default initializer function to fill the connect and I/O tables with the POSIX-layer default functions. This means that you can literally write your initial cut of your resource manager as we did above, in a few function calls.

Which function you'll want to implement first really depends on what kind of resource manager you're writing. If it's a filesystem type of resource manager where you're taking over a mountpoint and everything below it, you'll most likely be best off starting with the *open connect function handler*. (The _RESMGR_FLAG_DIR flag indicates that the pathname should be treated as a directory.) On the other hand, if it's a discretely manifested resource manager that does “traditional” I/O operations (i.e., you primarily access it with client calls like *read()* and *write()*), then the best place to start would be the *read I/O function handler* and/or *write I/O function handler*. The third possibility is that it's a discretely manifested resource manager that doesn't do traditional I/O operations, but instead relies on *devctl()* or *ioctl()* client calls to perform the majority of its functionality. In that case, you'd start at the *device control I/O function handler*.

Regardless of where you start, you'll want to make sure that your functions are getting called in the expected manner. The really cool thing about the POSIX-layer default functions is that they can be placed directly into the connect or I/O functions table. This means that if you simply want to gain control, perform a *printf()* to say “I'm here in the io_open!”, and then “do whatever should be done,” you're going to have an easy time of it. Here's a portion of a resource manager that takes over the *open connect function handler*:

```
// forward reference
int io_open (resmgr_context_t *, io_open_t *,
```

```
                RESMGR_HANDLE_T *, void *);

int
main ()
{
    // everything as before, in the /dev/null example
    // except after this line:
    iofunc_func_init (_RESMGR_CONNECT_NFUNCS, &cfuncs,
                      _RESMGR_IO_NFUNCS, &ifuncs);

    // add the following to gain control:
    cfuncs.open = io_open;
```

Assuming that you've prototyped the *open connect function handler* correctly, as in the code example, you can just use the default one from within your own!

```
int
io_open (resmgr_context_t *ctp, io_open_t *msg,
         RESMGR_HANDLE_T *handle, void *extra)
{
    printf ("I'm here in the io_open!\n");
    return (iofunc_open_default (ctp, msg, handle, extra));
}
```

In this manner, you're still using the default POSIX-layer *iofunc_open_default()* handler, but you've also gained control to do a *printf()*.

Obviously, you could do this for the *read I/O function handler*, *write I/O function handler*, and *device control I/O function handler* as well as any others that have POSIX-layer default functions. In fact, this is a really good idea, because it shows you that the client *really is* calling your resource manager as expected.

The general flow of a resource manager

As we alluded to in the client and resource manager overview sections above, the general flow of a resource manager begins on the client side with the *open()*. This gets translated into a connect message and ends up being received by the resource manager's open connect function handler.

This is really key, because the *open connect function handler* is the “gate keeper” for your resource manager. If the message causes the gate keeper to fail the request, you *will not* get any I/O requests, because the client never got a valid file descriptor. Conversely, if the message is accepted by the gate keeper, the client now has a valid file descriptor and you should expect to get I/O messages.

But the open connect function handler plays a greater role. Not only is it responsible for verifying whether the client can or can't open the particular resource, it's also responsible for:

- initializing internal library parameters
- binding a context block to this request
- binding an attribute structure to the context block.

© 2020, QNX Software Systems Limited

The first two operations are performed via the base layer function *resmgr_open_bind()*; the binding of the attribute structure is done via a simple assignment.

Once the open connect function handler has been called, it's out of the picture. The client may or may not send I/O messages, but in any case will eventually terminating the "session" with a message corresponding to the *close()* function. Note that if the client suffers an unexpected death (e.g., gets hit with SIGSEGV, or the node that it's running on crashes), the operating system will synthesize a *close()* message so that the resource manager can clean up. Therefore, you are *guaranteed* to get a *close()* message!

Messages that *should* be connect messages but aren't

Here's an interesting point you may have noticed. The client's prototype for *chown()* is:

```
int
chown (const char *path,
       uid_t owner,
       gid_t group);
```

Remember, a connect message always contains a pathname and is either a one-shot message or establishes a context for further I/O messages.

So, why isn't there a *connect* message for the client's *chown()* function? In fact, why *is* there an *I/O* message?!? There's certainly no file descriptor implied in the client's prototype!

The answer is, "to make your life simpler!"

Imagine if functions like *chown()*, *chmod()*, *stat()*, and others required the resource manager to look up the pathname and then perform some kind of work. (This is, by the way, the way it was implemented in QNX 4.) The usual problems with this are:

- Each function has to call the lookup routine.
- Where file descriptor versions of these functions exist, the driver has to provide two separate entry points; one for the pathname version, and one for the file descriptor version.

In any event, what happens under QNX Neutrino is that the client constructs a *combine message*—really just a single message that comprises multiple resource manager messages. Without combine messages, we could simulate *chown()* with something like this:

```
int
chown (const char *path, uid_t owner, gid_t group)
{
    int fd, sts;

    if ((fd = open (path, O_RDWR)) == -1) {
        return (-1);
    }
    sts = fchown (fd, owner, group);
    close (fd);
    return (sts);
}
```

where *fchown()* is the file-descriptor-based version of *chown()*. The problem here is that we are now issuing three function calls (and three separate message passing transactions), and incurring the overhead of *open()* and *close()* on the client side.

With combine messages, under QNX Neutrino a single message that looks like this is constructed directly by the client's *chown()* library call:

_IO_CONNECT_COMBINE_CLOSE	_IO_CHOWN

Figure 53: A combine message.

The message has two parts, a connect part (similar to what the client's *open()* would have generated) and an I/O part (the equivalent of the message generated by the *fchown()*). There is no equivalent of the *close()* because we implied that in our particular choice of connect messages. We used the _IO_CONNECT_COMBINE_CLOSE message, which effectively states "Open this pathname, use the file descriptor you got for handling the rest of the message, and when you run off the end or encounter an error, close the file descriptor."

The resource manager that you write doesn't have a clue that the client called *chown()* or that the client did a distinct *open()*, followed by an *fchown()*, followed by a *close()*. It's all hidden by the base-layer library.

Combine messages

As it turns out, this concept of combine messages isn't useful just for saving bandwidth (as in the *chown()* case, above). It's also critical for ensuring atomic completion of operations.

Suppose the client process has two or more threads and one file descriptor. One of the threads in the client does an *lseek()* followed by a *read()*. Everything is as we expect it. If another thread in the client does the same set of operations, *on the same file descriptor*, we'd run into problems. Since the *lseek()* and *read()* functions don't know about each other, it's possible that the first thread would do the *lseek()*, and then get preempted by the second thread. The second thread gets to do its *lseek()*, and then its *read()*, before giving up CPU. The problem is that since the two threads are sharing the *same* file descriptor, the first thread's *lseek()* offset is now at the wrong place—it's at the position given by the second thread's *read()* function! This is also a problem with file descriptors that are *dup()*'d across processes, let alone the network.

An obvious solution to this is to put the *lseek()* and *read()* functions within a mutex—when the first thread obtains the mutex, we now know that it has exclusive access to the file descriptor. The second thread has to wait until it can acquire the mutex before it can go and mess around with the position of the file descriptor.

Unfortunately, if someone forgot to obtain a mutex *for each and every file descriptor operation*, there'd be a possibility that such an "unprotected" access would cause a thread to read or write data to the wrong location.

Let's look at the C library call *readblock()* (from **<unistd.h>**):

```
int
readblock (int fd,
           size_t blksize,
           unsigned block,
```


© 2020, QNX Software Systems Limited

```
            int numblks,
            void *buff);
```

(The *writeblock()* function is similar.)

You can imagine a fairly "simplistic" implementation for *readblock()*:

```
int
readblock (int fd, size_t blksize, unsigned block,
           int numblks, void *buff)
{
    lseek (fd, blksize * block, SEEK_SET); // get to the block
    read (fd, buff, blksize * numblks);
}
```

Obviously, this implementation isn't useful in a multithreaded environment. We'd have to at least put a mutex around the calls:

```
int
readblock (int fd, size_t blksize, unsigned block,
           int numblks, void *buff)
{
    pthread_mutex_lock (&block_mutex);
    lseek (fd, blksize * block, SEEK_SET); // get to the block
    read (fd, buff, blksize * numblks);
    pthread_mutex_unlock (&block_mutex);
}
```

(We're assuming the mutex is already initialized.)

This code is still vulnerable to "unprotected" access; if some other thread in the process does a simple non-mutexed *lseek()* on the file descriptor, we've got a bug.

The solution to this is to use a combine message, as we discussed above for the *chown()* function. In this case, the C library implementation of *readblock()* puts *both* the *lseek() and* the *read()* operations into a single message and sends that off to the resource manager:

_IO_LSEEK	_IO_READ

Figure 54: The *readblock()* function's combine message.

The reason that this works is because message passing is atomic. From the client's point of view, either the entire message has gone to the resource manager, or none of it has. Therefore, an intervening "unprotected" *lseek()* is irrelevant—when the *readblock()* operation is received by the resource manager, it's done in one shot. (Obviously, the damage will be to the unprotected *lseek()*, because after the *readblock()* the file descriptor's offset is at a different place than where the original *lseek()* put it.)

But what about the resource manager? How does it ensure that it *processes* the entire *readblock()* operation in one shot? We'll see this shortly, when we discuss the operations performed for each message component.

POSIX-layer data structures

There are three data structures that relate to the POSIX-layer support routines. Note that as far as the *base layer* is concerned, you can use any data structures you want; it's the POSIX layer that requires you to conform to a certain content and layout. The benefits delivered by the POSIX layer are well worth this tiny constraint. As we'll see later, you can add your own content to the structures as well.

The three data structures are illustrated in the following diagram, showing some clients using a resource manager that happens to manifest two devices:

Figure 55: Data structures: the big picture.

The data structures (defined in **<sys/iofunc.h>**) are:

***iofunc_ocb_t*—OCB structure**

Contains information on a per-file-descriptor basis

***iofunc_attr_t*—attributes structure**

Contains information on a per-device basis

***iofunc_mount_t*—mount structure**

Contains information on a per-mountpoint basis

When we talked about the I/O and connect tables, you saw the OCB and attributes structures—in the I/O tables, the OCB structure was the last parameter passed. The attributes structure was passed as the *handle* in the connect table functions (third argument). The mount structure is usually a global structure and is bound to the attributes structure "by hand" (in the initialization code that you supply for your resource manager).

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

© 2020, QNX Software Systems Limited

The `iofunc_ocb_t` OCB structure

The OCB structure contains information on a per-file-descriptor basis. What this means is that when a client performs an *open()* call and gets back a file descriptor (as opposed to an error indication), the resource manager will have created an OCB and associated it with the client. This OCB will be around for as long as the client has the file descriptor open.

Effectively, the OCB and the file descriptor are a matched pair. Whenever the client calls an I/O function, the resource manager library will automatically associate the OCB, and pass it along with the message to the I/O function specified by the I/O function table entry. This is why the I/O functions all had the *ocb* parameter passed to them. Finally, the client will close the file descriptor (via *close()*), which will cause the resource manager to dissociate the OCB from the file descriptor and client. Note that the client's *dup()* function simply increments a reference count. In this case, the OCB gets dissociated from the file descriptor and client only when the reference count reaches zero (i.e., when the same number of *close()*s have been called as *open()* and *dup()*s.)

As you might suspect, the OCB contains things that are important on a per-open or per-file-descriptor basis. Here are the contents (from **<sys/iofunc.h>**):

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

Ignore the comment about the *offset* field for now; we'll come back to it immediately after this discussion.

The `iofunc_ocb_t` members are:

attr

A pointer to the attributes structure related to this OCB. A common coding idiom you'll see in the I/O functions is "`ocb->attr`," used to access a member of the attributes structure.

ioflag

The open mode; how this resource was opened (e.g., read only). The open modes (as passed to *open()* on the client side) correspond to the *ioflag* values as follows:

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

offset

The current *lseek()* offset into this resource.

sflag

The sharing flag (see **<share.h>**) used with the client's *sopen()* function call. These are the flags SH_COMPAT, SH_DENYRW, SH_DENYWR, SH_DENYRD, and SH_DENYNO.

flags

System flags. The flags currently supported are IOFUNC_OCB_PRIVILEGED, which indicates whether a privileged process issued the connect message that resulted in this OCB, IOFUNC_OCB_MMAP, which indicates whether this OCB is in use by a *mmap()* call on the client side, and IOFUNC_OCB_MMAP_UNIQUE, which is a hint for the *mmap()* handler to provide a unique mapping. No other flags are defined at this time. You can use the bits defined by IOFUNC_OCB_FLAGS_PRIVATE for your own private flags.

If you wish to store additional data along with the "normal" OCB, rest assured that you can "extend" the OCB. We'll discuss this in the *"Advanced topics" section.*

The strange case of the *offset* member

The *offset* field is, to say the least, interesting. Have a look at **<sys/iofunc.h>** to see how it's implemented. Depending on what preprocessor flags you've set, you may get one of six(!) possible layouts for the *offset* area. But don't worry too much about the implementation—there are really only two cases to consider, depending on whether you want to support 64-bit offsets:

- yes—the *offset* member is 64 bits
- no (32-bit integers)—the *offset* member is the lower 32 bits; another member, *offset_hi* contains the upper 32 bits.

For our purposes here, unless we're specifically going to talk about 32 versus 64 bits, we'll just assume that all offsets are 64 bits, of type `off_t`, and that the platform knows how to deal with 64-bit quantities.

The `iofunc_attr_t` attributes structure

Whereas the OCB was a per-open or per-file-descriptor structure, the attributes structure is a per-device data structure. You saw that the standard `iofunc_ocb_t` OCB had a member called *attr* that's a pointer to the attribute structure. This was done so the OCB has access to information about the device. Let's take a look at the attributes structure (from **<sys/iofunc.h>**):

```
typedef struct _iofunc_attr {
  IOFUNC_MOUNT_T          *mount;
  uint32_t                flags;
  int32_t                 lock_tid;
  uint16_t                lock_count;
  uint16_t                count;
  uint16_t                rcount;
  uint16_t                wcount;
```


© 2020, QNX Software Systems Limited

```
    uint16_t                    rlocks;
    uint16_t                    wlocks;
    struct _iofunc_mmap_list    *mmap_list;
    struct _iofunc_lock_list    *lock_list;
    void                        *list;
    uint32_t                    list_size;
    SEE_BELOW!!!                nbytes;
    SEE_BELOW!!!                inode;
    uid_t                       uid;
    gid_t                       gid;
    time_t                      mtime;
    time_t                      atime;
    time_t                      ctime;
    mode_t                      mode;
    nlink_t                     nlink;
    dev_t                       rdev;
    unsigned                    mtime_ns;
    unsigned                    atime_ns;
    unsigned                    ctime_ns;
} iofunc_attr_t;
```

The *nbytes* and *inode* members have the same set of `#ifdef` conditionals as the *offset* member of the OCB (see *"The strange case of the offset member"* above).

Note that some of the fields of the attributes structure are useful only to the POSIX helper routines.

Let's look at the fields individually:

mount

A pointer to the optional `iofunc_mount_t` mount structure. This is used in the same way that the pointer from the OCB to the attribute structure was used, except that this value can be NULL in which case the mount structure defaults are used (see *"The `iofunc_mount_t` mount structure"* below). As mentioned, the mount structure is generally bound "by hand" into the attributes structure in code that you supply for your resource manager initialization.

flags

Contains flags that describe the state of other attributes structure fields. We'll discuss these shortly.

lock_tid

In order to prevent synchronization problems, multiple threads using the same attributes structure will be mutually exclusive. The *lock_tid* contains the thread ID of the thread that currently has the attributes structure locked.

lock_count

Indicates how many threads are trying to use this attributes structure. A value of zero indicates that the structure is unlocked. A value of one or more indicates that one or more threads are using the structure.

count

Indicates the number of OCBs that have this attributes structure open for any reason. For example, if one client has an OCB open for read, another client has another OCB open for read/write, and both OCBs point to this attribute structure, then the value of *count* would be 2, to indicate that two clients have this resource open.

rcount

Count readers. In the example given for *count*, *rcount* would also have the value 2, because two clients have the resource open for reading.

wcount

Count writers. In the example given for *count*, *wcount* would have the value 1, because only one of the clients has this resource open for writing.

rlocks

Indicates the number of OCBs that have read locks on the particular resource. If zero, means there are no read locks, but there may be write locks.

wlocks

Same as *rlocks* but for write locks.

mmap_list

Used internally by POSIX *iofunc_mmap_default()*.

lock_list

Used internally by POSIX *iofunc_lock_default()*.

list

Reserved for future use.

list_size

Size of area reserved by *list*.

nbytes

Size of the resource, in bytes. For example, if this resource described a particular file, and that file was 7756 bytes in size, then the *nbytes* member would contain the number 7756.

inode

Contains a file or resource serial number, that must be unique per mountpoint. The *inode* should never be zero, because zero traditionally indicates a file that's not in use.

© 2020, QNX Software Systems Limited

uid

User ID of the owner of this resource.

gid

Group ID of the owner of this resource.

mtime

File modification time, updated or at least invalidated whenever a client *write()* is processed.

atime

File access time, updated or at least invalidated whenever a client *read()* that returns more than zero bytes is processed.

ctime

File change time, updated or at least invalidated whenever a client *write()*, *chown()*, or *chmod()* is processed.

mode

File's mode. These are the standard S_* values from **<sys/stat.h>**, such as S_IFCHR, or in octal representation, such as 0664 to indicate read/write permission for owner and group, and read-only permission for other.

nlink

Number of links to the file, returned by the client's *stat()* function call.

rdev

For a character special device, this field consists of a major and minor device code (10 bits minor in the least-significant positions; next 6 bits are the major device number). For other types of devices, contains the device number. (See below in *"Of device numbers, inodes, and our friend rdev,"* for more discussion.)

mtime_ns*, *atime_ns*, and *ctime_ns

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

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

As with the OCB, you can extend the "normal" attributes structure with your own data. See the *"Advanced topics" section.*

The `iofunc_mount_t` mount structure

The mount structure contains information that's common across multiple attributes structures.

Here are the contents of the mount structure (from **<sys/iofunc.h>**):

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

The *flags* member contains just one flag, IOFUNC_MOUNT_32BIT. This flag indicates that *offset* in the OCB, and *nbytes* and *inode* in the attributes structure, are 32-bit. Note that you can define your own flags in *flags*, using any of the bits from the constant IOFUNC_MOUNT_FLAGS_PRIVATE.

The *conf* member contains the following flags:

IOFUNC_PC_CHOWN_RESTRICTED

Indicates if the filesystem is operating in a "`chown`-restricted" manner, meaning if only **root** is allowed to `chown` a file.

IOFUNC_PC_NO_TRUNC

Indicates that the filesystem doesn't truncate the name.

IOFUNC_PC_SYNC_IO

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

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

IOFUNC_PC_LINK_DIR

Indicates that linking/unlinking of directories is allowed.

IOFUNC_PC_ACL

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

The *dev* member contains the device number and is described below in *"Of device numbers, inodes, and our friend rdev."*

The *blocksize* describes the native blocksize of the device in bytes. For example, on a typical rotating-medium storage system, this would be the value 512.

© 2020, QNX Software Systems Limited

Finally, the *funcs* pointer points to a structure (from **<sys/iofunc.h>**):

```
typedef struct _iofunc_funcs {
  unsigned      nfuncs;

  IOFUNC_OCB_T *(*ocb_calloc)
                    (resmgr_context_t *ctp,
                     IOFUNC_ATTR_T *attr);

  void          (*ocb_free)
                    (IOFUNC_OCB_T *ocb);
} iofunc_funcs_t;
```

As with the connect and I/O functions tables, the *nfuncs* member should be stuffed with the current size of the table. Use the constant _IOFUNC_NFUNCS for this.

The *ocb_calloc* and *ocb_free* function pointers can be filled with addresses of functions to call whenever an OCB is to be allocated or deallocated. We'll discuss why you'd want to use these functions later when we talk about extending OCBs.

Of device numbers, inodes, and our friend *rdev*

The mount structure contains a member called *dev*. The attributes structure contains two members: *inode* and *rdev*. Let's look at their relationships by examining a traditional disk-based filesystem. The filesystem is mounted on a block device (which is the entire disk). This block device might be known as **/dev/hd0** (the first hard disk in the system). On this disk, there might be a number of partitions, such as **/dev/hd0t177** (the first QNX filesystem partition on that particular device). Finally, within that partition, there might be an arbitrary number of files, one of which might be **/hd/spud.txt**.

The *dev* (or "device number") member, contains a number that's unique to the node that this resource manager is registered with. The *rdev* member is the *dev* number of the root device. Finally, the *inode* is the file serial number. (Note that you can obtain major and minor device numbers by calling *rsrcdbmgr_devno_attach()*; see the QNX Neutrino *C Library Reference* for more details. You are limited to 64 major devices and 1024 minor devices per major device.)

Let's relate that to our disk example. The following table shows some example numbers; after the table we'll take a look at where these numbers came from and how they're related.

Device	*dev*	*inode*	*rdev*
/dev/hd0	6	2	1
/dev/hd0t177	1	12	77
/hd/spud.txt	77	47343	N/A

For the raw block device, **/dev/hd0**, the process manager assigned both the *dev* and *inode* values (the 6 and the 2 in the table above). The resource manager picked a unique *rdev* value (of 1) for the device when it started.

For the partition, **/dev/hd0t177**, the *dev* value came from the raw block device's *rdev* number (the 1). The inode was selected by the resource manager as a unique number (within the *rdev*). This is where the 12 came from. Finally, the *rdev* number was selected by the resource manager as well—in this case, the writer of the resource manager selected 77 because it corresponded to the partition type.

Finally, for the file, **/hd/spud.txt**, the *dev* value (77) came from the partition's *rdev* value. The *inode* was selected by the resource manager (in the case of a file, the number is selected to correspond to some internal representation of the file—it doesn't matter what it is so long as it's not zero, and it's unique within the *rdev*). This is where the 47343 came from. For a file, the *rdev* field is not meaningful.

© 2020, QNX Software Systems Limited

Handler routines

Not all outcalls correspond to client messages—some are synthesized by the kernel, and some by the library.

I've organized this section into the following:

- *general notes*
- *connect functions notes*

followed by an *alphabetical listing of connect and I/O messages*.

General notes

Each handler function gets passed an internal context block (the *ctp* argument) that should be treated as "read-only," except for the *iov* member. This context block contains a few items of interest, as described above in "`resmgr_context_t` internal context block." Also, each function gets passed a pointer to the message (in the *msg* argument). You'll be using this message pointer extensively, as that contains the parameters that the client's C library call has placed there for your use.

The function that you supply must return a value (all functions are prototyped as returning in `int`). The values are selected from the following list:

_RESMGR_NOREPLY

Indicates to the resource manager library that it should *not* perform the *MsgReplyv()*—the assumption is that you've either performed it yourself in your handler function, or that you're going to do it some time later.

_RESMGR_NPARTS (n)

The resource manager library should return an *n*-part IOV when it does the *MsgReplyv()* (the IOV is located in `ctp -> iov`). Your function is responsible for filling in the *iov* member of the *ctp* structure, and then returning *_RESMGR_NPARTS()* with the correct number of parts.

The *iov* member of *ctp* is allocated dynamically, so it must be *big enough* to hold the number of array elements that you're writing into the *iov* member! See the section "`resmgr_attr_t` control structure" above, for information on setting the *nparts_max* member.

_RESMGR_DEFAULT

This instructs the resource manager library to perform the *low-level default* function (This is *not* the same as the *iofunc_*_default()* functions!) You'd rarely ever use this return value. In general, it causes the resource manager library to return an errno of ENOSYS to the client, which indicates that the function is not supported.

An *errno* value

Indicates to the resource manager library that it should call *MsgError()* with this value as the *error* parameter. This generally causes the client function (e.g., *open()*) to return -1 and set *errno* on the client side to the returned value.

_RESMGR_ERRNO (errno)

(Deprecated) This return value had been used to "wrap" an errno number as the return value of the message. For example, if a client issued an *open()* request for a read-only device, it would be appropriate to return the error value EROFS. Since this function is deprecated, you can return the error number directly instead of wrapping it with the *_RESMGR_ERRNO()* macro (e.g., `return (EROFS);` instead of the more cumbersome `return (_RESMGR_ERRNO (EROFS));`.)

_RESMGR_PTR (ctp, addr, len)

This is a convenience macro that accepts the context pointer *ctp*, and fills its first IOV element to point to the address specified by *addr* for the length specified by *len*, and then returns the equivalent of `_RESMGR_NPARTS (1)` to the library. You'd generally use this if you return single-part IOVs from your function.

Locking, unlocking, and combine message handling

We saw the client side of a combine message when we looked at *readblock()* (in "Combine messages"). The client was able to atomically construct a message that contained multiple resource manager "submessages"—in the example, these were messages corresponding to the individual functions *lseek()* and *read()*. From the client's perspective, the two (or more) functions were at least *sent* atomically (and, due to the nature of message passing, will be *received* atomically by the resource manager). What we haven't yet talked about is how we ensure that the messages are *processed* atomically.

This discussion applies not only to combine messages, but to *all* I/O messages received by the resource manager library (except the close message, which we'll come back to shortly).

The very first thing that the resource manager library does is to lock the attribute structure corresponding to the resource being used by the received message. Then, it processes one or more submessages from the incoming message. Finally, it unlocks the attribute structure.

This ensures that the incoming messages are handled atomically, for no other thread in the resource manager (in the case of a multithreaded resource manager, of course) can "jump in" and modify the resource while a thread is busy using it. Without the locking in place, two client threads could both issue what they believe to be an atomic combine message (say *lseek()* and *read()*). Since the resource manager might have two different threads running in it and processing messages, the two resource manager threads could possibly preempt each other, and the *lseek()* components could interfere with each other. With locking and unlocking, this is prevented, because each message that accesses a resource will be completed in its entirety atomically.

Locking and unlocking the resource is handled by default helper functions (*iofunc_lock_ocb_default()* and *iofunc_unlock_ocb_default()*), which are placed in the I/O table at the *lock_ocb* and *unlock_ocb*

© 2020, QNX Software Systems Limited

positions. You can, of course, override these functions if you want to perform further actions during this locking and unlocking phase.

Note that the resource *is unlocked before* the close OCB I/O function handler is called. This is necessary because the close OCB I/O function handler will free the OCB, which would effectively invalidate the pointer used to access the attributes structure, which is where the lock is stored! Also note that none of the connect functions do this locking, because the handle that's passed to them does not *have to be* an attribute structure (and the locks are stored in the attribute structure).

Connect functions notes

Before we dive into the individual messages, however, it's worth pointing out that the connect functions all have an identical message structure (rearranged slightly, see **<sys/iomsg.h>** for the original):

```
struct _io_connect {
    // Internal use
    uint16_t type;
    uint16_t subtype;
    uint32_t file_type;
    uint16_t reply_max;
    uint16_t entry_max;
    uint32_t key;
    uint32_t handle;
    uint32_t ioflag;
    uint32_t mode;
    uint16_t sflag;
    uint16_t access;
    uint16_t zero;
    uint8_t  eflag;

    // End-user parameters
    uint16_t path_len;
    uint8_t  extra_type;
    uint16_t extra_len;
    char     path [1];
};
```

You'll notice that I've divided the `struct _io_connect` structure into two areas, an “Internal use” part and an “End-user parameters” part.

Internal use part

The first part consists of fields that the resource manager library uses to:

- determine the type of message sent from the client
- validate (ensure that the message is not spoofed)

- track access mode (used by helper functions)

To keep things simple, I recommend that you *always* use the helper functions (the *iofunc_*_default()* ones) in *all* connect functions. These will return a pass/fail indication, and after that point, you can then use the "End-user parameters" members within the connect function.

End-user parameter part

The second half of the members directly concern your implementation of the connect functions:

path_len **and** ***path***

The pathname (and its length) that's the *operand* (i.e., the pathname you're operating on).

extra_type **and** ***extra_len***

Additional parameters (pathnames, for example) relevant to the connect function.

To get a sense of how the *path* member is used as "the pathname you're operating on," let's examine something like the *rename()* function. This function takes two pathnames; the "original" pathname and the "new" pathname. The original pathname is passed in *path*, because it's the thing being worked on (it's the filename that's undergoing the name change). The new pathname is the argument to the operation. You'll see that the *extra* parameter passed to the connect functions conveniently contains a pointer to the argument of the operation—in this case, the new pathname. (Implementation-wise, the new pathname is stored just past the original pathname in the *path* pointer, with alignment taken into consideration, but you don't have to do anything about this—the *extra* parameter conveniently gives you the correct pointer.)

© 2020, QNX Software Systems Limited

Alphabetical listing of connect and I/O functions

This section gives an alphabetical listing of the connect and I/O function entry points that you can fill in (the two tables passed to *resmgr_attach()*). Remember that if you simply call *iofunc_func_init()*, all these entries will be filled in with the appropriate defaults; you'd want to modify a particular entry only if you wish to handle that particular message. In the "Examples" section, below, you'll see some examples of the common functions.

It may seem confusing at first, but note that there are in fact *two* unblock outcalls—one is a connect function and one is an I/O function. This is correct; it's a reflection of *when* the unblock occurs. The connect version of the unblock function is used when the kernel unblocks the client immediately after the client has sent the connect message; the I/O version of the unblock function is used when the kernel unblocks the client immediately after the client has sent an I/O message.

Access control list I/O function handler

Prototype:

```
int (*acl)( resmgr_context_t *ctp,
            io_acl_t *msg,
            RESMGR_OCB_T *ocb )
```

Classification:

I/O

Default handler:

iofunc_acl_default()

Helper functions:

iofunc_acl()

Client functions:

acl_*()

Messages:

_IO_ACL

Data structure:

```
struct _io_acl {
        uint16_t                        type;
        uint16_t                        combine_len;
```

```
        uint32_t                        subtype;
        int32_t                         zero[2];
        /* struct _acl_header           hdr; */
        /* void                         acl_data */
};

enum _io_acl_subtypes {
        _IO_ACL_GET,
        _IO_ACL_SET,
        _IO_ACL_CHK
};

struct _io_acl_reply {
        uint32_t                        zero[4];
        /* struct _acl_header           hdr; */
        /* void                         acl_data */
};

typedef union {
        struct _io_acl                  i;
        struct _io_acl_reply            o;
} io_acl_t;
```

Description:

Gets, sets, or checks the access control list (ACL).

Returns:

The status via the helper macro *_RESMGR_STATUS()* and the reply buffer (with reply data, if required).

Referenced by:

resmgr_io_funcs_t I/O table

Permission checking:

The default implementation *iofunc_acl_default()* calls the helper function *iofunc_acl())*, which does not do any permission checking and none is required.

Change file mode I/O function handler

Prototype:

```
int (*chmod) ( resmgr_context_t *ctp,
               io_chmod_t *msg,
               RESMGR_OCB_T *ocb )
```


© 2020, QNX Software Systems Limited

Classification:

I/O function

Default handler:

iofunc_chmod_default()

Helper functions:

iofunc_ability_check(), *iofunc_chmod()*

Client functions:

chmod(), *fchmod()*

Messages:

_IO_CHMOD

Data structure:

```
struct _io_chmod {
  uint16_t type;
  uint16_t combine_len;
  mode_t   mode;
};

typedef union {
  struct _io_chmod i;
} io_chmod_t;
```

Description: Responsible for changing the mode for the resource identified by the passed *ocb* to the value specified by the *mode* message member.

Returns:

The status via the helper macro *_RESMGR_STATUS()*.

Referenced by:

resmgr_io_funcs_t I/O table

Permission checking:

The default implementation *iofunc_chmod_default()* calls the helper function *iofunc_chmod()* to check that the client is the owner of the file (i.e., that it has the same user ID) and that it's not a read-only filesystem. The helper function also prevents the user from setting the sticky bit and the set group ID bit unless it is **root**.

Change ownership I/O function handler

Prototype:

```
int (*chown)( resmgr_context_t *ctp,
              io_chown_t *msg,
              RESMGR_OCB_T *ocb )
```

Classification:

I/O function

Default handler:

iofunc_chown_default()

Helper functions:

iofunc_ability_check(), *iofunc_chown()*

Client functions:

chown(), *fchown()*

Messages:

_IO_CHOWN

Data structure:

```
struct _io_chown {
  uint16_t type;
  uint16_t combine_len;
  int32_t  gid;
  int32_t  uid;
};

typedef union {
  struct _io_chown i;
} io_chown_t;
```

Description:

Responsible for changing the user ID and group ID fields for the resource identified by the passed *ocb* to *uid* and *gid*, respectively. Note that the mount structure flag IOFUNC_PC_CHOWN_RESTRICTED and the OCB *flag* field should be examined to determine whether the filesystem allows *chown()* to be performed by non-**root** users.

Returns:

The status via the helper macro *_RESMGR_STATUS()*.

© 2020, QNX Software Systems Limited

Referenced by:

resmgr_io_funcs_t I/O table

Permission checking:

The default implementation *iofunc_chown_default()* calls the helper function *iofunc_chown()* to check the IOFUNC_PC_CHOWN_RESTRICTED flag to determine whether the user is allowed to change the user ID on the entry. The helper function also checks whether the client has the `iofunc/chown` (IOFUNC_ABILITY_CHOWN) dynamic ability (using *iofunc_ability_check()*, which allows it to set the user ID as if it were **root**. Finally, the helper checks to make sure it's not a read-only filesystem.

When *iofunc_chown()* sets the new user ID or group ID, it removes the set user ID and set group ID bits if they were set on the file. A value of -1 for the user or group ID means that the caller does not want to change that ID.

Change timestamps I/O function handler

Prototype:

```
int (*utime) ( resmgr_context_t *ctp,
               io_utime_t *msg,
               RESMGR_OCB_T *ocb )
```

Classification:

I/O

Default handler:

iofunc_utime_default()

Helper functions:

iofunc_utime()

Client function: *utime()*

Messages:

_IO_UTIME or _IO_UTIME64

Data structure:

```
struct _io_utime {
    uint16_t                    type;
    uint16_t                    combine_len;
    int32_t                     cur_flag;
    struct __utimbuf32          times;
};
```

```
struct _io_utime64 {
    uint16_t                    type;
    uint16_t                    combine_len;
    int32_t                     cur_flag;
    int64_t                     atime_s;
    int64_t                     atime_ns;
    int64_t                     mtime_s;
    int64_t                     mtime_ns;
};

typedef union {
    struct _io_utime            i;
    struct _io_utime            i64;
} io_utime_t;
```

Description:

Changes the access and modification times to either "now" (if they are zero) or the specified values. Note that this message handler may be required to modify the IOFUNC_ATTR_* flags in the attribute structure as per POSIX rules. You'll almost never use this outcall yourself, but will instead use the POSIX-layer helper function.

Returns:

The status via the helper macro *_RESMGR_STATUS()*.

Referenced by:

resmgr_io_funcs_t I/O table

Permission checking:

The default implementation *iofunc_utime_default()* calls the helper function *iofunc_utime()* to check that the entry is either owned by the current user or the current user has write access to it. The helper also checks for a read-only filesystem. The owner of the entity is allowed to set arbitrary times, while a user with write permission only is only allowed to set the current time (*cur_flag* is 1).

Close duplicate OCB I/O function handler

Prototype:

```
int (*close_dup) ( resmgr_context_t *ctp,
                   io_close_t *msg,
                   RESMGR_OCB_T *ocb )
```

Classification:

I/O function

© 2020, QNX Software Systems Limited

Default handler:

iofunc_close_dup_default()

Helper functions:

iofunc_close_dup()

Client functions:

close(), *fclose()*

Messages:

_IO_CLOSE

Data structure:

```
struct _io_close {
  uint16_t type;
  uint16_t combine_len;
};

typedef union {
  struct _io_close i;
} io_close_t;
```

Description:

This is the *real* function handler for the client's *close()* or *fclose()* function calls.

If you take over this function, you would almost always call *iofunc_close_dup_default()* in addition to the work you do. This is because the base layer keeps track of the number of *open()*, *dup()*, and *close()* messages issued for a particular OCB, and will then synthesize a *close OCB I/O function handler* outcall (see below) when the *last close()* message has been received for a particular OCB.

Note that the receive IDs present in `ctp->rcvid` may not necessarily match up with those passed to the open connect function handler.

If your resource manager leaves clients blocked (e.g., on *read()*, *write()*, or *devctl()* calls), you'll need to take over this function, so that you can keep track of which clients are blocked, and unblock them if necessary. For more information, see "Unblocking if someone closes the file descriptor" in the Unblocking Clients and Handling Interrupts chapter of *Writing a Resource Manager*.

Returns:

The status via the helper macro *_RESMGR_STATUS()*.

Referenced by:

resmgr_io_funcs_t I/O table

Permission checking:

The default implementation *iofunc_close_dup_default()* calls the helper function *iofunc_close_dup()*, which does not do any permission checking.

Close OCB I/O function handler

Prototype:

```
int (*close_ocb) ( resmgr_context_t *ctp,
                   void *reserved,
                   RESMGR_OCB_T *ocb )
```

Classification:

I/O function (synthesized by library)

Default handler:

iofunc_close_ocb_default()

Helper functions:

none

Client function:

none—synthesized by library

Messages:

none—synthesized by library

Data structure:

```
// synthesized by library
struct _io_close {
  uint16_t type;
  uint16_t combine_len;
};

typedef union {
  struct _io_close i;
} io_close_t;
```


© 2020, QNX Software Systems Limited

Description:

This is the function that gets synthesized by the base-layer library when the last *close()* has been received for a particular OCB. This is where you'd perform any final cleanup you needed to do before the OCB is destroyed. Note that the receive ID present in `ctp->rcvid` is zero, because this function is synthesized by the library and doesn't necessarily correspond to any particular message.

Returns:

The status via the helper macro *_RESMGR_STATUS()*.

Referenced by:

resmgr_io_funcs_t I/O table

Permission checking:

The default implementation *iofunc_close_ocb_default()* and helper function *iofunc_close_ocb()* do not do permission checking. No permission checking is required. You should only need to implement this function if you created your own OCB (see also *iofunc_close_ocb_calloc* and *iofunc_close_ocb_free*).

Create file link connect function handler

Prototype:

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

Classification:

Connect

Default handler:

none

Helper functions:

iofunc_link()

Client function:

link()

Messages:

_IO_CONNECT with subtype _IO_CONNECT_LINK

Data structure:

```
struct _io_connect {
  // internal fields (as described above)
  uint16_t path_len;
  uint8_t  extra_type;
  uint16_t extra_len;
  char     path [1];
};

struct _io_connect_link_reply {
  uint32_t reserved1;
  uint32_t file_type;
  uint8_t  eflag;
  uint8_t  reserved2[1];
  uint16_t chroot_len;
  uint32_t umask;
  uint16_t nentries;
  uint16_t path_len;
};

struct _io_connect_ftype_reply {
  uint16_t status;      /* Typically an errno */
  uint16_t reserved;
  uint32_t file_type;   /* _FTYPE_? in sys/ftype.h */
};

typedef union {
  struct _io_connect              connect;
  struct _io_connect_link_reply   link_reply;
  struct _io_connect_ftype_reply  ftype_reply;
} io_link_t;

typedef union _io_link_extra {
  struct _msg_info               info;
  void                           *ocb;
  char                           path [1];
  struct _io_resmgr_link_extra   resmgr;
} io_link_extra_t;
```


© 2020, QNX Software Systems Limited

Description:

Creates a new link with the name given in the *path* member of *msg* to the already-existing pathname specified by the *path* member of *extra* (passed to your function). For convenience, the *ocb* member of *extra* contains a pointer to the OCB for the existing pathname.

Returns:

The status via the helper macro *_RESMGR_STATUS()*.

Referenced by:

resmgr_connect_funcs_t connect table

Permission checking:

The *iofunc_link()* function uses *iofunc_check_access()* with the S_ISUID, S_ISGID, and S_IREAD flags to check access on the OCB for the existing pathname.

Device control I/O function handler

Prototype:

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

Classification:

I/O

Default handler:

iofunc_devctl_default()

Helper functions:

iofunc_devctl(), *iofunc_devctl_verify()*

Client functions:

devctl(), *ioctl()*

Messages:

_IO_DEVCTL

Data structure:

```
struct _io_devctl {
  uint16_t type;
  uint16_t combine_len;
```

```
  int32_t   dcmd;
  uint32_t  nbytes;
  int32_t   zero;
};

struct _io_devctl_reply {
  uint32_t  zero;
  int32_t   ret_val;
  uint32_t  nbytes;
  int32_t   zero2;
};

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

Description:

Performs the device I/O operation as passed from the client's *devctl()* in *dcmd*. The client encodes a direction into the top two bits of *dcmd*, indicating how the *devctl()* is to transfer data (the "to" field refers to the _POSIX_DEVDIR_TO bit; the "from" field refers to the _POSIX_DEVDIR_FROM bit):

to field	*from* field	Meaning
0	0	No data transfer
0	1	Transfer from driver to client
1	0	Transfer from client to driver
1	1	Transfer bidirectionally

In the case of no data transfer, the driver is expected to simply perform the command given in *dcmd*.

Accessing data sent by the client

If the client provides data, it immediately follows the `io_devctl_t` header provided by the *msg* parameter. Before you access any of the data, it is necessary to validate its length.

To be strictly correct, the length of the message from the client is *ctp->size* if *ctp->rcvid* is 0; otherwise, it is (*ctp->info.srcmsglen* – *ctp->offset*). In practice, you can simplify length checking based on whether your handler will deal with short messages only:

1) If the handler will deal with short messages only (less than 1500 bytes), you can always use *ctp->size*. This type of handler is slightly simpler.

2) If the handler has to handle large messages, you can always use (*ctp->info.srcmsglen* – *ctp->offset*).

© 2020, QNX Software Systems Limited

If your handler supports multiple device control operations, you might have operations from both categories.

You need to verify that the actual message length is at least as large as the expected length. If you need to access any part of the message to determine the expected size, you need to ensure that *ctp->size* is large enough to include this data.

After you've verified the message length, you need to access the data. If your device control operation is handling short messages only (category 1 above), the message payload is contained entirely in the message buffer. If your operation has to handle large messages (category 2), you probably have to use *resmgr_msgreadv()* to read at least part of the payload.

Returning data to the client

If you can, before you perform a request, you should validate that the amount of data that will be returned plus the size of the `io_devctl_t` header is less than or equal to *ctp->info.dstmsglen*. Otherwise, the client will not receive all the data that you send. Although, it is conventional to set *nbytes* in the `io_devctl_t` output header to the length of the payload being returned, *nbytes* is not accessible to a client using *devctl()* or *devctlv()*. If the amount of data your device control operation returns is variable, it is a good idea that the returned data include the size or a means to calculate it.

Structure padding

Note that the input and output data structures are zero-padded so that they align with each other. This means that the implicit data area begins at the same address in the input and output structures.

Unrecognized commands

If using the helper routine *iofunc_devctl()*, beware that it'll return the constant _RESMGR_DEFAULT in the case where it can't do anything with the *devctl()* message. This return value is there to decouple legitimate *errno* return values from an "unrecognized command" return value. Upon receiving a _RESMGR_DEFAULT, the base-layer library will respond with an *errno* of ENOSYS, which the client's *devctl()* library function will translate into ENOTTY.

Checking the open mode

It's up to your function to check the open mode against the operation; no checking is done anywhere in either the client's *devctl()* library or in the resource manager library. For example, it's possible to open a resource manager "read-only" and then issue a *devctl()* to it telling it to "format the hard disk" (which is very much a "write" operation). It would be prudent to verify the open mode first before proceeding with the operation.

For more information on checking the security and validity of messages, see "*Permission checking*."

Available *dcmd* values

Note that the range of *dcmd* values you can use is limited (0x0000 through 0x0FFF inclusive is reserved for QNX Software Systems). Other values may be in use; take a look through the include files that have the name **<sys/dcmd_*.h>**.

Returns:

The status via the helper macro *_RESMGR_STATUS()* and the reply buffer (with reply data, if required).

For an example, take a look at *"A simple device control I/O function handler example,"* below.

Referenced by:

resmgr_io_funcs_t I/O table

Permission checking:

The default implementation *iofunc_devctl_default()* handler does not perform permission checking because this handler does not implement any functionality that needs protecting. A resource manager that implements *devctl()* must apply permission checking according to the command to be executed. This is complicated because of the flexible nature of *devctl()* operations. For example, when the baud rate of a serial device is changed, it is acceptable to open the device for read-only access and then perform a *devctl()* to change the baud rate. Although this operation performs a modification, it does not require the file descriptor to have been opened for write. Consequently, each individual *devctl()* command often needs to have appropriate permission checks.

Possible methods for dealing with *devctl()* include the following strategies:

- Enforce a global permission check based on the file open mode. For example, it might be acceptable for a resource manager to insist that no *devctl()* operation be performed at all if the OCB has not been opened for write access, as shown in the following code (placed at the beginning of the *devctl()* handler function):

  ```
  if (!(ocb->ioflag & _IO_FLAG_WR)) {
      return EPERM;
  }
  ```

- Perform command-specific permission checking that inspects each *devctl()* operation to determine the privileges that must be enforced depending on the specific function of the command. You can also implement a check similar to the following one, which verifies that the file was opened with read permissions:

  ```
  if ( (ocb->ioflag & _IO_FLAG_RD) == 0 ) {
      return EPERM;
  }
  ```

QNX does not recommend using *iofunc_devctl_verify()* to perform validation.

For an example handler that includes permission and length checking, see "*A simple device control I/O function handler example*".

© 2020, QNX Software Systems Limited

Duplicate a file descriptor I/O function handler

Prototype

```
int (*dup) ( resmgr_context_t *ctp,
             io_dup_t *msg,
             RESMGR_OCB_T *ocb )
```

Classification:

I/O

Default handler:

NULL—handled by base layer

Helper functions:

none

Client functions:

dup(), *dup2()*, *fcntl()*, *fork()*, *spawn*()*, *vfork()*

Messages:

_IO_DUP

Data structure:

```
struct _io_dup {
  uint16_t            type;
  uint16_t            combine_len;
  struct _msg_info32  info;
  uint32_t            reserved;
  uint32_t            key;
};

typedef union {
  struct _io_dup   i;
} io_dup_t;
```

Description:

This is the *dup()* message handler. As with the close duplicate OCB I/O function handler, you won't likely handle this message yourself. Instead, the base-layer library handles it.

Returns:

The status via the helper macro *_RESMGR_STATUS()*.

Referenced by:

resmgr_io_funcs_t I/O table

Permission checking:

The default functionality and permission checking is handled by the base layer (regardless of whether you use the default implementation). The base layer verifies one of the following conditions:

- the process requesting a *dup()* is the same as the process that opened the original file descriptor
- the *MsgKeyData()* is valid
- the client has the dynamic ability `iofunc/dup` (IOFUNC_ABILITY_DUP) (using *iofunc_ability_check()*)

There are no additional permission checks that need to be performed in a resource manager that implements this function.

File descriptor information I/O function handler

Prototype:

```
int (*fdinfo) ( resmgr_context_t *ctp,
                io_fdinfo_t *msg,
                RESMGR_OCB_T *ocb )
```

Classification:

I/O

Default handler:

iofunc_fdinfo_default()

Helper functions:

iofunc_fdinfo()

Client function:

iofdinfo()

Messages:

_IO_FDINFO

Data structure:

```
struct _io_fdinfo {
  uint16_t        type;
  uint16_t        combine_len;
  uint32_t        flags;
  uint32_t        path_len;
```


© 2020, QNX Software Systems Limited

```
  uint32_t          reserved;
};

struct _io_fdinfo_reply {
  uint32_t          zero [2];
  struct _fdinfo    info;
};

typedef union {
  struct _io_fdinfo        i;
  struct _io_fdinfo_reply  o;
} io_fdinfo_t;
```

Description:

This function is used to allow clients to retrieve information directly about the attributes and pathname which is associated with a file descriptor. The client-side function *iofdinfo()* is used. The path string implicitly follows the `struct _io_fdinfo_reply` data structure. Use of the default function is sufficient for discretely-manifested pathname resource managers.

Returns:

The length of the path string being returned is set via the helper macro *_IO_SET_FDINFO_LEN()*.

Referenced by:

resmgr_io_funcs_t I/O table

Permission checking:

The default implementation *iofunc_fdinfo_default()* checks the length of the path variable and then calls the helper function *iofunc_fdinfo()*. The helper function does not do any permission checking and none is required.

File locking I/O function handler

Prototype:

```
int (*lock) ( resmgr_context_t *ctp,
              io_lock_t *msg,
              RESMGR_OCB_T *ocb )
```

Classification:

I/O

Default handler:

iofunc_lock_default()

Helper functions:

iofunc_lock()

Client functions:

fcntl(), *lockf()*, *flock()*

Messages:

_IO_LOCK

Data structure:

```
struct _io_lock {
  uint16_t              type;
  uint16_t              combine_len;
  uint32_t              subtype;
  uint32_t              nbytes;
};

struct _io_lock_reply {
  uint32_t              zero [3];
};

typedef union {
  struct _io_lock       i;
  struct _io_lock_reply o;
} io_lock_t;
```

Description:

This provides advisory range-based file locking for a device. The default function is most likely sufficient for most resource managers.

Returns:

The status via the helper macro *_RESMGR_STATUS()*.

Referenced by:

resmgr_io_funcs_t I/O table

Permission checking:

The default implementation *iofunc_lock_default()* calls the helper function *iofunc_lock()*.The helper function ensures that write locks can only be set if you opened the file for writing, read locks can only be set if you opened the file for reading, and exclusive locks can only be set if you opened the file for reading or writing.

© 2020, QNX Software Systems Limited

Flush data I/O function handler

Prototype:

```
int (*sync) ( resmgr_context_t *ctp,
              io_sync_t *msg,
              RESMGR_OCB_T *ocb )
```

Classification:

I/O

Default handler:

iofunc_sync_default()

Helper functions:

iofunc_sync_verify(), *iofunc_sync()*

Client functions:

fsync(), *fdatasync()*

Messages:

_IO_SYNC

Data structure:

```
struct _io_sync {
  uint16_t        type;
  uint16_t        combine_len;
  uint32_t        flag;
};

typedef union {
  struct _io_sync i;
} io_sync_t;
```

Description:

This is the entry point for a flush command. The helper function *iofunc_sync()* is passed the *flag* member from the input message, and returns one of the following values, which indicate what actions your resource manager must take:

- O—do nothing.
- O_SYNC—everything associated with the file (including the file contents, directory structures, inodes, etc.) must be present and recoverable from media.
- O_DSYNC—only the data portion of the file must be present and recoverable from media.

Note that this outcall will occur only if you've agreed to provide sync services by setting the mount structure flag.

Returns:

Returns the status via the helper macro *_RESMGR_STATUS()*.

Referenced by:

resmgr_io_funcs_t I/O table

Permission checking:

The default implementation *iofunc_sync_default()* calls the helper function *iofunc_sync_verify()*, but does not actually synchronize anything after verifying. The *iofunc_sync_verify()* function verifies that the filesystem was mounted with the IOFUNC_PC_SYNC_IO flag. The *iofunc_sync()* helper function verifies that it is not a read-only filesystem when it determines whether synchronization is needed.

Lock OCB I/O function handler

Prototype:

```
int (*lock_ocb) ( resmgr_context_t *ctp,
                  void *reserved,
                  RESMGR_OCB_T *ocb )
```

Classification:

I/O (synthesized by library)

Default handler:

iofunc_lock_ocb_default()

Helper functions:

none

Client functions:

all

Messages:

none—synthesized by library

Data structure:

none

© 2020, QNX Software Systems Limited

Description:

This function is responsible for locking the attributes structure pointed to by the OCB. This is done to ensure that only one thread at a time is operating on both the OCB and the corresponding attributes structure. The lock (and corresponding unlock) functions are synthesized by the resource manager library before and after completion of message handling. See the section on *"Combine messages"* above for more details. You'll almost never use this outcall yourself; instead, use the POSIX-layer default function.

Returns:

The status via the helper macro *_RESMGR_STATUS()*.

Referenced by:

resmgr_io_funcs_t I/O table

Permission checking:

The default implementation *iofunc_lock_ocb_default()* does not do any permission checking and has no helper function. No permission checking is required.

Make filesystem node connect function handler

Prototype:

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

Classification:

Connect

Default handler:

none

Helper functions:

iofunc_mknod()

Client functions:

mknod(), *mkdir()*, *mkfifo()*

Messages:

_IO_CONNECT, subtype _IO_CONNECT_MKNOD

Data structure:

```
struct _io_connect {
  // internal fields (as described above)
  uint16_t path_len;
  uint8_t  extra_type;
  uint16_t extra_len;
  char     path [1];
};

struct _io_connect_link_reply {
  uint32_t reserved1;
  uint32_t file_type;
  uint8_t  eflag;
  uint8_t  reserved2[1];
  uint16_t chroot_len;
  uint32_t umask;
  uint16_t nentries;
  uint16_t path_len;
};

struct _io_connect_ftype_reply {
  uint16_t status;      /* Typically an errno */
  uint16_t reserved;
  uint32_t file_type;   /* _FTYPE_? in sys/ftype.h */
};

typedef union {
    struct _io_connect                  connect;
    struct _io_connect_link_reply       link_reply;
    struct _io_connect_ftype_reply      ftype_reply;
} io_mknod_t;
```

Description:

Creates a new filesystem entry point. The message is issued to create a file, named by the *path* member, using the filetype encoded in the *mode* member (from the "internal fields" part of the `struct _io_connect` structure, not shown).

This is really used only for the *mkfifo()*, *mkdir()*, and *mknod()* client functions.

Returns:

The status via the helper macro *_RESMGR_STATUS()*.

© 2020, QNX Software Systems Limited

Referenced by:

resmgr_connect_funcs_t connect table

Permission checking:

QNX Neutrino does not use block and character device nodes for access to device drivers; instead, it relies on device drivers to register as resource managers. Therefore, *mknod()* is restricted to creating files, directories, and FIFOs. The helper function *iofunc_mknod()* checks for write access in the parent directory.

Map memory I/O function handler

Prototype:

```
int (*mmap) ( resmgr_context_t *ctp,
              io_mmap_t *msg,
              RESMGR_OCB_T *ocb )
```

Classification:

I/O

Default handler:

iofunc_mmap_default()

Helper functions:

iofunc_mmap()

Client functions:

mmap(), *munmap()*, *mmap_device_io()*, *mmap_device_memory()*

Messages:

_IO_MMAP

Data structure:

```
struct _io_mmap {
  uint16_t              type;
  uint16_t              combine_len;
  uint32_t              prot;
  uint64_t              offset;
  struct _msg_info32    info;
  uint32_t              required_prot;
  uint32_t              zero [5];
};
```

```
struct _io_mmap_reply {
  uint32_t              zero;
  uint32_t              allowed_prot;
  uint64_t              offset;
  int32_t               coid;
  int32_t               fd;
};

typedef union {
  struct _io_mmap       i;
  struct _io_mmap_reply o;
} io_mmap_t;
```

Description:

Allows the memory manager to map files from your resource manager into memory. Generally, you should not code this function yourself (use the defaults provided by *iofunc_func_init()*—the default handler), unless you specifically wish to disable the functionality (for example, a serial port driver could choose to return ENOSYS, because it doesn't make sense to support this operation).

When a client calls *mmap()*, the function sends a _MEM_MMAP message to the memory manager, which then sends an _IO_MMAP message to the appropriate resource manager. The resource manager replies to the memory manager, which in turn replies to the client.

Note that a side effect of calling this function is that an OCB will be created (i.e., *iofunc_ocb_calloc()* or your replacement for it will be called), but this should have no consequences to a properly implemented resource manager.

Returns:

The status via the helper macro *_RESMGR_STATUS()*.

Referenced by:

resmgr_io_funcs_t I/O table

Permission checking:

The default implementation *iofunc_mmap_default()* calls the helper function *iofunc_mmap()*. The helper function checks that the requested mapping permissions do not conflict with the open mode for the file descriptor and that the mountpoint is not read only. The helper also checks that the file is executable before setting PROT_EXEC in *allowed_prot*. The helper obtains an exclusive write lock on the file if it is being mapped with execute permissions.

© 2020, QNX Software Systems Limited

Mount connect function handler

Prototype:

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

Classification:

Connect

Default handler:

none

Client functions:

mount(), *umount()*

Helper functions:

none

Messages:

_IO_CONNECT with the _IO_CONNECT_MOUNT subtype.

Data structure:

```
struct _io_connect {
  // internal fields (as described above)
  uint16_t path_len;
  uint8_t  extra_type;
  uint16_t extra_len;
  char     path [1];
};

struct _io_connect_link_reply {
  uint32_t reserved1;
  uint32_t file_type;
  uint8_t  eflag;
  uint8_t  reserved2[1];
  uint16_t chroot_len;
  uint32_t umask;
  uint16_t nentries;
  uint16_t path_len;
```

```
};

struct _io_connect_ftype_reply {
  uint16_t status;       /* Typically an errno */
  uint16_t reserved;
  uint32_t file_type;    /* _FTYPE_? in sys/ftype.h */
};

typedef union {
  struct _io_connect              connect;
  struct _io_connect_link_reply   link_reply;
  struct _io_connect_ftype_reply  ftype_reply;
} io_mount_t;
```

Description:

This function is called whenever a *mount()* or *umount()* client function sends your resource manager a message. For more information about the mount connect function handler, see "Handling *mount()*" in the Handling Other Messages chapter of *Writing a Resource Manager*.

Returns:

The status via the helper macro *_IO_SET_CONNECT_RET()*.

Referenced by:

resmgr_connect_funcs_t connect table

Permission checking:

QNX Neutrino does not provide a default mount function. Any resource manager that implements the mount function should restrict mount abilities to sufficiently privileged clients only. For additional information, see "Mount in the resource manager" in *Writing a Resource Manager*.

Open connect function handler

Prototype:

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

Classification:

Connect

© 2020, QNX Software Systems Limited

Default handler:

iofunc_open_default()

Helper functions:

iofunc_ability_check(), *iofunc_ocb_attach()*, *iofunc_open()*

Client functions:

open(), *fopen()*, *sopen()* (and others)

Messages:

_IO_CONNECT with one of _IO_CONNECT_COMBINE, _IO_CONNECT_COMBINE_CLOSE or _IO_CONNECT_OPEN subtypes.

Data structure:

```
struct _io_connect {
  // internal fields (as described above)
  uint16_t path_len;
  uint8_t  extra_type;
  uint16_t extra_len;
  char     path [1];
};

struct _io_connect_link_reply {
  uint32_t reserved1;
  uint32_t file_type;
  uint8_t  eflag;
  uint8_t  reserved2[1];
  uint16_t chroot_len;
  uint32_t umask;
  uint16_t nentries;
  uint16_t path_len;
};

struct _io_connect_ftype_reply {
  uint16_t status;      /* Typically an errno */
  uint16_t reserved;
  uint32_t file_type;   /* _FTYPE_? in sys/ftype.h */
};

typedef union {
  struct _io_connect             connect;
  struct _io_connect_link_reply  link_reply;
```

```
    struct _io_connect_ftype_reply ftype_reply;
} io_open_t;
```

Description:

This is the handler function for new file descriptors. It checks that the client indeed has the appropriate permissions to open the file, binds the OCB to the internal library structures (via *resmgr_bind_ocb()*, or *iofunc_ocb_attach()*), and returns an *errno*. Note that not all input and output structure members are relevant for this function.

Returns:

The status via the helper macro *_IO_SET_CONNECT_RET()*.

Referenced by:

resmgr_connect_funcs_t connect table

Permission checking:

When you override the default function, QNX recommends that you call *iofunc_open_default()* (which implements several permission checks depending on the access requested) before you implement any additional functionality. Note that because a file can be opened without read or write permissions, it is not sufficient to check that *iofunc_open_default()* (or the related *iofunc_open()*) returns EOK before you implement things like locking (e.g., for a resource that should only be opened by one client at a time).

Open file descriptor I/O function handler

Prototype:

```
int (*openfd) ( resmgr_context_t *ctp,
                io_openfd_t *msg,
                RESMGR_OCB_T *ocb )
```

Classification:

I/O

Default handler:

iofunc_openfd_default()

Helper functions:

iofunc_ability_check(), *iofunc_openfd()*

Client function:

openfd()

© 2020, QNX Software Systems Limited

Messages:

_IO_OPENFD

Data structure:

```
struct _io_openfd {
  uint16_t            type;
  uint16_t            combine_len;
  uint32_t            ioflag;
  uint16_t            sflag;
  uint16_t            reserved1;
  struct _msg_info32  info;
  uint32_t            reserved2;
  uint32_t            key;
};

typedef union {
  struct _io_openfd i;
} io_openfd_t;
```

Description:

This function is similar to the handler provided for the open connect function handler, except that instead of a pathname, an already-open file descriptor is passed (by virtue of passing you the *ocb* in the function call).

Returns:

The status via the helper macro *_RESMGR_STATUS()*.

Referenced by:

resmgr_io_funcs_t I/O table

Permission checking:

The default implementation *iofunc_openfd_default()* checks the extended type and then calls the helper function *iofunc_openfd*. The helper function makes sure that the new permissions are not less restrictive than the access mode that is currently open.

pathconf I/O function handler

Prototype:

```
int (*pathconf) ( resmgr_context_t *ctp,
                  io_pathconf_t *msg,
                  RESMGR_OCB_T *ocb )
```

Classification:

I/O

Default handler:

iofunc_pathconf_default()

Helper functions:

iofunc_pathconf()

Client functions:

fpathconf(), *pathconf()*

Messages:

_IO_PATHCONF

Data structure:

```
struct _io_pathconf {
  uint16_t            type;
  uint16_t            combine_len;
  short               name;
  uint16_t            zero;
};

typedef union {
  struct _io_pathconf i;
} io_pathconf_t;
```

Description:

The handler for this message is responsible for returning the value of the configurable parameter *name* for the resource associated with this OCB. Use the default function and add additional cases for the *name* member as appropriate for your device.

Returns:

The status via the helper macro *_IO_SET_PATHCONF_VALUE()* and the data via message reply.

Referenced by:

resmgr_io_funcs_t I/O table

Permission checking:

The default implementation *iofunc_pathconf_default()* and helper function *iofunc_pathconf()* do not perform any permission checking.

© 2020, QNX Software Systems Limited

Power I/O function handler

Prototype:

```
int (*power) ( resmgr_context_t * ctp,
               io_power_t * msg,
               RESMGR_OCB_T * ocb )
```

This function is reserved by QNX Software Systems for future use. You should initialize the I/O table using *iofunc_func_init()* and not modify this entry.

Read I/O function handler

Prototype:

```
int (*read) ( resmgr_context_t *ctp,
              io_read_t *msg,
              RESMGR_OCB_T *ocb )
```

Classification:

I/O

Default handler:

iofunc_read_default()

Helper functions:

iofunc_read_verify()

Client functions:

read(), *readdir()*

Messages:

_IO_READ, _IO_READ64

Data structure:

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

struct _io_read64 {
```

```
    uint16_t            type;
    uint16_t            combine_len;
    uint32_t            nbytes;
    uint32_t            xtype;
    uint32_t            nbytes_hi;
};

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

Description:

Responsible for reading data from the resource. The client specifies the number of bytes it's prepared to read in the *nbytes* input member. You return the data, advance the offset in the OCB, and update the appropriate time fields.

Note the following:

- The client library uses the _IO_READ64 form only when the length is greater than 4 GB. You can use the *_IO_READ_GET_NBYTES()* macro (defined in **<sys/iofunc.h>**) to determine the number of bytes.
- The *xtype* member may specify a per-read-message override flag. This should be examined. If you don't support any extended override flags, you should return an EINVAL. We'll see the handling of one particularly important (and tricky!) override flag called _IO_XTYPE_OFFSET in the *read I/O function handler* and *write I/O function handler* examples below.
- The _IO_READ or _IO_READ64 message arrives not only for regular files, but also for reading the contents of directories. You must ensure that you return an integral number of `struct dirent` members in the directory case. For more information about returning directory entries, see the example in the "Advanced topics" section under "*Returning directory entries.*"
- In QNX Neutrino 7.0 or later, *readdir()* sets *xtype* to _IO_XTYPE_READDIR. Your handler for this message can give an error if someone tries to *read()* from a directory.

You should call the helper function *iofunc_read_verify()* to ascertain that the file was opened in a mode compatible with reading. You should also call the *iofunc_sync_verify()* function to verify if the data needs to be synchronized to the medium. (For a *read()*, that means that the data returned is guaranteed to be on-media.)

Returns:

The number of bytes read, or the status, via the helper macro *_IO_SET_READ_NBYTES()*, and the data itself via message reply. For an example of returning just data, take a look at *"A simple read I/O function handler example"* below. For a more complicated example of returning both data and directory entries, look in the "Advanced topics" section under "*Returning directory entries.*"

© 2020, QNX Software Systems Limited

Referenced by:

resmgr_io_funcs_t I/O table

Permission checking:

The default implementation *iofunc_read_default()* calls *iofunc_read_verify()* and then doesn't return any data. Because it is possible to open a file without read or write permissions, you must validate that the OCB was opened for read in each call. The *iofunc_read_verify()* function does this by default. In addition, the resource manager should validate the extended type (*xtype*; see "If you aren't expecting extended types (*xtype*)" in *Writing a Resource Manager*).

Read link connect function handler

Prototype:

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

Classification:

Connect

Default handler:

none

Helper functions:

iofunc_readlink()

Client function:

readlink()

Messages:

_IO_CONNECT with subtype _IO_CONNECT_READLINK

Data structure:

```
struct _io_connect {
  // internal fields (as described above)
  uint16_t path_len;
  uint8_t  extra_type;
  uint16_t extra_len;
  char     path [1];
};
```

```
struct _io_connect_link_reply {
  uint32_t reserved1;
  uint32_t file_type;
  uint8_t  eflag;
  uint8_t  reserved2[1];
  uint16_t chroot_len;
  uint32_t umask;
  uint16_t nentries;
  uint16_t path_len;
};

struct _io_connect_ftype_reply {
  uint16_t status;       /* Typically an errno */
  uint16_t reserved;
  uint32_t file_type;    /* _FTYPE_? in sys/ftype.h */
};

typedef union {
  struct _io_connect              connect;
  struct _io_connect_link_reply   link_reply;
  struct _io_connect_ftype_reply  ftype_reply;
} io_readlink_t;
```

Description:

Responsible for reading the contents of a symbolic link as specified by the *path* member of the input structure. The bytes returned are the contents of the symbolic link; the status returned is the number of bytes in the reply. A valid return should be done only for a symbolic link; all other accesses should return an error code.

Returns:

The status via the helper macro *_RESMGR_STATUS()* and the data via message reply.

Referenced by:

resmgr_connect_funcs_t connect table

Permission checking:

The *iofunc_readlink()* function does not perform any permission checking.

© 2020, QNX Software Systems Limited

Rename connect function handler

Prototype:

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

Classification:

Connect

Default handler:

none

Helper functions:

iofunc_rename()

Client function:

rename()

Messages:

_IO_CONNECT with subtype _IO_CONNECT_RENAME

Data structure:

```
struct _io_connect {
  // internal fields (as described above)
  uint16_t path_len;
  uint8_t  extra_type;
  uint16_t extra_len;
  char     path [1];
};

struct _io_connect_link_reply {
  uint32_t reserved1;
  uint32_t file_type;
  uint8_t  eflag;
  uint8_t  reserved2[1];
  uint16_t chroot_len;
  uint32_t umask;
  uint16_t nentries;
  uint16_t path_len;
```

```
};

struct _io_connect_ftype_reply {
  uint16_t status;       /* Typically an errno */
  uint16_t reserved;
  uint32_t file_type;    /* _FTYPE_? in sys/ftype.h */
};

typedef union _io_rename_extra {
  char     path [1];
} io_rename_extra_t;

typedef union {
  struct _io_connect              connect;
  struct _io_connect_link_reply   link_reply;
  struct _io_connect_ftype_reply  ftype_reply;
} io_rename_t;
```

Description:

Performs the rename operation, given the new name in *path* and the original name in the *path* member of the passed *extra* parameter. Implementation note: the *pathname* of the original name is given (rather than an OCB) specifically for the case of handling a rename of a file that's hard-linked to another file. If the OCB were given, there would be no way to tell apart the two (or more) versions of the hard-linked file.

This function will be called only with two filenames that are on the same filesystem (same device). Therefore, there's no need to check for a case where you'd return EXDEV. This doesn't prevent you from returning EXDEV if you don't wish to perform the *rename()* yourself (for example, it may be very complicated to do the rename operation from one directory to another). In the case of returning EXDEV, the shell utility `mv` will perform a `cp` followed by an `rm` (the C library function *rename()* will do no such thing—it will return only an *errno* of EXDEV).

Also, all symlinks will be resolved, where applicable, before this function is called, and the pathnames passed will be absolute and rooted in the filesystem for which this resource manager is responsible.

Returns:

The status via the helper macro *_RESMGR_STATUS()*.

Referenced by:

resmgr_connect_funcs_t connect table

Permission checking:

The default *iofunc_rename()* function relies on permission checking in *iofunc_link()* and *iofunc_unlink()*.

© 2020, QNX Software Systems Limited

Register for notification I/O function handler

Prototype:

```
int (*notify) ( resmgr_context_t *ctp,
                io_notify_t *msg,
                RESMGR_OCB_T *ocb )
```

Classification:

I/O

Default handler:

none

Helper functions:

iofunc_notify(), *iofunc_notify_remove()*, *iofunc_notify_remove_strict()*, *iofunc_notify_trigger()*, *iofunc_notify_trigger_strict()*

Client functions:

select(), *ionotify()*

Messages:

_IO_NOTIFY, _IO_NOTIFY64

Data structure:

```
struct _io_notify {
    uint16_t                    type;
    uint16_t                    combine_len;
    int32_t                     action;
    int32_t                     flags;
    struct __sigevent32         event;

    /* Following fields only valid if (flags & _NOTIFY_COND_EXTEN) */
    int32_t                     mgr[2];    /* For use by manager */
    int32_t                     flags_extra_mask;
    int32_t                     flags_exten;
    int32_t                     nfds;
    int32_t                     fd_first;
    int32_t                     nfds_ready;
    int64_t                     timo;
    /* struct pollfd            fds[nfds]; */
};
```

```
struct _io_notify64 {
    uint16_t                    type;
    uint16_t                    combine_len;
    int32_t                     action;
    int32_t                     flags;
    struct __sigevent32         old_event;

    /* Following fields only valid if (flags & _NOTIFY_COND_EXTEN) */
    int32_t                     mgr[2];    /* For use by manager */
    int32_t                     flags_extra_mask;
    int32_t                     flags_exten;
    int32_t                     nfds;
    int32_t                     fd_first;
    int32_t                     nfds_ready;
    int64_t                     timo;
    union {
        struct __sigevent32     event32;
        struct __sigevent64     event64;
    };
    /* struct pollfd            fds[nfds]; */
};

struct _io_notify_reply {
    uint32_t                    zero;
    uint32_t                    flags;   /* actions above */
    int32_t                     flags2;  /* flags above */
    struct __sigevent32         event;

    /* Following fields only updated by new managers (if valid) */
    int32_t                     mgr[2];    /* For use by manager */
    int32_t                     flags_extra_mask;
    int32_t                     flags_exten;
    int32_t                     nfds;
    int32_t                     fd_first;
    int32_t                     nfds_ready;
    int64_t                     timo;
    /* struct pollfd            fds[nfds]; */
};

struct _io_notify_reply64 {
    uint32_t                    zero;
    uint32_t                    flags;  /* actions */
    int32_t                     flags2; /* flags above */
```


© 2020, QNX Software Systems Limited

```
    struct __sigevent32         old_event;

    /* Following fields only updated by new managers (if valid) */
    int32_t                     mgr[2];    /* For use by manager */
    int32_t                     flags_extra_mask;
    int32_t                     flags_exten;
    int32_t                     nfds;
    int32_t                     fd_first;
    int32_t                     nfds_ready;
    int64_t                     timo;
    union {
        struct __sigevent32     event32;
        struct __sigevent64     event64;
    };
    /* struct pollfd            fds[nfds]; */
};

typedef union {
    struct _io_notify           i;
    struct _io_notify64         i64;
    struct _io_notify_reply     o;
    struct _io_notify_reply64   o64;
} io_notify_t;
```

Description:

The handler is responsible for installing, polling, or removing a notification handler. The *action* and *flags* determine the kind of notification operation and conditions; the *event* is a `struct sigevent` structure that defines the notification event (if any) that the client wishes to be signaled with. You'd use the *MsgDeliverEvent()* or *iofunc_notify_trigger()* functions to deliver the *event* to the client.

Returns:

The status via the helper macro *_RESMGR_STATUS()*; the flags are returned via message reply.

Referenced by:

resmgr_io_funcs_t I/O table

Permission checking:

The *iofunc_notify()* helper function does not perform permission checking. However, if an OCB has not been opened for read, it should probably not be allowed to request notification of conditions occurring on that OCB that relate to reading data (e.g., the _NOTIFY_CONDE_RDNORM flag is set). Notifications that are not associated with reading or writing data (e.g., _NOTIFY_CONDE_HUP and _NOTIFY_CONDE_NVAL) do not need to be tied to read or write access. Although the information leakage is quite limited in this case, (e.g., learning when a file descriptor is ready for read or write or

has out-of-band data available), it could potentially be used in various attacks such as race conditions. QNX recommends that you deny a *notify()* call to callers that have not opened the file descriptor for read or write. Thorough permission checking would apply an appropriate decision depending on the flags supplied to the *notify()* function.

For resource managers that deal with user input (e.g., a keyboard), be careful not to leak timing data through the delivery of notify messages. Attackers can use this timing data to build a profile of what the user is typing. Similarly, do not provide notification for process-specific characteristics such as CPU usage and changes in allocated memory.

Resize file I/O function handler

Prototype:

```
int (*space) ( resmgr_context_t *ctp,
               io_space_t *msg,
               RESMGR_OCB_T *ocb )
```

Classification:

I/O

Default handler:

none

Helper functions:

iofunc_space_verify()

Client functions:

fcntl(), *ftruncate()*, *ltrunc()*

Messages:

_IO_SPACE

Data structure:

```
struct _io_space {
  uint16_t        type;
  uint16_t        combine_len;
  uint16_t        subtype;
  short           whence;
  uint64_t        start;
  uint64_t        len;
};

typedef union {
```


© 2020, QNX Software Systems Limited

```
    struct _io_space i;
    uint64_t         o;
  } io_space_t;
```

Description:

This is used to allocate or free space occupied by the resource. The *subtype* parameter indicates whether to allocate (if set to F_ALLOCSP) or deallocate (if set to F_FREESP) storage space. The combination of *whence* and *start* give the location where the beginning of the allocation or deallocation should occur; the member *len* indicates the size of the operation.

Returns:

The number of bytes (size of the resource) via the helper macro *_RESMGR_STATUS()*.

Referenced by:

resmgr_io_funcs_t I/O table

Permission checking:

The *iofunc_space_verify()* helper function verifies that the file is opened with write permissions and that the filesystem is not read only. If the file is executable and has the set user ID bit set, *iofunc_space_verify()* only allows the operation if the caller is **root**. There is no default handler.

Set file position I/O function handler

Prototype:

```
int (*lseek) ( resmgr_context_t *ctp,
               io_lseek_t *msg,
               RESMGR_OCB_T *ocb )
```

Classification:

I/O

Default handler:

iofunc_lseek_default()

Helper functions:

iofunc_lseek()

Client functions:

lseek(), *fseek()*, *rewinddir()*

Messages:

_IO_LSEEK

Data structure:

```
struct _io_lseek {
  uint16_t          type;
  uint16_t          combine_len;
  short             whence;
  uint16_t          flags;
  uint64_t          offset;
};

typedef union {
  struct _io_lseek i;
  uint64_t         o;
} io_lseek_t;
```

Description:

Handles the client's *lseek()* function. Note that a resource manager that handles directories will also need to interpret the _IO_LSEEK message for directory operations. The *whence* and *offset* parameters are passed from the client's *lseek()* function. The routine should adjust the OCB's *offset* parameter after interpreting the *whence* and *offset* parameters from the message and should return the new offset or an error.

The only currently defined flag is _IO_LSEEK_IGNORE_NON_SEEKABLE. The client sets this flag if it wants the resource manager to ignore the request if lseek operations aren't supported (for example, an *aio_read()* on a pipe). In this case, the resource manager shouldn't change the offset.

Returns:

The status via the helper macro *_RESMGR_STATUS()*, and optionally (if no error and if not part of a *combine message*) the current offset.

Referenced by:

resmgr_io_funcs_t I/O table

Permission checking:

The default implementation *iofunc_lseek_default()* and helper function *iofunc_iofunc_lseek()* do not perform permission checking. They allow the client to adjust the current offset in the OCB even if the file was not opened for reading or writing. It is up to subsequent read and write operations on the OCB to perform all check operations related to the offset.

© 2020, QNX Software Systems Limited

Shutdown I/O function handler

Prototype:

```
int (*shutdown) ( resmgr_context_t * ctp,
                  io_shutdown_t *msg,
                  RESMGR_OCB_T *ocb )
```

This function is reserved by QNX Software Systems for future use. You should initialize the I/O table using *iofunc_func_init()* and not modify this entry.

stat I/O function handler

Prototype:

```
int (*stat) ( resmgr_context_t *ctp,
              io_stat_t *msg,
              RESMGR_OCB_T *ocb )
```

Classification:

I/O

Default handler:

iofunc_stat_default()

Helper functions:

iofunc_stat(), *iofunc_stat_format()*

Client functions:

stat(), *lstat()*, *fstat()*

Messages:

_IO_STAT

Data structure:

```
struct _io_stat {
    uint16_t                    type;
    uint16_t                    combine_len;
    union {
       uint32_t                 zero;
       uint32_t                 format;
    };
};
```

```
typedef union {
    struct _io_stat             i;
    struct __stat_t32_2001      o_t32_2001;
    struct __stat_t32_2008      o_t32_2008;
    struct __stat_t64_2008      o_t64_2008;
    struct stat                 o;
} io_stat_t;
```

Description:

Handles the message that requests information about the resource associated with the passed OCB. Note that the attributes structure contains all the information required to fulfill the *stat()* request; the helper function *iofunc_stat_format()* fills a `struct stat` structure based on the attributes structure. Also, the helper function modifies the stored *dev*/*rdev* members to be unique from a single node's point of view (useful for performing *stat()* calls to files over a network). There's almost no reason to write your own handler for this function.

Returns:

The format of the information via the helper macro *_RESMGR_STATUS()*, and the `struct stat` via message reply.

Referenced by:

resmgr_io_funcs_t I/O table

Permission checking:

The default implementation *iofunc_stat_default()* is expected to work on files opened without either read or write permission. The helper function *iofunc_stat()* does not do any permission checking. No additional permission checking is required.

If you are implementing a resource manager that deals with user input, be careful not to leak timing data through the last accessed or last modified timestamps. Similarly, do not use the size parameter to keep track of bytes sent or received over serial ports or other potentially sensitive information that should not be observable.

Unblock connect function handler

Prototype:

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

Classification:

Connect (synthesized by kernel, synthesized by library)

© 2020, QNX Software Systems Limited

Default handler:

none

Helper functions:

iofunc_unblock()

Client function:

none—kernel action due to signal or timeout

Messages:

none—synthesized by library

Data structure:

(See I/O version of the unblock function)

Description:

This is the connect message version of the unblock outcall, synthesized by the library as a result of a kernel pulse due to the client's attempt to unblock during the connect message phase. See the unblock I/O function handler for more details.

Returns:

The status via the helper macro *_RESMGR_STATUS()*.

See the section in the Message Passing chapter, titled *"Using the* _NTO_MI_UNBLOCK_REQ*"* for a detailed discussion of unblocking strategies.

Referenced by:

resmgr_connect_funcs_t connect table

Permission checking:

The *iofunc_unblock()* function does not perform any permission checking.

Unblock I/O function handler

Prototype:

```
int (*unblock) ( resmgr_context_t *ctp,
                 io_pulse_t *msg,
                 RESMGR_OCB_T *ocb )
```

Classification:

I/O (synthesized by kernel, synthesized by library)

Default handler:

iofunc_unblock_default()

Helper functions:

iofunc_unblock()

Client function:

none—kernel action due to signal or timeout

Messages:

none—synthesized by library

Data structure:

pointer to message structure being interrupted

Description:

This is the I/O message version of the unblock outcall, synthesized by the library as a result of a kernel pulse due to the client's attempt to unblock during the I/O message phase. The unblock connect function handler is substantially the same (see the preceding section).

Common to both unblock handlers (connect and I/O) is the characteristic that the client wishes to unblock, but is at the mercy of the resource manager. The resource manager *must* reply to the client's message in order to unblock the client. (This is discussed in the *Message Passing* chapter when we looked at the *ChannelCreate()* flags, particularly the _NTO_CHF_UNBLOCK flag).

Returns:

The status via the helper macro *_RESMGR_STATUS()*.

See the section in the Message Passing chapter, titled *"Using the* _NTO_MI_UNBLOCK_REQ*"* for a detailed discussion of unblocking strategies.

Referenced by:

resmgr_io_funcs_t I/O table

Permission checking:

The default implementation *iofunc_unblock_default()* does not perform any permission checking.

Unlink connect function handler

Prototype:

```
int (*unlink) ( resmgr_context_t *ctp,
                io_unlink_t *msg,
```


© 2020, QNX Software Systems Limited

```
                    RESMGR_HANDLE_T *handle,
                    void *reserved )
```

Classification:

Connect

Default handler:

none

Helper functions:

iofunc_unlink()

Client function:

unlink()

Messages:

_IO_CONNECT with subtype _IO_CONNECT_UNLINK

Data structure:

```
struct _io_connect {
  // internal fields (as described above)
  uint16_t path_len;
  uint8_t  extra_type;
  uint16_t extra_len;
  char     path [1];
};

struct _io_connect_link_reply {
  uint32_t reserved1;
  uint32_t file_type;
  uint8_t  eflag;
  uint8_t  reserved2[1];
  uint16_t chroot_len;
  uint32_t umask;
  uint16_t nentries;
  uint16_t path_len;
};

struct _io_connect_ftype_reply {
  uint16_t status;      /* Typically an errno */
  uint16_t reserved;
```

```
  uint32_t file_type;    /* _FTYPE_? in sys/ftype.h */
};

typedef union {
  struct _io_connect              connect;
  struct _io_connect_link_reply   link_reply;
  struct _io_connect_ftype_reply  ftype_reply;
} io_unlink_t;
```

Description:

Responsible for unlinking the file whose pathname is passed in the input message structure's *path* member.

Returns:

The status via the helper macro *_RESMGR_STATUS()*.

Referenced by:

resmgr_connect_funcs_t connect table

Permission checking:

When you override the default function, QNX recommends that you call *iofunc_unlink()*, which verifies the following conditions before it updates the link counts:

- The client has write permission on the directory that contains the entity being unlinked.
- The entity being unlinked is not "." or "..".
- If it is a directory that is being unlinked, that the directory is empty.
- If it is a directory that is being unlinked, that the client is user ID (UID) 0 and the filesystem allows **root** to unlink directories (IOFUNC_PC_LINK_DIR; see "The optional `iofunc_mount_t` (mount) structure" in *Writing a Resource Manager*).
- If the sticky bit is set, the client is the owner of the entity being unlinked.
- It is a writable filesystem.

If you need to allow users other than UID 0 to unlink directories, re-implement all these checks yourself in your resource manager.

Unlock OCB I/O function handler

Prototype:

```
int (*unlock_ocb) ( resmgr_context_t *ctp,
                    void *reserved,
                    RESMGR_OCB_T *ocb )
```


© 2020, QNX Software Systems Limited

Classification:

I/O (synthesized by library)

Default handler:

iofunc_unlock_ocb_default()

Helper functions:

none

Client functions:

all

Messages:

none—synthesized by library

Data structure:

none

Description:

Inverse of *lock OCB I/O function handler*, above. That is, it's responsible for unlocking the attributes structure pointed to by the OCB. This operation releases the attributes structure so that other threads in the resource manager may operate on it. See the section on *"Combine messages"* above for more details.

Returns:

The status via the helper macro *_RESMGR_STATUS()*.

Referenced by:

resmgr_io_funcs_t I/O table

Permission checking:

The default implementation *iofunc_unlock_ocb_default()* does not do any permission checking and has no helper function. No permission checking is required.

User-defined message I/O function handler

Prototype:

```
int (*msg) ( resmgr_context_t *ctp,
             io_msg_t *msg,
             RESMGR_OCB_T *ocb )
```

Classification:

I/O

Default handler:

none.

Helper functions:

none.

Client function:

none—manually assembled and sent via *MsgSend()*

Messages:

_IO_MSG

Data structure:

```
struct _io_msg {
  uint16_t        type;
  uint16_t        combine_len;
  uint16_t        mgrid;
  uint16_t        subtype;
};

typedef union {
  struct _io_msg i;
} io_msg_t;
```

Description:

Like *devctl()*, the _IO_MSG interface allows you to send miscellaneous requests. The *mgrid* is used to identify a particular manager—you should not perform actions for requests that don't conform to your manager ID. The *subtype* is effectively the command that the client wishes to perform. Any data that's transferred implicitly follows the input structure. Data that's returned to the client is sent on its own, with the status returned via *_RESMGR_STATUS()*.

The methods for validating messages for this handler are the same as for the *device control I/O function handler*, but note the following differences:

- Unlike the device control I/O function handler, which has *devctl()* and *devctlv()*, the user-defined message handler does not have **libc** functions to help send messages.
- The size of the input header is `io_msg_t`, not `io_devctl_t`.
- No output header is defined.

© 2020, QNX Software Systems Limited

Returns:

The status via the helper macro *_RESMGR_STATUS()*.

Referenced by:

resmgr_io_funcs_t I/O table

Permission checking:

There is no default handler or helper for this function. It handles messages sent manually through *MsgSend()*. Any permission checking is specific to the function being implemented. If the message handler is supporting a variety of commands similar to *devctl()*, a per-command access check might be necessary.

Write I/O function handler

Prototype:

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

Classification:

I/O

Default handler:

iofunc_write_default()

Helper functions:

iofunc_write_verify()

Client functions:

write(), *fwrite()*, etc.

Messages:

_IO_WRITE, _IO_WRITE64

Data structure:

```
struct _io_write {
    uint16_t            type;
    uint16_t            combine_len;
    uint32_t            nbytes;
    uint32_t            xtype;
    uint32_t            zero;
```

```
    /* unsigned char    data[nbytes]; */
};

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

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

Description:

This message handler is responsible for getting data that the client wrote to the resource manager. It gets passed the number of bytes the client is attempting to write in the *nbytes* member; the data implicitly follows the input data structure (unless the *xtype* override is _IO_XTYPE_OFFSET; see *"A simple write I/O function handler example"* below!) The implementation will need to re-read the data portion of the message from the client, using *resmgr_msgreadv()* or the equivalent. The return status is the number of bytes actually written or an *errno*.

Note the following:

- To make sure the handler supports writes that are greater than 4 GB (whether the form is _IO_WRITE or _IO_WRITE64), use the *_IO_WRITE_GET_NBYTES()* macro (defined in **<sys/iofunc.h>**).
- You should use the helper function *iofunc_write_verify()* to ascertain that the file was opened in a mode compatible with writing. You should also call the *iofunc_sync_verify()* function to verify if the data needs to be synchronized to the medium.

Returns:

The status via the helper macro *_IO_SET_WRITE_NBYTES()*.

For an example, take a look at *"A simple write I/O function handler example"* below.

Referenced by:

resmgr_io_funcs_t I/O table

Permission checking:

The default implementation *iofunc_write_default()* calls *iofunc_write_verify()* and then silently discards all data written. Because it is possible to open a file without read or write permissions, you must

© 2020, QNX Software Systems Limited

validate that the OCB was opened for write in each call. By default, this validation is done using *iofunc_write_verify()*. In addition, the resource manager should validate the extended type (*xtype*).

Examples

I'm now going to show you a number of "cookbook" examples you can cut and paste into your code, to use as a basis for your projects. These aren't complete resource managers—you'll need to add the thread pool and dispatch "skeleton" shown immediately below, and ensure that your versions of the I/O functions are placed into the I/O functions table *after* you've done the *iofunc_func_init()*, in order to override the defaults!

I'll start with a number of simple examples that show basic functionality for the various resource manager message handlers:

- *read I/O function handler*
- *write I/O function handler*
- *device control I/O function handler*

And then in the advanced topics section, we'll look at *a read I/O function handler that returns directory entries*.

The basic skeleton of a resource manager

The following can be used as a template for a resource manager with multiple threads. (We've already seen a template that can be used for a single-threaded resource manager above in "The resource manager library," when we discussed a **/dev/null** resource manager).

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

static resmgr_connect_funcs_t   connect_func;
static resmgr_io_funcs_t        io_func;
static iofunc_attr_t            attr;

int main (int argc, char **argv)
{
    thread_pool_attr_t    pool_attr;
    thread_pool_t         *tpp;
    dispatch_t            *dpp;
    resmgr_attr_t         resmgr_attr;
    int                   id;

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

    memset (&pool_attr, 0, sizeof (pool_attr));
```


© 2020, QNX Software Systems Limited

```
    pool_attr.handle = dpp;
    pool_attr.context_alloc = (void *) dispatch_context_alloc;
    pool_attr.block_func    = (void *) dispatch_block;
    pool_attr.handler_func  = (void *) dispatch_handler;
    pool_attr.context_free  = (void *) dispatch_context_free;

    // 1) set up the number of threads that you want
    pool_attr.lo_water = 2;
    pool_attr.hi_water = 4;
    pool_attr.increment = 1;
    pool_attr.maximum = 50;

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

    iofunc_func_init (_RESMGR_CONNECT_NFUNCS, &connect_func,
                      _RESMGR_IO_NFUNCS, &io_func);
    iofunc_attr_init (&attr, S_IFNAM | 0777, 0, 0);

    // 2) override functions in "connect_func" and "io_func" as
    // required here

    memset (&resmgr_attr, 0, sizeof (resmgr_attr));
    resmgr_attr.nparts_max = 1;
    resmgr_attr.msg_max_size = 2048;

    // 3) replace "/dev/whatever" with your device name
    if ((id = resmgr_attach (dpp, &resmgr_attr, "/dev/whatever",
                _FTYPE_ANY, 0, &connect_func, &io_func,
                &attr)) == -1) {
        fprintf (stderr,
                 "%s:  Unable to attach name.\n", argv [0]);
        return (EXIT_FAILURE);
    }

    // Never returns
    thread_pool_start (tpp);

    return (EXIT_SUCCESS);
}
```

For more information about the dispatch interface (i.e., the *dispatch_create()* function), see the documentation in the QNX Neutrino *C Library Reference*.

Step 1

Here you'd use the thread pool functions to create a pool of threads that will be able to service messages in your resource manager. Generally, I recommend that you start off with a single-threaded resource manager, as we did with the **/dev/null** example mentioned above.

Once you have the basic functionality running, you can *then* add threads. You'd modify the *lo_water*, *hi_water*, *increment*, and *maximum* members of the `pool_attr` structure as described in the *Threads & Processes* chapter where we discuss the thread pool functions.

Step 2

Here you'd add whatever functions *you* want to supply. These are the outcalls we just discussed (e.g., read I/O function handler, device control I/O function handler, etc.) For example, to add your own handler for the _IO_READ message that points to a function supplied by you called *my_io_read()*, you'd add the following line of code:

```
io_func.io_read = my_io_read;
```

This overrides the POSIX-layer default function that got put into the table by *iofunc_func_init()* with a pointer to your function, *my_io_read()*.

Step 3

You probably don't want your resource manager to be called **/dev/whatever**, so you should select an appropriate name. Note that the *resmgr_attach()* function is where you bind the attributes structure (the *attr* parameter) to the name—if you wish to have multiple devices handled by your resource manager, you'd call *resmgr_attach()* multiple times, with different attributes structures (so that you could tell the different registered names apart at runtime).

A simple read I/O function handler example

To illustrate how your resource manager might return data to a client, consider a simple resource manager that always returns the constant string `"Hello, world!\n"`. There are a number of issues involved, even in this very simple case:

Matching of the client's data area size to the data being returned

In our case, the resource manager is returning a fixed string of 14 bytes—there is exactly that much data available. This is identical to a read-only file on a disk that contains the string in question; the only real difference is that this “file” is maintained in our C program via the statement:

```
char    *data_string = "Hello, world!\n";
```

The client, on the other hand, can issue a *read()* request of any size—the client could ask for one byte, 14 bytes, or more. The impact of this on the read I/O function handler that you're going to provide is that you must be able to match the client's requested data size with what's available.

Handling of the EOF case

A natural fallout of the way you handle the client's data area size considerations is the corner case of dealing with the End-Of-File (EOF) on the fixed string. Once the client has read the final `\n` character, further attempts by the client to read more data should return EOF.

© 2020, QNX Software Systems Limited

Maintenance of context information (the *lseek()* index)

Both the “Data area size considerations” and the “Handling of EOF case” scenarios will require that context be maintained in the OCB passed to your read I/O function handler, specifically the *offset* member.

Updating of POSIX *stat()* information

One final consideration: when data is read from a resource manager, the POSIX *access time* (*atime*) variable needs to be updated. This is so that a client *stat()* function will show that someone has indeed accessed the device.

The code

Here's the code that addresses all the above points. We'll go through it step-by-step in the discussion that follows:

```
/*
 * io_read1.c
*/

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/neutrino.h>
#include <sys/iofunc.h>

// our data string
char    *data_string = "Hello, world!\n";

int
io_read (resmgr_context_t *ctp, io_read_t *msg, iofunc_ocb_t *ocb)
{
    int     sts;
    int     nbytes;
    int     nleft;
    int     off;
    int     xtype;
    struct _xtype_offset *xoffset;

    // 1) verify that the device is opened for read
    if ((sts = iofunc_read_verify (ctp, msg, ocb, NULL)) != EOK) {
        return (sts);
    }

    // 2) check for and handle an XTYPE override
    xtype = msg -> i.xtype & _IO_XTYPE_MASK;
    if (xtype == _IO_XTYPE_OFFSET) {
        xoffset = (struct _xtype_offset *) (&msg -> i + 1);
        off = xoffset -> offset;
    } else if (xtype == _IO_XTYPE_NONE) {
        off = ocb -> offset;
    } else {   // unknown, fail it
```

```
        return (ENOSYS);
    }

    // 3) how many bytes are left?
    nleft = ocb -> attr -> nbytes - off;

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

    // 5) if returning data, write it to client
    if (nbytes) {
        MsgReply (ctp -> rcvid, nbytes, data_string + off, nbytes);

        // 6) set up POSIX stat() "atime" data
        ocb -> attr -> flags |= IOFUNC_ATTR_ATIME |
                                IOFUNC_ATTR_DIRTY_TIME;

        // 7) advance the lseek() index by the number of bytes
        // read if not _IO_XTYPE_OFFSET
        if (xtype == _IO_XTYPE_NONE) {
            ocb -> offset += nbytes;
        }
    } else {
        // 8) not returning data, just unblock client
        MsgReply (ctp -> rcvid, EOK, NULL, 0);
    }

    // 9) indicate we already did the MsgReply to the library
    return (_RESMGR_NOREPLY);
}
```

Step 1

Here we ensured that the client's *open()* call had in fact specified that the device was to be opened for reading. If the client opened the device for writing only, and then attempted to perform a read from it, it would be considered an error. In that case, the helper function *iofunc_read_verify()* would return EBADF, and not EOK, so we'd return that value to the library, which would then pass it along to the client.

Step 2

Here we checked to see if the client had specified an *xtype-override*—a per-message override (e.g., because while the device had been opened in non-blocking mode, this specifies for this one request that we'd like blocking behavior). Note that the blocking aspect of the “xtype” override can be noted by the *iofunc_read_verify()* function's last parameter—since we're illustrating a very simple example, we just passed in a NULL indicating that we don't care about this aspect.

More important, however, is to see how particular “xtype” modifiers are handled. An interesting one is the _IO_XTYPE_OFFSET modifier, which, if present, indicates that the message passed from the client contains an offset and that the read operation should *not* modify the “current file position” of the file descriptor (this is used by the function *pread()*, for example).

© 2020, QNX Software Systems Limited

If the _IO_XTYPE_OFFSET modifier is not present, then the read operation can go ahead and modify the "current file position." We use the variable *xtype* to store the "xtype" that we received in the message, and the variable *off* to represent the current offset that we should be using during processing. You'll see some additional handling of the _IO_XTYPE_OFFSET modifier below, in step 7.

If there is a different "xtype override" than _IO_XTYPE_OFFSET (and not the no-op one of _IO_XTYPE_NONE), we fail the request with ENOSYS. This simply means that we don't know how to handle it, and we therefore return the error up to the client.

Steps 3 & 4

To calculate how many bytes we can actually return to the client, we perform steps 3 and 4, which figure out how many bytes are available on the device (by taking the total device size from *ocb->attr->nbytes* and subtracting the current offset into the device). Once we know how many bytes are left, we take the smaller of that number and the number of bytes that the client specified that they wish to read. For example, we may have seven bytes left, and the client wants to read only two. In that case, we can return only two bytes to the client. Alternatively, if the client wanted 4096 bytes, but we had only seven left, we could return only seven bytes.

Step 5

Now that we've calculated how many bytes we're going to return to the client, we need to do different things based on whether or not we're returning data. If we are returning data, then after the check in step 5, we reply to the client with the data. Notice that we use `data_string + off` to return data starting at the correct offset (the *off* is calculated based on the xtype override).

Also notice the second parameter to *MsgReply()*—it's documented as the *status* argument, but in this case we're using it to return the number of bytes. This is because the implementation of the client's *read()* function knows that the return value from its *MsgSendv()* (which is the *status* argument to *MsgReply()*, by the way) is the number of bytes that were read. This is a common convention.

Step 6

Since we're returning data from the device, we know that the device has been accessed. We set the IOFUNC_ATTR_ATIME and IOFUNC_ATTR_DIRTY_TIME bits in the *flags* member of the attribute structure. This serves as a reminder to the stat I/O function handler that the access time is not valid and should be fetched from the system clock before replying. If we really wanted to, we could have stuffed the current time into the *atime* member of the attributes structure and cleared the IOFUNC_ATTR_DIRTY_TIME flag. But this isn't very efficient, since we're expecting to get a lot more *read()* requests from the client than *stat()* requests. However, your usage patterns may dictate otherwise.

So which time does the client see when it finally *does* call *stat()*? The *iofunc_stat_default()* function provided by the resource manager library will look at the *flags* member of the attribute structure to see if the times are valid (the *atime*, *ctime*, and *mtime* fields). If they are not (as will be the case after our read I/O function handler has been called that returned data), the *iofunc_stat_default()*

function will update the time(s) with the current time. The real value of the time is also updated on a *close()*, as you'd expect.

Step 7

Now we advance the *lseek()* offset by the number of bytes that we returned to the client, only if we are *not* processing the _IO_XTYPE_OFFSET override modifier. This ensures that, in the non-_IO_XTYPE_OFFSET case, if the client calls *lseek()* to get the current position, or (more importantly) when the client calls *read()* to get the next few bytes, the offset into the resource is set to the correct value. In the case of the _IO_XTYPE_OFFSET override, we leave the *ocb* version of the offset alone.

Step 8

Contrast step 6 with this step. Here we only unblock the client, we don't perform any other functions. Notice also that there is no data area specified to the *MsgReply()*, because we're not returning data.

Step 9

Finally, in step 9, we perform processing that's common regardless of whether or not we returned data to the client. Since we've already unblocked the client via the *MsgReply()*, we certainly don't want the resource manager library doing that for us, so we tell it that we've already done that by returning _RESMGR_NOREPLY.

Effective use of other messaging functions

As you'll recall from the Message Passing chapter, we discussed a few other message-passing functions—namely *MsgWrite()*, *MsgWritev()*, and *MsgReplyv()*. The reason I'm mentioning them here again is because your read I/O function handler may be in an excellent position to use these functions. In the simple example shown above, we were returning a contiguous array of bytes from one memory location. In the real world, you may need to return multiple pieces of data from various buffers that you've allocated.

A classical example of this is a ring buffer, as might be found in a serial device driver. Part of the data may be near the end of the buffer, with the rest of it “wrapped” to the top of the buffer. In this case, you'll want to use a two-part IOV with *MsgReplyv()* to return both parts. The first part of the IOV would contain the address (and length) of the bottom part of the data, and the second part of the IOV would contain the address (and length) of the top part of the data. Or, if the data is going to arrive in pieces, you may instead choose to use *MsgWrite()* or *MsgWritev()* to place the data into the client's address space as it arrives and then specify a final *MsgReply()* or *MsgReplyv()* to unblock the client. As we've seen above, there's no requirement to actually *transfer* data with the *MsgReply()* function—you can use it to simply unblock the client.

A simple write I/O function handler example

The read I/O function handler example was fairly simple; let's take a look at the write I/O function handler. The major hurdle to overcome with the write I/O function handler is to access the data. Since the resource manager library reads in a small portion of the message from the client, the data content that the client sent (immediately after the _IO_WRITE header) may have only partially arrived at the

 © 2020, QNX Software Systems Limited

write I/O function handler. For example, for a client writing one megabyte, only the header and up to 1500 bytes of the data get read by the resource manager library.

You use *resmgr_msgread()* to get any data that exceeds the 1500 or so bytes that are available from the message buffer. However, you can simplify the process and reduce the likelihood of bugs by ignoring the data in the message buffer and using *resmgr_msgread()* to get all the data. Making use of the data from the message buffer might improve performance, but you are unlikely to notice any improvement unless your resource manager deals with a very large number of write operations that contain a small amount of data. The best way to improve performance is for clients to perform larger writes.

One further wrinkle introduced in the write I/O function handler example is the handling of the _IO_XTYPE_OFFSET modifier (and associated data; it's done slightly differently than in the read I/O function handler example).

Here's the code:

```
/*
 * io_write1.c
*/

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/neutrino.h>
#include <sys/iofunc.h>

void
process_data (int offet, void *buffer, int nbytes)
{
    // do something with the data
}

int
io_write (resmgr_context_t *ctp, io_write_t *msg,
          iofunc_ocb_t *ocb)
{
    int     sts;
    size_t  nbytes;
    size_t  off;
    size_t  start_data_offset;
    int     xtype;
    char    *buffer;
    struct _xtype_offset *xoffset;

    // verify that the device is opened for write
    if ((sts = iofunc_write_verify (ctp, msg, ocb, NULL)) != EOK)
    {
        return (sts);
    }

    // 1) check for and handle an XTYPE override
    xtype = msg -> i.xtype & _IO_XTYPE_MASK;
    if (xtype == _IO_XTYPE_OFFSET) {
        xoffset = (struct _xtype_offset *) (&msg -> i + 1);
```

```
        start_data_offset = sizeof (msg -> i) + sizeof (*xoffset);
        off = xoffset -> offset;
    } else if (xtype == _IO_XTYPE_NONE) {
        off = ocb -> offset;
        start_data_offset = sizeof (msg -> i);
    } else {   // unknown, fail it
        return (ENOSYS);
    }

    // 2) allocate a buffer big enough for the data
    nbytes = _IO_WRITE_GET_NBYTES(msg);
    if ((buffer = malloc (nbytes)) == NULL) {
        return (ENOMEM);
    }

    // 3) (re-)read the data from the client
    if (resmgr_msgread (ctp, buffer, nbytes,
                        start_data_offset) == -1)
    {
        free (buffer);
        return (errno);
    }

    // 4) do something with the data
    process_data (off, buffer, nbytes);

    // 5) free the buffer
    free (buffer);

    // 6) set up the number of bytes for the client's "write"
    // function to return
    _IO_SET_WRITE_NBYTES (ctp, nbytes);

    // 7) if any data written, update POSIX structures and OCB offset
    if (nbytes) {
        ocb -> attr -> flags |= IOFUNC_ATTR_MTIME | IOFUNC_ATTR_DIRTY_TIME;
        if (xtype == _IO_XTYPE_NONE) {
            ocb -> offset += nbytes;
        }
    }

    // 8) tell the resource manager library to do the reply, and that it
    // was okay
    return (EOK);
}
```

As you can see, a few of the initial operations performed were identical to those done in the read I/O function handler example—the *iofunc_write_verify()* is analogous to the *iofunc_read_verify()* function, and the xtype override check is the same.

Step 1

Here we performed much the same processing for the “xtype override” as we did in the read I/O function handler example, except for the fact that the offset is *not* stored as part of the

© 2020, QNX Software Systems Limited

incoming message structure. The reason it's not stored there is because a common practice is to use the size of the incoming message structure to determine the starting point of the actual data being transferred from the client. We take special pains to ensure the offset of the start of the data (*doffset*) is correct in the xtype handling code.

Step 2

Here we allocate a buffer that's big enough for the data. The number of bytes that the client is writing is presented to us in the *nbytes* member of the *msg* union. This is stuffed automatically by the client's C library in the *write()* routine. Note that if we don't have sufficient memory to handle the *malloc()* request, we return the error number ENOMEM to the client—effectively, we're passing on the return code to the client to let it know why its request wasn't completed.

Step 3

Here we use the helper function *resmgr_msgread()* to read the entire data content from the client directly into the newly allocated buffer. In most cases we could have just used *MsgRead()*, but in the case where this message is part of a "combine message," *resmgr_msgread()* performs the appropriate "magic" for us (see the "Combine message" section for more information on *why* we need to do this.)

The parameters to *resmgr_msgread()* are fairly straightforward; we give it the internal context pointer (*ctp*), the buffer into which we want the data placed (*buffer*), and the number of bytes that we wish read (the *nbytes* member of the message *msg* union). The last parameter is the offset into the current message, which we calculated above, in step 1. The offset effectively skips the header information that the client's C library implementation of *write()* put there, and proceeds directly to the data. This actually brings about two interesting points:

- We could use an arbitrary offset value to read chunks of the client's data in any order and size we want.
- We could use *resmgr_msgreadv()* (note the "v") to read data from the client into an IOV, perhaps describing various buffers, similar to what we did with the cache buffers in the filesystem discussion in the Message Passing chapter.

Step 4

Here you'd do whatever you want with the data—I've just called a made-up function called *process_data()* and passed it the buffer and size.

Step 5

This step is crucial! Forgetting to do it is easy, and will lead to "memory leaks." Notice how we also took care to free the memory in the case of a failure in step 3.

Step 6

We're using the macro *_IO_SET_WRITE_NBYTES()* (see the entry for *iofunc_write_verify()* in the QNX Neutrino *C Library Reference*) to store the number of bytes we've written, which will then be passed back to the client as the return value from the client's *write()*. It's important to note that you should return the *actual* number of bytes! The client is depending on this.

Step 7

Now we do similar housekeeping for *stat()*, *lseek()*, and further *write()* functions as we did for the read I/O function handler routine (and again, we modify the offset in the *ocb* only in the case of this *not* being a _IO_XTYPE_OFFSET type of message). Since we're writing to the device, however, we use the IOFUNC_ATTR_MTIME constant instead of the IOFUNC_ATTR_ATIME constant. The MTIME flag means “modification” time, and a *write()* to a resource certainly “modifies” it.

Step 8

The last step is simple: we return the constant EOK, which tells the resource manager library that it should reply to the client. This ends our processing. The resource manager will use the number of bytes that we stashed away with the *_IO_SET_WRITE_NBYTES()* macro in the reply and the client will unblock; the client's C library *write()* function will return the number of bytes that were written by our device.

A simple device control I/O function handler example

The client's *devctl()* call is formally defined as:

```
#include <sys/types.h>
#include <unistd.h>
#include <devctl.h>

int
devctl (int fd,
        int dcmd,
        void *dev_data_ptr,
        size_t nbytes,
        int *dev_info_ptr);
```

We should first understand this function before we look at the resource manager side of things. The *devctl()* function is used for “out of band” or “control” operations. For example, you may be writing data to a sound card (the actual digital audio samples that the sound card should convert to analog audio), and you may decide that you need to change the number of channels from 1 (mono) to 2 (stereo), or the sampling rate from the CD-standard (44.1 kHz) to the DAT-standard (48 kHz). You may also find that you need to change the sample rate for recording sound from 22kHz to 44.1kHz depending on the application. The *devctl()* function is the appropriate way to do this. When you write a resource manager, you may find that you don't need any *devctl()* support at all and that you can perform all the functionality needed simply through the standard *read()* and *write()* functions. You may, on the other hand, find that you need to mix *devctl()* calls with the *read()* and *write()* calls, or indeed that your device uses only *devctl()* functions and does *not* use *read()* or *write()*.

The *devctl()* function takes these arguments:

fd

The file descriptor of the resource manager that you're sending the *devctl()* to.

 © 2020, QNX Software Systems Limited

dcmd

The command itself—a combination of two bits worth of direction, and 30 bits worth of command (see discussion below).

dev_data_ptr

A pointer to a data area that can be sent to, received from, or both.

nbytes

The size of the *dev_data_ptr* data area.

dev_info_ptr

An extra information variable that can be set by the resource manager.

The top two bits in the *dcmd* encode the direction of data transfer, if any. For details, see the description in the I/O reference section (under "*Device control I/O function handler*").

When the _IO_DEVCTL message is received by the resource manager, it's handled by your device control I/O function handler. Here is a very simple example, which we'll assume is used to set the number of channels and the sampling rate for the audio device we discussed above:

```
/*
 * io_devctl1.c
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <sys/neutrino.h>
#include <sys/iofunc.h>
#include <devctl.h>

#define AUDIO_DAC 1
#define AUDIO_ADC 2

extern void audio_set_nchannels(unsigned num);
extern uint32_t audio_get_samplerate(unsigned port);
extern void audio_set_samplerate(unsigned port, unsigned rate);

#define DCMD_AUDIO_SET_CHANNEL_MONO         __DION(_DCMD_MIXER, 1)
#define DCMD_AUDIO_SET_CHANNEL_STEREO       __DION(_DCMD_MIXER, 2)
#define DCMD_AUDIO_GET_SAMPLE_RATE_DAC      __DIOF(_DCMD_MIXER, 4, uint32_t)
#define DCMD_AUDIO_SET_SAMPLE_RATE_DAC      __DIOT(_DCMD_MIXER, 5, uint32_t)
#define DCMD_AUDIO_GET_SAMPLE_RATE_ADC      __DIOF(_DCMD_MIXER, 6, uint32_t)
#define DCMD_AUDIO_SET_SAMPLE_RATE_ADC      __DIOT(_DCMD_MIXER, 7, uint32_t)

int
io_devctl (resmgr_context_t *ctp, io_devctl_t *msg,
           iofunc_ocb_t *ocb)
{
```

```
int     sts;

/* 1) Create the reply buffer at the start of ctp->msg so we make
 * sure we have enough space */
struct _io_devctl_reply *reply = (struct _io_devctl_reply *)ctp->msg;

/* Create a pointer to the rate variable for the _GET_ devctls. */
unsigned nbytes = 0;
uint32_t *rate = _DEVCTL_DATA(*reply);

/* 2) Verify we have the entire devctl header in the buffer */
if( ctp->size < sizeof(msg->i) ) {
    return EBADMSG;
}

/* 3) See if it's a standard devctl() */
if ((sts = iofunc_devctl_default (ctp, msg, ocb)) !=
    _RESMGR_DEFAULT)
{
    return (sts);
}

/* How many bytes did the client send in addition to the devctl header? */
size_t payload_size = ctp->size - sizeof(msg->i);

/* 4) See which command it was, and act on it */
switch (msg -> i.dcmd) {
case DCMD_AUDIO_SET_CHANNEL_MONO:
    /* Read or write access are sufficient */
    if( !(ocb->ioflag & (_IO_FLAG_WR | _IO_FLAG_RD)) ) {
        return EPERM;
    }
    audio_set_nchannels (1);
    break;

case DCMD_AUDIO_SET_CHANNEL_STEREO:
    /* Read or write access are sufficient */
    if( !(ocb->ioflag & (_IO_FLAG_WR | _IO_FLAG_RD)) ) {
        return EPERM;
    }
    audio_set_nchannels (2);
    break;

case DCMD_AUDIO_GET_SAMPLE_RATE_DAC:
    /* Write access is required for accessing the DAC */
    if( !(ocb->ioflag & _IO_FLAG_WR) ) {
        return EPERM;
    }

    /* Verify that the client wants enough bytes */
    if( ctp->info.dstmsglen < (sizeof(*reply) + sizeof(uint32_t)) ) {
        return EINVAL;
    }
```


© 2020, QNX Software Systems Limited

```
        /* Set up the reply to the client */
        nbytes = sizeof(uint32_t);
        *rate = audio_get_samplerate (AUDIO_DAC);
        break;

    case DCMD_AUDIO_SET_SAMPLE_RATE_DAC:
        /* Write access is required for accessing the DAC */
        if( !(ocb->ioflag & _IO_FLAG_WR) ) {
            return EPERM;
        }

        /* Verify that we got all the bytes. */
        if( payload_size < sizeof(uint32_t) ) {
            return EBADMSG;
        }

        rate = _DEVCTL_DATA(msg->i);
        audio_set_samplerate (AUDIO_DAC, *rate);
        break;

    case DCMD_AUDIO_GET_SAMPLE_RATE_ADC:
        /* Read access is required for accessing the ADC */
        if( !(ocb->ioflag & _IO_FLAG_RD) ) {
            return EPERM;
        }

        /* Verify that the client wants enough bytes */
        if( ctp->info.dstmsglen < (sizeof(*reply) + sizeof(uint32_t)) ) {
            return EINVAL;
        }

        /* Set up the reply to the client */
        nbytes = sizeof(uint32_t);
        *rate = audio_get_samplerate (AUDIO_ADC);
        break;

    case DCMD_AUDIO_SET_SAMPLE_RATE_ADC:
        /* Read access is required for accessing the ADC */
        if( !(ocb->ioflag & _IO_FLAG_RD) ) {
            return EPERM;
        }

        /* Verify that we got all the bytes. */
        if( payload_size < sizeof(uint32_t) ) {
            return EBADMSG;
        }

        rate = _DEVCTL_DATA(msg->i);
        audio_set_samplerate (AUDIO_ADC, *rate);
        break;

    /* 5) In case it's a command that we don't recognize, fail it */
    default:
        return (ENOSYS);
```

```
    }

    /* 6) Tell the client that it worked */
    memset(reply, 0, sizeof(*reply));
    reply->nbytes = nbytes;
    SETIOV (ctp->iov, reply, sizeof(*reply) + nbytes);
    return (_RESMGR_NPARTS (1));
}
```

Step 1

We set up where the reply is going to be written to. The client is expecting the response in an `_io_devctl_reply` structure, with (depending on the value of *dcmd*) the sample rate immediately following the structure in memory. This response is going to take *sizeof(msg->o)* + *sizeof(uint32_t)* bytes. The buffer that we have and can use for sending replies is *ctp->msg* and it is *ctp->msg_max_size* bytes long. Because it's the same buffer that we used for receiving the commands from the client, we need to be careful to finish processing the input before we start writing the output to the buffer. The *devctl()* message from the client is sitting at offset *ctp->offset* into the buffer *ctp->msg*. If we have enough space between *ctp->offset* and *ctp->msg_max_size* for the entire reply, we could use that. However, it's much safer to just write our reply at the beginning of the buffer, which gives us the full *ctp->msg_max_size* bytes of data to write our response.

Step 2

A basic sanity check to make sure the entire *devctl()* message is available. We do not want the client to send only a partial message.

Step 3

We see again the use of a helper function, this time *iofunc_devctl_default()*, which is used to perform all default processing for *devctl()*. The *iofunc_devctl_default()* function is called when you don't supply your own device control I/O function handler and let *iofunc_func_init()* initialize the I/O and connect functions tables for you.

We include *iofunc_devctl_default()* in our device control I/O function handler function because we want it to handle all the regular *devctl()* cases for us. We examine the return value; if it's not _RESMGR_DEFAULT, then this means that the *iofunc_devctl_default()* function “handled” the request, so we just pass along its return value as our return value.

If the constant _RESMGR_DEFAULT *is* the return value, then we know that the helper function *didn't* handle the request and that we should check to see if it's one of ours.

Step 4

The checking for our custom *devctl()* commands is done here. We simply check the *dcmd* values that the client code sent to see if there's a match. Note that we call the fictitious functions *audio_set_nchannels()*, *audio_get_samplerate()* and *audio_set_samplerate()* to accomplish the actual “work” for the client.

As part of our processing of the various audio *devctl()* commands, we need to check permissions. If the client is getting or setting the sample rate of the digital to analog converter (DAC), which is output, then they need to have opened the resource manager handle for writing. If the client is setting the sample rate of the analog to digital converter (ADC), which is input, then they need to have opened the resource manager handle for reading. Note here

© 2020, QNX Software Systems Limited

that the permissions being checked are related to whether the DAC or ADC is being accessed, not based on whether *devctl()* is sending or receiving data from the client (i.e., the use of *__DIOF* and *__DIOT* does not correlate with read and write permissions).

In *devctl()* calls that set sample rate, we also need to make sure that the client sent enough data. The length checks at the beginning of the *devctl()* function are not sufficient because each *devctl()* might be expecting the client to send a different amount of data. We do the full check for data now that we know exactly how much data to expect. Once we've done this check, we can safely access this data and set the sample rate.

In *devctl()* calls that get sample rate, we make sure the client has enough space to receive the response from the resource manager.

Step 5

This step is simply good defensive programming. We return an error code of ENOSYS to tell the client that we didn't understand their request.

Step 6

If we get to this line, we successfully handled the *dcmd* passed to *devctl()*. We need to respond to the client. We return a value to the resource manager library encoded by the macro *_RESMGR_NPARTS()*, telling it that we're returning a one-part IOV. This value is then returned to the client.

Instead of using the macro *_RESMGR_NPARTS()*, we could have used the *_RESMGR_PTR()* macro:

```
/* 6) tell the client it worked */
memset(reply, 0, sizeof(*reply));
reply->nbytes = nbytes;
return (_RESMGR_PTR (ctp, reply, sizeof(*reply) + nbytes));
```

The *devctl()* command requires us to send a response to the client for *devctl()* to be successful. It is insufficient to just return EOK without any response message.

Advanced topics

Now that we've covered the “basics” of resource managers, it's time to look at some more complicated aspects.

Extending the OCB

To extend the OCB, you'll need to provide two functions—one to allocate (and initialize) the new OCB and one to free it—to override the defaults, *iofunc_ocb_calloc()* and *iofunc_ocb_free()*. Then, you'll need to bind your two customized functions into the mount structure. (Yes, this does mean that you'll need a mount structure, if only for this one purpose.) Finally, you'll need to define your own OCB typedef, so that the prototypes for the code are all correct.

Let's look at the OCB typedef first, and then we'll see how to override the functions:

```
#define IOFUNC_OCB_T struct my_ocb
#include <sys/iofunc.h>
```

This tells the included file, **<sys/iofunc.h>**, that the manifest constant IOFUNC_OCB_T now points to your new and improved OCB structure.

It's *very* important to keep in mind that the “normal” OCB *must* appear as the first entry in your extended OCB! This is because the POSIX helper library passes around a pointer to what it expects is a normal OCB—it doesn't know about your extended OCB, so therefore the first data element at the pointer location must be the normal OCB.

Here's our extended OCB:

```
typedef struct my_ocb
{
    iofunc_ocb_t     normal_ocb;
    int              my_extra_flags;
    ...
} my_ocb_t;
```

Finally, here's the code that illustrates how to override the allocation and deallocation functions in the mount structure:

```
// declare
iofunc_mount_t      mount;
iofunc_funcs_t      mount_funcs;

// set up the mount functions structure
// with our allocate/deallocate functions

// _IOFUNC_NFUNCS is from the .h file
mount_funcs.nfuncs = _IOFUNC_NFUNCS;
```


© 2020, QNX Software Systems Limited

```
// your new OCB allocator
mount_funcs.ocb_calloc = my_ocb_calloc;

// your new OCB deallocator
mount_funcs.ocb_free = my_ocb_free;

// set up the mount structure
memset (&mount, 0, sizeof (mount));
```

Then all you have to do is bind the mount functions to the mount structure, and the mount structure to the attributes structure:

```
...

mount.funcs = &mount_funcs;
attr.mount = &mount;
```

The *my_ocb_calloc()* and *my_ocb_free()* functions are responsible for allocating and initializing an extended OCB and for freeing the OCB, respectively. They are prototyped as:

```
IOFUNC_OCB_T *
my_ocb_calloc (resmgr_context_t *ctp, IOFUNC_ATTR_T *attr);

void
my_ocb_free (IOFUNC_OCB_T *ocb);
```

This means that the *my_ocb_calloc()* function gets passed both the internal resource manager context and the attributes structure. The function is responsible for returning an initialized OCB. The *my_ocb_free()* function gets passed the OCB and is responsible for releasing the storage for it.

It's important to realize that the OCB may be allocated by functions other than the normal open connect function handler—for example, the memory manager may allocate an OCB. The impact of this is that your OCB allocating function must be able to initialize the OCB with the *attr* argument.

There are two interesting uses for these two functions (that have nothing to do with extending the OCB):

Monitoring the allocation and deallocation of OCBs

In this case, you can simply “tie in” to the allocator/deallocator and monitor the usage of the OCBs (for example, you may wish to limit the total number of OCBs outstanding at any given time). This may prove to be a good idea if you're not taking over the open connection function handler, and yet still need to intercept the creation of (and possibly deletion of) OCBs.

Providing more efficient allocation and deallocation

Another reason for overriding the library's built-in OCB allocator/deallocator is that you may wish to keep the OCBs on a free list, instead of using the library's *calloc()* and *free()* functions. If you're allocating and deallocating OCBs at a high rate, this may prove to be more efficient.

Locking in the resource manager

The resource manager framework provides facilities for safe thread synchronization in your resource manager. Before you write a multi-threaded resource manager, make sure you are familiar with the default behavior of the shared data structures that thread synchronization uses and the requirements for safely overriding them.

Shared data structures for multi-threaded resource managers

The OCB contains the open mode of the file descriptor (RDONLY, WRONLY, RDWR), its current offset, and a count of how many times it has been duplicated (using *dup()*). Some of these items are static, but the offset changes frequently. Successive operations on the same file descriptor have a dependency on the offset set by the previous operation.

The attributes structure represents the file itself. It stores information that changes rapidly such as the file credentials, the access permissions, size, and timestamps.

Because POSIX requires that simultaneous reads and writes to a file don't result in half-updates, the attribute structure needs to be locked for all operations. The resource manager framework meets this requirement by effectively having a single mutex for each attribute structure. By default, this mutex is manipulated by *io_func_attr_lock()* and *io_func_attr_unlock()*. The resource manager framework acquires it on the relevant attribute structure before it invokes any I/O handlers and releases it once the handler returns.

Because this mechanism finds the attribute structure to lock via the OCB associated with the file descriptor, it only works for file descriptor-based operations. Connect operations are pathname-based and do not have a file descriptor or OCB. Instead, the attribute structure of the mountpoint (the path given to *resmgr_attach()*) is the *handle* parameter of *resmgr_attach()*. In filesystem resource managers (i.e., the _RESMGR_FLAG_DIR flag is passed to *resmgr_attach()*), the framework can provide the attribute structure for the mountpoint, but does not know how to locate the correct attribute structure for the file or directory being processed. Therefore, it is the responsibility of your own connect handlers to determine and acquire the attribute structure associated with the named file.

Do not simply allocate a fresh attribute structure on every *open()*. For both locking and memory-mapped files to function correctly, every *open()* called on a file requires a call to *iofunc_ocb_attach()* against the same attribute structure.

Overriding the locking behavior

Before it invokes your I/O handler function, the resource manager framework invokes the functions pointed to by the `resmgr_io_funcs` members *lock_ocb* and *unlock_ocb* to lock and unlock the OCB structure (see *iofunc_func_init()* and *resmgr_attach()*). By default, *iofunc_lock_ocb_default()* and *iofunc_unlock_ocb_default()* are used, which simply lock or unlock the attribute structure associated with the OCB.

 © 2020, QNX Software Systems Limited

The attribute structure is locked and unlocked via the functions that the `iofunc_mount_t` member *funcs* points to (`_iofunc_funcs` members *attr_lock*, *attr_unlock*, and *attr_trylock*). When these function pointers are not populated (the default behavior), the OCB locking functions call *iofunc_attr_lock()* or *iofunc_attr_unlock()*, which means that the resource manager framework expects that calls to *lock_ocb()* and *unlock_ocb()* also lock or unlock the related attribute structure. This default behavior is "one-level" locking—attribute locks are used to lock OCBs—and should be sufficient for most resource managers.

However, if required, you can implement "two-level" locking, which allows your resource manager handler to release the lock on a shared attribute but preserve the OCB lock. This behavior is useful if your resource manager supports independent operations on the same named entry.

For both "one-level" and "two-level" locking:

- You can either use the default *iofunc_attr_lock()* and *iofunc_attr_unlock()* for the attribute lock or implement your own and populate *attr_lock* and *attr_unlock*.
- Make sure locking is recursive (see *pthread_mutexattr_setrecursive()*).

For "one-level" locks, *iofunc_lock_ocb_default()* and *iofunc_unlock_ocb_default()* use your *attr_lock* and *attr_unlock* if you populate them. You don't need to create your own functions for *lock_ocb* and *unlock_ocb* in `resmgr_io_funcs`.

Creating "two-level" locks requires you to create your own functions for *lock_ocb* and *unlock_ocb*. To allow your code to work well with the default resource manager handlers, make sure that these functions use the same attribute structure lock functions as the ones that `iofunc_mount_t` references.

For "two-level" locks, make sure your code also includes the following behavior:

- Calls to *lock_ocb()* also lock the attribute.
- The OCB is locked first and the attribute structure is locked second, which allows you to unlock `iofunc_attr_t` as needed while retaining the OCB lock.

For both default and custom locking (using one or two levels), it's safe for a handler to temporarily unlock the attribute structure (or the OCB, or both the attribute and OCB, in the case of "two-level" locks), as long as it locks the affected items before it returns. If you use custom lock functions, make sure they cannot fail.

Having handlers that temporarily unlock structures means that other threads are allowed to run handlers. In particular, if you unlock the OCB, another thread could close it. After you lock the OCB again, the OCB is still valid (it is not free until after the handler returns) but you might need to be careful about anything in the OCB that your *close_ocb* handler might have invalidated.

Extending the attributes structure

You may wish to extend the attributes structure in cases where you need to store additional device information. Since the attributes structure is associated on a "per-device" basis, this means that any extra information you store there will be accessible to all OCBs that reference that device (since the OCB contains a pointer to the attributes structure). Often things like serial baud rate, etc. are stored in extended attributes structures.

Extending the attributes structure is much simpler than dealing with extended OCBs, simply because attributes structures are allocated and deallocated by your code anyway.

You have to perform the same "trick" of overriding the header files with the "new" attributes structure as we did with the extended OCB above:

```
#define IOFUNC_ATTR_T struct my_attr
#include <sys/iofunc.h>
```

Next you actually define the contents of your extended attribute structures. Note that the extended attribute structure *must* have the "normal" attribute structure encapsulated as the *very first* element, just as we did with the extended OCB (and for the same reasons).

Blocking within the resource manager

So far we've avoided talking about blocking within the resource manager. We assume that you will supply an outcall function (e.g., a read I/O function handler), and that the data will be available immediately. What if you need to block, waiting for the data? For example, performing a *read()* on the serial port might need to block until a character arrives. Obviously, we can't predict how long this will take.

Blocking within a resource manager is based on the same principles that we discussed in the Message Passing chapter—after all, a resource manager is really a server that handles certain, well-defined messages. When the message corresponding to the client's *read()* request arrives, it does so with a receive ID, and the client is blocked. If the resource manager has the data available, it will simply return the data as we've already seen in the various examples above. However, if the data isn't available, the resource manager will need to keep the client blocked (if the client has indeed specified blocking behavior for the operation) to continue processing other messages. What this really means is that the thread (in the resource manager) that received the message from the client *should not block*, waiting for the data. If it did block, you can imagine that this could eventually use up a great number of threads in the resource manager, with each thread waiting for some data from some device.

The correct solution to this is to store the receive ID that arrived with the client's message onto a queue somewhere, and return the special constant _RESMGR_NOREPLY from your handler. This tells the resource manager library that processing for this message has completed, but that the client shouldn't be unblocked yet.

Some time later, when the data arrives, you would then retrieve the receive ID of the client that was waiting for the message, and construct a reply message containing the data. Finally, you would reply to the client.

You could also extend this concept to implementing timeouts within the server, much as we did with the example in the Clocks, Timers, and Getting a Kick Every So Often chapter (in the *"Server-maintained timeouts"* section). To summarize, after some period of time, the client's request was deemed to have "timed out" and the server replied with some form of failure message to the receive ID it had stored away.

Returning directory entries

In the example for the read I/O function handler above, we saw how to return data. As mentioned in the description of the read I/O function handler (in the "Alphabetical listing of Connect and I/O

© 2020, QNX Software Systems Limited

functions"), the read I/O function handler may return directory entries as well. Since this isn't something that everyone will want to do, I discuss it here.

First of all, let's look at *why* and *when* you'd want to return directory entries rather than raw data from the read I/O function handler.

If you discretely manifest entries in the pathname space, and those entries are *not* marked with the _RESMGR_FLAG_DIR, then you won't have to return directory entries in the read I/O function handler. If you think about this from a "filesystem" perspective, you're effectively creating "file" types of objects. If, on the other hand, you *do* specify _RESMGR_FLAG_DIR, then you're creating a "directory" type of object. Nobody other than you knows what the *contents* of that directory are, so you have to be the one to supply this data. That's exactly why you'd return directory entries from your read I/O function handler.

Generally speaking...

Generally speaking, returning directory entries is just like returning raw data, except:

- You must return an *integral* number of `struct dirent` entries.
- You must fill in the `struct dirent` entries.

The first point means that you cannot return, for example, seven and a half `struct dirent` entries. If eight of these structures don't fit into the allotted space, then you must return only seven.

The second point is fairly obvious; it's mentioned here only because filling in the `struct dirent` can be a little tricky compared to the "raw data" approach for a "normal" read I/O function handler.

The `struct dirent` structure and friends

Let's take a look at the `struct dirent` structure, since that's the data structure returned by the read I/O function handler in case of a directory read. We'll also take a quick look at the client calls that deal with directory entries, since there are some interesting relations to the `struct dirent` structure.

In order for a client to work with directories, the client uses the functions *closedir()*, *opendir()*, *readdir()*, *rewinddir()*, *seekdir()*, and *telldir()*.

Notice the similarity to the "normal" file-type functions (and the commonality of the resource manager messages):

Directory Function	File Function	Message (resmgr)
closedir()	*close()*	_IO_CLOSE
opendir()	*open()*	_IO_CONNECT
readdir()	*read()*	_IO_READ
rewinddir()	*lseek()*	_IO_LSEEK
seekdir()	*lseek()*	_IO_LSEEK
telldir()	*lseek()*	_IO_LSEEK

If we assume for a moment that the *opendir()* and *closedir()* functions will be handled automatically for us, we can focus on just the _IO_READ and _IO_LSEEK messages and related functions.

Offsets

The _IO_LSEEK message and related function is used to "seek" (or "move") within a file. It does the exact same thing within a directory; you can move to the "first" directory entry (by explicitly giving an offset to *seekdir()* or by calling *rewinddir()*), or any arbitrary entry (by using *seekdir()*), or you can find out the current location in the directory entry list (by using *telldir()*).

The "trick" with directories, however, is that the seek offsets are *entirely up to you to define and manage*. This means that you may decide to call your directory entry offsets "0," "1," "2" and so on, or you may instead call them "0," "64," "128" and so on. The only important thing here is that the offsets must be consistent in both the set file position I/O function handler as well as the read I/O function handler.

In the example below, we'll assume that we're using the simple "0," "1," "2," ... approach. (You might use the "0," "64," "128," ... approach if those numbers correspond to, for example, some kind of on-media offsets. Your choice.)

Contents

So now all that's left is to "simply" fill in the `struct dirent` with the "contents" of our directory. Here's what the `struct dirent` looks like (from **<dirent.h>**):

```
struct dirent {
    ino_t       d_ino;
    off_t       d_offset;
    uint16_t    d_reclen;
    uint16_t    d_namelen;
    char        d_name [1];
};
```

Here's a quick explanation of the various members:

d_ino

The "inode"—a mountpoint-unique serial number that cannot be zero (zero traditionally indicates that the entry corresponding to this inode is free/empty).

d_offset

The offset into the directory we just talked about above. In our example, this will be a simple number like "0," "1," "2," etc. In some filesystems, this is the offset of the *next* directory.

d_reclen

The size of the entire `struct dirent` field and any extensions that may be placed within it. The size includes any alignment filler required.

d_namelen

The number of characters in the *d_name* field, *not including* the NUL terminator.

© 2020, QNX Software Systems Limited

d_name

The name of this directory entry, which must be NUL terminated.

When returning the `struct dirent` entries, the return code passed back to the client is the number of bytes returned.

Example

In this example, we're going to create a resource manager called **/dev/atoz** that will be a directory resource manager. It's going to manifest the "files" **/dev/atoz/a** through to **dev/atoz/z**, with a `cat` of any of the files returning the uppercase letter corresponding to the filename. Here's a sample command-line session to give you an idea of how this works:

```
# cd /dev
# ls
atoz    null    ptyp2   socket  ttyp0   ttyp3
enet0   ptyp0   ptyp3   text    ttyp1   zero
mem     ptyp1   shmem   tty     ttyp2
# ls -ld atoz
dr-xr-xr-x  1 root      0                26 Sep 05 07:59 atoz
# cd atoz
# ls
a       e       i       m       q       u       y
b       f       j       n       r       v       z
c       g       k       o       s       w
d       h       l       p       t       x
# ls -l e
-r--r--r--  1 root      0                 1 Sep 05 07:59 e
# cat m
M# cat q
Q#
```

The example above illustrates that the directory **atoz** shows up in the **/dev** directory, and that you can do an `ls` of the directory itself and `cd` into it. The **/dev/atoz** directory has a size of "26," which is the number that we selected in the code. Once in the **atoz** directory, doing another `ls` shows the contents: the files **a** through **z**. Doing an `ls` of a particular file, say **e**, shows that the file is readable by all (the `-r--r--r--` part) and is one byte in size. Finally, doing a few random `cat`'s shows that the files indeed have the stated contents. (Note that since the files contain only one byte, there's no linefeed after the character is printed, which is why the prompt shows up on the same line as the output.)

Now that we've seen the characteristics, let's take a look at the code, which is organized into the following functions:

main() **and declarations**

Main function; this is where we initialize everything and start the resource manager running.

my_open()

The handler routine for the _IO_CONNECT message.

my_read()

The handler routine for the _IO_READ message.

my_read_dir() **and** ***my_read_file()***

These two routines perform the actual work of the *my_read()* function.

dirent_size() **and** ***dirent_fill()***

Utility functions to deal with `struct dirent` structure.

Note that while the code is broken up here into several short sections with text, you can find the complete version of *atoz.c* in the Sample Programs appendix.

main() and declarations

The first section of code presented is the *main()* function and some of the declarations. There's a convenience macro, *ALIGN()*, that's used for alignment by the *dirent_fill()* and *dirent_size()* functions.

The *atoz_attrs* array contains the attributes structures used for the "files" in this example. We declare NUM_ENTS array members, because we have NUM_ENTS (26) files "**a**" through "**z**." The attributes structure used for the directory itself (i.e., the **/dev/atoz** directory) is declared within *main()* and is called simply *attr*. Notice the differences in the way the two types of attributes structures are filled:

file attribute structure

Marked as a regular file (the S_IFREG constant) with an access mode of 0444 (meaning everyone has read access, no one has write access). The size is "1"—the file contains only one byte, namely, the uppercase letter corresponding to the filename. The inodes for these individual files are numbered "1" through "26" inclusive (it would have been more convenient to number them "0" through "25," but "0" is reserved).

directory attribute structure

Marked as a directory file (the S_IFDIR constant) with an access mode of 0555 (meaning that everyone has read and seek access, no one has write access). The size is "26"—this is simply a number picked based on the number of entries in the directory. The inode is "27"—a number known not to be in use by any of the other attributes structures.

Notice how we've overridden only the *open* member of the *connect_func* structure and the *read* member of the *io_func* structure. We've left all the others to use the POSIX defaults.

Finally, notice how we created the name **/dev/atoz** using *resmgr_attach()*. Most importantly, we used the flag _RESMGR_FLAG_DIR, which tells the process manager that it can resolve requests *at and below* this mountpoint.

```
/*
 *  atoz.c
 *
 *  /dev/atoz using the resource manager library
*/
```


© 2020, QNX Software Systems Limited

```
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <errno.h>
#include <dirent.h>
#include <limits.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>

#define ALIGN(x) (((x) + 3) & ~3)
#define NUM_ENTS            26

static  iofunc_attr_t   atoz_attrs [NUM_ENTS];

int
main (int argc, char **argv)
{
    dispatch_t              *dpp;
    resmgr_attr_t           resmgr_attr;
    dispatch_context_t      *ctp;
    resmgr_connect_funcs_t  connect_func;
    resmgr_io_funcs_t       io_func;
    iofunc_attr_t           attr;
    int                     i;

    // create the dispatch structure
    if ((dpp = dispatch_create ()) == NULL) {
        perror ("Unable to dispatch_create");
        exit (EXIT_FAILURE);
    }

    // initialize the various data structures
    memset (&resmgr_attr, 0, sizeof (resmgr_attr));
    resmgr_attr.nparts_max = 1;
    resmgr_attr.msg_max_size = 2048;

    // bind default functions into the outcall tables
    iofunc_func_init (_RESMGR_CONNECT_NFUNCS, &connect_func,
                      _RESMGR_IO_NFUNCS, &io_func);

    // create and initialize the attributes structure
    // for the directory.  Inodes 1-26 are reserved for the
    // files 'a' through 'z'.  The number of bytes is 26
    // because that's how many entries there are.
    iofunc_attr_init (&attr, S_IFDIR | 0555, 0, 0);
    attr.inode = NUM_ENTS + 1;
    attr.nbytes = NUM_ENTS;

    // and for the "a" through "z" names
    for (i = 0; i < NUM_ENTS; i++) {
        iofunc_attr_init (&atoz_attrs [i],
                          S_IFREG | 0444, 0, 0);
        atoz_attrs [i].inode = i + 1;
```

```
        atoz_attrs [i].nbytes = 1;
    }

    // add our functions; we're interested only in
    // the open connect function handler and read I/O function handlers
    connect_func.open = my_open;
    io_func.read = my_read;
    io_func.read64 = my_read;

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

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

    // wait here forever, handling messages
    while (1) {
        if ((ctp = dispatch_block (ctp)) == NULL) {
            perror ("Unable to dispatch_block");
            exit (EXIT_FAILURE);
        }
        dispatch_handler (ctp);
    }

    // you'll never get here
    return (EXIT_SUCCESS);
}
```

my_open()

While *my_open()* is very short, it has a number of crucial points.

Notice how we decide if the resource being opened is a “file” or a “directory”, based only on the pathname length. We can do this “trick” because we know that there are no other directories in this resource manager apart from the main one. If you want to have multiple directories below the mountpoint, you have to do more complicated analysis of the *path* member of the *msg* structure. For our simple example, if there's nothing in the pathname, we know it's the directory. Also, notice the extremely simplified pathname validation checking: we simply compare to make sure that there's only one character passed to us, and that the character lies within the range “a” through “z” inclusive. Again, for more complex resource managers, you'd be responsible for parsing the name past the registered mountpoint.

Now, the most important feature! Notice how we used the POSIX-layer default functions to do all the work for us! The *iofunc_open_default()* function is usually installed in the connect functions table at the same spot that our new *my_open()* function is now occupying. This means that it takes the identical set of arguments! All we have to do is decide *which* attributes structure we want to have bound with the OCB that the default function is going to create: either the directory one (in which case we pass

 © 2020, QNX Software Systems Limited

attr), or one of the 26 different ones for the 26 different files (in which case we pass an appropriate element out of *atoz_attrs*). This is key, because the handler that you put in the *open* slot in the connect functions table acts as the gatekeeper to all further accesses to your resource manager.

```
static int
my_open (resmgr_context_t *ctp, io_open_t *msg,
         iofunc_attr_t *attr, void *extra)
{
    // an empty path means the directory, is that what we have?
    if (msg -> connect.path [0] == 0) {
        return (iofunc_open_default (ctp, msg, attr, extra));

    // else check if it's a single char 'a' -> 'z'
    } else if (msg -> connect.path [1] == 0 &&
               (msg -> connect.path [0] >= 'a' &&
                msg -> connect.path [0] <= 'z')) {

        // yes, that means it's the file (/dev/atoz/[a-z])
        return (iofunc_open_default (ctp, msg,
                atoz_attrs + msg -> connect.path [0] - 'a',
                extra));
    } else {
        return (ENOENT);
    }
}
```

my_read()

In the *my_read()* function, to decide what kind of processing we needed to do, we looked at the attribute structure's *mode* member. If the *S_ISDIR()* macro says that it's a directory, we call *my_read_dir()*; if the *S_ISREG()* macro says that it's a file, we call *my_read_file()*. (For details about these macros, see the entry for *stat()* in the QNX Neutrino *C Library Reference*.) Note that if we can't tell what it is, we return EBADF; this indicates to the client that something bad happened.

The code here doesn't know anything about our special devices, nor does it care; it simply makes a decision based on standard, well-known data.

```
static int
my_read (resmgr_context_t *ctp, io_read_t *msg,
         iofunc_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 -> mode)) {
        return (my_read_dir (ctp, msg, ocb));
    } else if (S_ISREG (ocb -> attr -> mode)) {
        return (my_read_file (ctp, msg, ocb));
    } else {
        return (EBADF);
    }
}
```

my_read_dir()

In *my_read_dir()* is where the fun begins. From a high level perspective, we allocate a buffer that's going to hold the result of this operation (called *reply_msg*). We then use *dp* to "walk" along the output buffer, stuffing `struct dirent` entries as we go along. The helper routine *dirent_size()* is used to determine if we have sufficient room in the output buffer to stuff the next entry; the helper routine *dirent_fill()* is used to perform the stuffing. (Note that these routines are not part of the resource manager library; they're discussed and documented below.)

On first glance this code may look inefficient; we're using *sprintf()* to create a two-byte filename (the filename character and a NUL terminator) into a buffer that's _POSIX_PATH_MAX (256) bytes long. This was done to keep the code as generic as possible.

Finally, notice that we use the OCB's *offset* member to indicate to us which particular filename we're generating the `struct dirent` for at any given time. This means that we also have to update the *offset* field whenever we return data.

The return of data to the client is accomplished in the "usual" way, via *MsgReply()*. Note that the status field of *MsgReply()* is used to indicate the number of bytes that were sent to the client.

```
static int
my_read_dir (resmgr_context_t *ctp, io_read_t *msg,
             iofunc_ocb_t *ocb)
{
    size_t  nbytes;
    size_t  nleft;
    struct  dirent *dp;
    char    *reply_msg;
    char    fname [_POSIX_PATH_MAX];

    // allocate a buffer for the reply
    reply_msg = calloc (1, _IO_READ_GET_NBYTES(msg));
    if (reply_msg == NULL) {
        return (ENOMEM);
    }
```


© 2020, QNX Software Systems Limited

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

    // we have "nleft" bytes left
    nleft = _IO_READ_GET_NBYTES(msg);
    while (ocb -> offset < NUM_ENTS) {

        // create the filename
        sprintf (fname, "%c", ocb -> offset + 'a');

        // 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;
        }
    }

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

    // release our buffer
    free (reply_msg);

    // tell resource manager library we already did the reply
    return (_RESMGR_NOREPLY);
}
```

my_read_file()

In *my_read_file()*, we see much the same code as we saw in the simple read example above. The only strange thing we're doing is we "know" there's only one byte of data being returned, so if *nbytes* is nonzero, then it must be one (and nothing else). So, we can construct the data to be returned to the client by stuffing the character variable *string* directly. Notice how we used the *inode* member of the attribute structure as the basis of which data to return. This is a common trick used in resource managers that must deal with multiple resources. Another trick would be to extend the attributes structure (as discussed above in "Extending the attributes structure") and have either the data stored there directly or a pointer to it.

```
static int
my_read_file (resmgr_context_t *ctp, io_read_t *msg,
              iofunc_ocb_t *ocb)
{
    size_t  nbytes;
    size_t  nleft;
    char    string;

    // 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 -> nbytes - ocb -> offset;

    // and how many we can return to the client
    nbytes = min (nleft, _IO_READ_GET_NBYTES(msg));

    if (nbytes) {
        // create the output string
        string = ocb -> attr -> inode - 1 + 'A';

        // return it to the client
        MsgReply (ctp -> rcvid, nbytes,
                  &string + ocb -> offset,
                  nbytes);

        // update flags and offset
        ocb -> attr -> flags |= IOFUNC_ATTR_ATIME
                             | IOFUNC_ATTR_DIRTY_TIME;
        ocb -> offset += nbytes;
    } else {
```


© 2020, QNX Software Systems Limited

```
        // nothing to return, indicate End Of File
        MsgReply (ctp -> rcvid, EOK, NULL, 0);
    }

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

dirent_size()

The helper routine *dirent_size()* simply calculates the number of bytes required for the `struct dirent`, given the alignment constraints:

```
int
dirent_size (char *fname)
{
  return (ALIGN (sizeof (struct dirent) - 4 + strlen (fname) + 1));
}
```

We subtract four bytes because the `dirent` structure includes space for the first four characters of the name, and we add one for the null character at the end of the name. We could also calculate the size like this:

```
ALIGN (offsetof (struct dirent, d_name) + strlen (fname) + 1)
```

Again, this routine is slight overkill for our simple resource manager, because we know how big each directory entry is going to be—all filenames are exactly one byte in length. However, it's a useful utility routine.

dirent_fill()

Finally, the helper routine *dirent_fill()* is used to stuff the values passed to it (namely, the *inode*, *offset*, and *fname* fields) into the directory entry also passed. As an added bonus, it returns a pointer to where the next directory entry should begin, taking into account alignment.

```
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 + 1);
    return ((struct dirent *) ((char *) dp +
            dp -> d_reclen));
}
```

Summary

Writing a resource manager is by far the most complicated task that we've discussed in this book.

A resource manager is a server that receives certain, well-defined messages. These messages fall into two broad categories:

Connect messages

Related to pathname-based operations, these may establish a context for further work.

I/O messages

Always arrive *after* a connect message and indicate the actual work that the client wishes to have done (e.g., *stat()*).

The operations of the resource manager are controlled by the thread pool functions (discussed in the Processes and Threads chapter) and the dispatch interface functions.

QNX Neutrino provides a set of POSIX helper functions in the resource manager library that perform much of the work of dealing with the client's Connect and I/O messages that arrive.

There are a number of data structures relating to the clients and devices manifested by the resource manager to keep in mind:

OCB

Allocated on a per-open basis, this contains the context for the client (e.g., current *lseek()* position)

Attributes structure

Allocated on a per-device basis, this contains information about the device (e.g., size of the device, permissions, etc.)

Mount structure

Allocated on a per-resource-manager basis, and contains information about the characteristics of the entire resource manager.

The clients communicate with the resource manager via message passing by resolving the pathname (via the *open()* and other calls) into a node descriptor, process ID, channel ID, and handle.

Finally you supply the functionality you wish to actually *do* in your resource manager by overriding some of the callouts in the Connect and I/O functions table.

© 2020, QNX Software Systems Limited

Appendix A
QNX 4 to QNX Neutrino

In this appendix, we'll take a look at QNX Software Systems' previous operating system, QNX 4, and see how it compares to QNX Neutrino. This appendix will mainly be of interest if you are a current QNX 4 customer and want to see:

- What's so great about QNX Neutrino?
- How hard will it be when I port to QNX Neutrino?

Or you may be developing for, or porting to, both operating systems.

Similarities

Let's first start with how the two generations of operating systems are similar:

- message passing is at the heart of the architecture
- network-distributed message passing
- realtime
- microkernel architecture
- processes are memory-protected
- POSIX compatibility
- relatively simple “device driver” model
- embeddable

Note that while some of the basic features listed above are indeed similar, in general QNX Neutrino has extended the support. For example, QNX Neutrino has more POSIX support than QNX 4, simply because a large number of the POSIX specifications were still in draft status when QNX 4 was released. While less of them are in draft status as of QNX Neutrino's release, there are still more *new* drafts being released as this book is written. It's a never-ending game of catch-up.

© 2020, QNX Software Systems Limited

Improvements

Now that you've seen what's the same about the two generations of OS, let's look at where QNX Neutrino has improved functionality over QNX 4:

- more POSIX standards supported
- more embeddable
- kernel is more readily customizable for a variety of hardware platforms
- thread support
- simpler device driver model
- portable architecture; currently supports ARM processors as well as x86
- supports SMP
- more documentation

While some of these improvements are “free,” meaning that there are no compatibility issues (for example, POSIX pthreads weren't supported under QNX 4), some things did require fundamental changes. I'll briefly mention the classes of changes that were required, and then we'll look in detail at the compatibility issues caused as well as suggestions on how to port to QNX Neutrino (or keep your code portable between the two).

Embeddability

QNX Neutrino totally redesigned the way that the operating system was embedded. Under QNX 4, in the original release, it was marginally embeddable. Then QNX Neutrino came along, designed to be embeddable. As a bonus, QNX 4 underwent some changes as a result of the experience gained in QNX Neutrino, and now QNX 4 is vastly more embeddable than it had been. In any event, embedding QNX 4 versus embedding QNX Neutrino is almost like night and day. QNX 4 has no real support for things like:

- kernel callouts (interrupt, timer)
- startup configurability
- image filesystem

whereas QNX Neutrino does. The definitive book on that subject is *Building Embedded Systems*.

Thread support

QNX 4 had a function called *tfork()* that let you use “threads” by creating a process with its code *and data* segments mapped to the same memory locations as the creating process. This gave the illusion of a thread by creating a process, and then changing the characteristics of the newly created process to make it look like a thread. While there is a thread library available for QNX 4 on QNX Software Systems' update system, the kernel itself doesn't support threads directly.

Under QNX Neutrino, the POSIX “pthread” model is used for all threading. This means that you'll see (and have seen in this book) familiar function calls like *pthread_create()*, *pthread_mutex_lock()*, and others.

Message passing

While the impact of threads on message passing may seem minimal, it resulted in a fundamental change to the way message passing was done (not to the fundamental *concepts* of message passing, like SEND/RECEIVE/REPLY, but to the *implementation*).

Under QNX 4, messages were targeted at process IDs. To send a message, you simply found the process ID of the target and did your *Send()*. For servers to receive a message under QNX 4 they just did a *Receive()*. This would block until a message arrived. The server would then reply with the *Reply()* function.

Under QNX Neutrino, message passing is identical (different function names, though). What's changed is the mechanism. The client now has to create a connection to a server before it can do the standard message-passing functions. And the server has to create a channel before it can do the standard message-passing functions.

Note that the QNX 4 *Creceive()* function, which would do a non-blocking *Receive()*, is missing from QNX Neutrino. We generally discourage such "polling" functions, especially when you can start a thread, but if you really insist on performing a non-blocking *MsgReceive()*, you should take a look at the *Clocks, Timers, and Getting a Kick Every So Often chapter (under "Kernel timeouts")*) for more information. For the short story version, here's the relevant code sample:

```
TimerTimeout (CLOCK_MONOTONIC, _NTO_TIMEOUT_RECEIVE,
              NULL, NULL, NULL);
rcvid = MsgReceive (…
```

Pulses and events

QNX 4 provided something called a "proxy." A proxy is best described as a "canned" (or "fixed") message, which could be sent by processes or kernel services (like a timer or interrupt service routine) to the owner of the proxy. The proxy is non-blocking for the sender and would arrive just like any other message. The way to identify a proxy (as opposed to another process actually sending a message) was to either look at the proxy message contents (not 100% reliable, as a process could send something that *looked* like the contents of the proxy) or to examine the process ID associated with the message. If the process ID of the message was the same as the proxy ID, then you could be assured it was a proxy, because proxy IDs and process IDs were taken from the same pool of numbers (there'd be no overlap).

QNX Neutrino extends the concept of proxies with "pulses." Pulses are still non-blocking messages, they can still be sent from a thread to another thread, or from a kernel service (like the timer and ISR mentioned above for proxies) to a thread. The differences are that while proxies were of fixed-content, QNX Neutrino pulses are fixed-length, but the content can be set by the sender of the pulse at any time. For example, an ISR could save away a key piece of data into the pulse and then send that to a thread.

Under QNX 4, some services were able to deliver a signal or a proxy, while other services were able to deliver only one or the other. To complicate matters, the delivery of these services was usually done in several different ways. For example, to deliver a signal, you'd have to use the *kill()* function. To deliver a proxy or signal as a result of a timer, you'd have to use a negative signal number (to indicate

© 2020, QNX Software Systems Limited

it was a proxy) or a positive signal number (to indicate it was a signal). Finally, an ISR could deliver *only* a proxy.

Under QNX Neutrino this was abstracted into an extension of the POSIX `struct sigevent` data structure. Anything that used or returned the `struct sigevent` structure can use a signal or a pulse.

In fact, this has been extended further, in that the `struct sigevent` can even cause a thread to be created! We talked about this in the *Clocks, Timers, and Getting a Kick Every So Often chapter (under "Getting notified with a thread"*).

Device driver model

Under the previous-previous version of the operating system (the QNX 2 family), writing device drivers was an arcane black art. Under QNX 4, it was initially a mystery, but then eventually some samples appeared. Under QNX Neutrino, there are books and courses on the topic. As it turns out, the QNX Neutrino model and the QNX 4 model are, at the highest architectural level, reasonably similar. Whereas QNX 4 had somewhat muddled concepts of what needed to be done as a "connect" function, and what needed to be done as an "I/O" function, QNX Neutrino has a very clear separation. Also, under QNX 4, you (the device driver writer) were responsible for most of the work—you'd supply the main message handling loop, you'd have to associate context on each I/O message, and so on. QNX Neutrino has simplified this greatly with the resource manager library.

ARM support

One of the driving changes behind the embeddability differences between QNX 4 and QNX Neutrino is the fact that QNX Neutrino supports ARM processors. Whereas QNX 4 was initially "at home" on an IBM PC with a BIOS and *very* standard hardware, QNX Neutrino is equally at home on multiple processor platforms with or without a BIOS (or ROM monitor), and with customized hardware chosen by the manufacturer (often, it would appear, without regard for the requirements of the OS). This means that the microkernel had to have provision for callouts, so you could, for example, decide what kind of interrupt controller hardware you had, and, without having to buy a source license for the operating system, run on that hardware.

A bunch of other changes you'll notice when you port QNX 4 applications to QNX Neutrino, especially on these different processor platforms, is that they're fussy about alignment issues. You can't access an N-byte object on anything other than an N-byte multiple of an address. Under the x86 (with the alignment flag turned off), you could access memory willy-nilly. By modifying your code to have properly aligned structures (for non-x86 processors), you'll also find that your code runs faster on x86, because the x86 processor can access aligned data faster.

Another thing that often comes to haunt people is the issue of big-endian versus little-endian. The x86 processor is a mono-endian processor (meaning it has only one "endian-ness"), and that's little-endian. ARM, for example, are bi-endian processors (meaning that the processor can operate in either big-endian or little-endian mode). Furthermore, these non-x86 processors are "RISC" (Reduced Instruction Set CPU) machines, meaning that certain operations, such as a simple C language `|=` (bitwise set operation) may or may not be performed in an atomic manner. This can have startling consequences! Look at the file **<atomic.h>** for a list of helper functions that ensure atomic operation.

SMP support

Released versions of QNX 4 are strictly single-processor, whereas QNX Neutrino has support for SMP.

SMP is a great feature, especially in an operating system that supports threads, but it's also a bigger gun that you can shoot yourself in the foot with. For example, on a single-processor box, an ISR will preempt a thread, but *never* the other way around. On a single-processor box, it's a worthwhile abstraction to “pretend” that threads run simultaneously, when they don't really.

On an SMP box, a thread and ISR can be running simultaneously, and multiple threads can also be running simultaneously. Not only is an SMP system a great workstation, it's also an excellent SQA (Software Quality Assurance) testing tool—if you've made any “bad” assumptions about protection in a multithreaded environment, an SMP system will find them eventually.

To illustrate just how true that statement is, one of the bugs in an early internal version of SMP had a “window” of *one machine cycle*! On one processor, what was supposedly coded to be an atomic read/modify/write operation could be interfered with by the second processor's compare and exchange instruction.

© 2020, QNX Software Systems Limited

Porting philosophy

Let's now turn our attention to the "big picture." We'll look at:

- *Message passing and clients & servers*
- *Interrupt Service Routines*

Message passing considerations

Under QNX 4, the way a client would find a server was either:

1. Use the global namespace.

   Or:

2. Perform an *open()* on an I/O manager.

Client/server using the global namespace

If the client/server relationship that you're porting depended on the global namespace, then the client used *qnx_name_locate()*, and the server would "register" its name via *qnx_name_attach()*.

In this case, you have two choices. You can try to retain the global namespace idiom, or you can modify your client and server to act like a standard resource manager. If you wish to retain the global namespace, then you should look at the *name_attach()* and *name_detach()* functions for your server, and *name_open()* and *name_close()* for your clients.

However, I'd recommend that you do the latter; it's "the QNX Neutrino way" to do everything with resource managers, rather than try to bolt a resource manager "kludge" onto the side of a global namespace server.

The modification is actually reasonably simple. Chances are that the client side calls a function that returns either the process ID of the server or uses the "VC" (*V*irtual *C*ircuit) approach to create a VC from the client's node to a remote server's node. In both cases, the process ID or the VC to the remote process ID was found based on calling *qnx_name_locate()*.

Here, the "magic cookie" that binds the client to the server is some form of process ID (we're considering the VC to be a process ID, because VCs are taken from the same number space, and for all intents and purposes, they look just like process IDs).

If you were to return a connection ID instead of a process ID, you'd have conquered the major difference. Since the QNX 4 client probably doesn't examine the process ID in any way (what meaning would it have, anyway?—it's just a number), you can probably trick the QNX 4 client into performing an *open()* on the "global name."

In this case, however, the global name would be the pathname that the resource manager attached as its "id." For example, the following is typical QNX 4 client code, stolen from my caller ID (CLID) server library:

```
/*
 *  CLID_Attach (serverName)
 *
```

```
 *  This routine is responsible for establishing a connection to
 *  the CLID server.
 *
 *  Returns the process ID or VC to the CLID server.
*/

// a place to store the name, for other library calls
static char CLID_serverName [MAX_CLID_SERVER_NAME + 1];

// a place to store the clid server id
static int clid_pid = -1;

int
CLID_Attach (char *serverName)
{
    if (serverName == NULL) {
        sprintf (CLID_serverName, "/PARSE/CLID");
    } else {
        strcpy (CLID_serverName, serverName);
    }
    clid_pid = qnx_name_locate (0, CLID_serverName,
                                sizeof (CLID_ServerIPC), NULL);
    if (clid_pid != -1) {
        CLID_IPC (CLID_MsgAttach);  // send it an ATTACH message
        return (clid_pid);
    }
    return (-1);
}
```

You could change this to be:

```
/*
 *  CLID_Attach (serverName) QNX Neutrino version
*/

int
CLID_Attach (char *serverName)
{
    if (serverName == NULL) {
        sprintf (CLID_serverName, "/PARSE/CLID");
    } else {
        strcpy (CLID_serverName, serverName);
    }
```


© 2020, QNX Software Systems Limited

```
    return (clid_pid = open (CLID_serverName, O_RDWR));
}
```

and the client wouldn't even notice the difference.

Two implementation notes: I've simply left the default name "**/PARSE/CLID**" as the registered name of the resource manager. Most likely a better name would be "**/dev/clid**"—it's up to you how "POSIX-like" you want to make things. In any event, it's a one-line change and is only marginally related to the discussion here.

The second note is that I've still *called* the file descriptor *clid_pid*, even though now it should *really* be called *clid_fd*. Again, this is a style issue and relates to just how much change you want to perform between your QNX 4 version and the QNX Neutrino one.

In any event, to be totally portable to both, you'll want to abstract the client binding portion of the code into a function call—as I did above with the *CLID_Attach()*.

At some point, the client would actually perform the message pass operation. This is where things get a little trickier. Since the client/server relationship is *not* based on an I/O manager relationship, the client generally creates "customized" messages. Again from the CLID library (*CLID_AddSingleNPANXX()* is the client's exposed API call; I've also included *checkAttach()* and *CLID_IPC()* to show the actual message passing and checking logic):

```
/*
 *  CLID_AddSingleNPANXX (npa, nxx)
*/

int
CLID_AddSingleNPANXX (int npa, int nxx)
{
    checkAttach ();
    CLID_IPCData.npa = npa;
    CLID_IPCData.nxx = nxx;
    CLID_IPC (CLID_MsgAddSingleNPANXX);
    return (CLID_IPCData.returnValue);
}

/*
 *  CLID_IPC (IPC message number)
 *
 *  This routine will call the server with the global buffer
 *  CLID_IPCData, and will stuff in the message number passed
 *  as the argument.
 *
 *  Should the server not exist, this routine will stuff the
 *  .returnValue field with CLID_NoServer.  Otherwise, no
```

```
 *  fields are affected.
*/

void
CLID_IPC (IPCMessage)
int     IPCMessage;
{
    if (clid_pid == -1) {
        CLID_IPCData.returnValue = CLID_NoServer;
        return;
    }
    CLID_IPCData.serverFunction = IPCMessage;
    CLID_IPCData.type = 0x8001;
    CLID_IPCData.subtype = 0;
    if (Send (clid_pid, &CLID_IPCData, &CLID_IPCData,
              sizeof (CLID_IPCData),
              sizeof (CLID_IPCData))) {
        CLID_IPCData.returnValue = CLID_IPCError;
        return;
    }
}

void
checkAttach ()
{
    if (clid_pid == -1) {
        CLID_Attach (NULL);
    }
}
```

As you can see, the *checkAttach()* function is used to ensure that a connection exists to the CLID server. If you didn't have a connection, it would be like calling *read()* with an invalid file descriptor. In my case here, the *checkAttach()* automagically creates the connection. It would be like having the *read()* function determine that there is no valid file descriptor and just create one out of the blue. Another style issue.

The customized messaging occurs in the *CLID_IPC()* function. It takes the global variable *CLID_IPCData* and tries to send it to the server using the QNX 4 *Send()* function.

The customized messages can be handled in one of two ways:

1. Functionally translate them into standard, file-descriptor-based POSIX calls.

   Or:

2. Encapsulate them into either a *devctl()* or a customized message wrapper using the _IO_MSG message type.

© 2020, QNX Software Systems Limited

In both cases, you've effectively converted the client to communicating using standard resource manager mechanisms for communications. What? You don't have a file descriptor? You have only a connection ID? Or vice versa? This isn't a problem! Under QNX Neutrino, *a file descriptor is a connection ID*!

Translating messages to standard file-descriptor-based POSIX calls

In the case of the CLID server, this really isn't an option. There is no standard POSIX file-descriptor-based call to "add an NPA/NXX pair to a CLID resource manager." However, there is the general *devctl()* mechanism, so if your client/server relationship requires this form, see "Translating messages to *devctl()* or _IO_MSG," below.

Now, before you write off this approach (translating to standard fd-based messages), let's stop and think about some of the places where this would be useful. In an audio driver, you *may* have used customized QNX 4 messages to transfer the audio data to and from the resource manager. When you really look at it, *read()* and *write()* are probably much more suited to the task at hand—bulk data transfer. Setting the sampling rate, on the other hand, would be much better accomplished via the *devctl()* function.

Granted, not every client/server relationship will have a bulk data transfer requirement (the CLID server is such an example).

Translating messages to *devctl()* or _IO_MSG

So the question becomes, how do you perform control operations? The easiest way is to use the *devctl()* call.

Our CLID library example (above) now becomes:

```
/*
 *  CLID_AddSingleNPANXX (npa, nxx)
*/

int
CLID_AddSingleNPANXX (int npa, int nxx)
{
    struct  clid_addnpanxx_t    msg;

    checkAttach ();  // keep or delete, style issue

    msg.npa = npa;
    msg.nxx = nxx;
    return (devctl (clid_pid, DCMD_CLID_ADD_NPANXX, &msg,
                    sizeof (msg), NULL));
}
```

As you can see, this was a relatively painless operation. (For those people who don't like *devctl()* because it forces data transfers to be the same size in both directions, see the discussion below on the _IO_MSG message.) Again, if you're maintaining source that needs to run on both operating systems, you'd abstract the message-passing function into one common point, and then supply different versions of a library, depending on the operating system.

We actually killed two birds with one stone:

1. Removed a global variable, and assembled the messages based on a stack variable—this now makes our code thread-safe.
2. Passed only the correct-sized data structure, instead of the maximum-sized data structure as we did in the previous (QNX 4) example.

Note that we had to define DCMD_CLID_ADD_NPANXX—we could have also kludged around this and used the CLID_MsgAddSingleNPANXX manifest constant (with appropriate modification in the header file) for the same purpose. I just wanted to highlight the fact that the two constants weren't identical.

The second point that we made in the list above (about killing birds) was that we passed only the "correct-sized data structure." That's actually a tiny lie. You'll notice that the *devctl()* has only one size parameter (the fourth parameter, which we set to `sizeof (msg)`). How does the data transfer *actually* occur? The second parameter to *devctl()* contains the device command (hence "DCMD").

Encoded within the top two bits of the device command is the direction, which can be one of four possibilities:

1. "00"—no data being transferred
2. "01"—transfer from driver to client
3. "10"—transfer from client to driver
4. "11"—transfer bidirectionally

If you're not transferring data (meaning that the command itself suffices), or if you're transferring data unidirectionally, then *devctl()* is fine.

The interesting case is when you're transferring data bidirectionally, because (since there's only one data size parameter to *devctl()*) both data transfers (to the driver and back) will transfer the entire data buffer! This is okay in the sub-case where the "input" and "output" data buffer sizes are identical, but consider the case where the data buffer going to the driver is a few bytes, and the data coming back from the driver is large.

Since we have only one size parameter, we're effectively forced to transfer the entire data buffer to the driver, even though only a few bytes were required!

This can be solved by "rolling your own" messages, using the general "escape" mechanism provided by the _IO_MSG message.

The _IO_MSG message is provided to allow you to add your own message types, while not conflicting with any of the "standard" resource manager message types—it's already a resource manager message type.

The first thing that you must do when using _IO_MSG is define your particular "custom" messages. In this example, we'll define two types, and model it after the standard resource manager messages—one data type will be the input message, and one will be the output message:

```
typedef struct
{
    int     data_rate;
    int     more_stuff;
} my_input_xyz_t;
```


© 2020, QNX Software Systems Limited

```
typedef struct
{
    int     old_data_rate;
    int     new_data_rate;
    int     more_stuff;
} my_output_xyz_t;

typedef union
{
    my_input_xyz_t  i;
    my_output_xyz_t o;
} my_message_xyz_t;
```

Here, we've defined a `union` of an input and output message, and called it `my_message_xyz_t`. The naming convention is that this is the message that relates to the "`xyz`" service, whatever that may be. The input message is of type `my_input_xyz_t`, and the output message is of type `my_output_xyz_t`. Note that "input" and "output" are from the point of view of the resource manager—"input" is data going *into* the resource manager, and "output" is data coming *from* the resource manager (back to the client).

We need to make some form of API call for the client to use—we *could* just force the client to manually fill in the data structures `my_input_xyz_t` and `my_output_xyz_t`, but I don't recommend doing that. The reason is that the API is supposed to "decouple" the implementation of the message being transferred from the functionality. Let's assume this is the API for the client:

```
int
adjust_xyz (int *data_rate,
            int *odata_rate,
            int *more_stuff);
```

Now we have a well-documented function, *adjust_xyz()*, that performs something useful from the client's point of view. Note that we've used pointers to integers for the data transfer—this was simply an example of implementation. Here's the source code for the *adjust_xyz()* function:

```
int
adjust_xyz (int *dr, int *odr, int *ms)
{
    my_message_xyz_t    msg;
    int                 sts;

    msg.i.data_rate = *dr;
    msg.i.more_stuff = *ms;
    sts = io_msg (global_fd, COMMAND_XYZ, &msg,
                  sizeof (msg.i),
                  sizeof (msg.o));
    if (sts == EOK) {
        *odr = msg.o.old_data_rate;
```

```
        *ms = msg.o.more_stuff;
    }
    return (sts);
}
```

This is an example uses *io_msg()* (the user-defined message I/O function handler, which we'll define shortly—it's *not* a standard QNX-supplied library call!). The *io_msg()* function does the magic of assembling the _IO_MSG message. To get around the problems that we discussed about *devctl()* having only one "size" parameter, we've given *io_msg()* two size parameters, one for the input (to the resource manager, `sizeof (msg.i)`) and one for the output (from the resource manager, `sizeof (msg.o)`). Notice how we update the values of **odr* and **ms only* if the *io_msg()* function returns an EOK.

This is a common trick, and is useful in this case because the passed arguments don't get modified *unless* the actual command succeeded. (This prevents the client program from having to maintain copies of its passed data, just in case the function fails.)

One last thing that I've done in the *adjust_xyz()* function, is that I depend on the *global_fd* variable containing the file descriptor of the resource manager. Again, there are a number of ways that you could handle it:

- Bury the file descriptor within the user-defined message I/O function handler (e.g., *io_msg()*) (this would be useful if you wanted to avoid having to pass around the file descriptor on each and every call; useful if you're ever going to talk to only the one resource manager, and thus most likely not suitable as a general purpose solution).

  Or:

- Pass the file descriptor from the client itself to each function in the API library (useful if the client's going to be responsible for talking to the resource manager in other ways, such as the standard POSIX file descriptor calls like *read()*, or if the client may be talking to multiple resource managers).

Here's the source for *io_msg()*:

```
long
io_msg (int fd, int cmd, void *msg, int isize, int osize)
{
    io_msg_t    io_message;
    iov_t       rx_iov [2];
    iov_t       tx_iov [2];
    int         sts;

    // set up the transmit IOV
    SETIOV (tx_iov + 0, &io_msg.o, sizeof (io_msg.o));
    SETIOV (tx_iov + 1, msg, osize);

    // set up the receive IOV
    SETIOV (rx_iov + 0, &io_msg.i, sizeof (io_msg.i));
    SETIOV (rx_iov + 1, msg, isize);
```


© 2020, QNX Software Systems Limited

```
    // set up the _IO_MSG itself
    memset (&io_message, 0, sizeof (io_message));
    io_message.type = _IO_MSG;
    io_message.mgrid = cmd;

    return (MsgSendv (fd, tx_iov, 2, rx_iov, 2));
}
```

Notice a few things.

The *io_msg()* function used a two-part IOV to "encapsulate" the custom message (as passed by *msg*) into the *io_message* structure.

The *io_message* was zeroed out and initialized with the _IO_MSG message identification type, as well as the *cmd* (which will be used by the resource manager to decide what kind of message was being sent).

The *MsgSendv()* function's return status was used directly as the return status of *io_msg()*.

The only "funny" thing that we did was in the *mgrid* field. QNX Software Systems reserves a range of values for this field, with a special range reserved for "unregistered" or "prototype" drivers. These are values in the range _IOMGR_PRIVATE_BASE through to _IOMGR_PRIVATE_MAX that you can use for your resource manager. In our example above, we assume that COMMAND_XYZ is something based on _IOMGR_PRIVATE_BASE:

```
#define COMMAND_XYZ (_IOMGR_PRIVATE_BASE + 0x0007)
```

Client/Server using an I/O manager

Now, what if the client that you're porting used an I/O manager? How would we convert that to QNX Neutrino? Answer: we already did. Once we establish a file-descriptor-based interface, we're using a resource manager. Under QNX Neutrino, you'd almost *never* use a "raw" message interface. Why not?

1. You'd have to worry about the _IO_CONNECT message that came in with the client's *open()* call, or you'd have to figure out how to find the resource manager if you weren't going to use *open()*.
2. You'd have to figure out a way to associate a client with a particular context block inside of the resource manager. This isn't rocket science, but it does involve some amount of data management.
3. You'd have to provide encapsulation of all your messages, instead of using the standard POSIX file-descriptor-based functions to do that for you.
4. Your resource manager won't work with *stdin*/*stdout*-based applications. For the audio driver example, you couldn't just do `mp3_decode spud.mp3 >/dev/audio`; the *open()* would most likely fail (if not, then the *write()* would, and so on).

Proxies

Under QNX 4, the only way to send a non-blocking message was to create a proxy via *qnx_proxy_attach()*. This function returns a proxy ID (which is taken from the same number space as process IDs), which you can then *Trigger()* or return from an interrupt service routine (see below).

Under QNX Neutrino, you'd set up a `struct sigevent` to contain a "pulse," and either use *MsgDeliverEvent()* to deliver the event or bind the event to a timer or ISR.

The usual trick under QNX 4 to detect proxy messages (via *Receive()* or *Creceive()*) was to compare the process ID returned by the receiving function against the proxy IDs that you're expecting. If you got a match, you knew it was a proxy. Alternatively, you could ignore the process ID returned by the receiving function and handle the message as if it were a "regular" message. Unfortunately, this has some porting complications.

Proxies for their IDs

If you're comparing the received process ID against the list of proxies that you're expecting, then you'll usually ignore the actual contents of the proxy. After all, since the proxy message couldn't be changed once you've created it, what additional information would you have gained by looking at the message once you knew it was one of your proxies? You could argue that as a convenience you'd place a message into the proxy that you could then look at with your standard message decoding. If that's the case, see below, Proxies for their contents.

Therefore, under QNX 4, you'd see code like:

```
pid = Receive (0, &msg, sizeof (msg));
if (pid == proxyPidTimer) {
    // we got hit with the timer, do something
} else if (pid == proxyPidISR) {
    // our ISR went off, do something
} else {
    // not one of our proxies, must have been a regular
    // message for a client.  Do something.
}
```

Under QNX Neutrino, you'd replace this code with the following:

```
rcvid = MsgReceive (chid, &msg, sizeof (msg), NULL);
if (rcvid == 0) {   // 0 indicates it was a pulse
    switch (msg.pulse.code) {
    case    MyCodeTimer:
        // we got hit with the timer, do something
        break;
    case    MyCodeISR:
        // our ISR went off, do something
        break;
    default:
        // unknown pulse code, log it, whatever.
        break;
    }
} else {
    // rcvid is not zero, therefore not a pulse but a
    // regular message from a client.  Do something.
}
```


© 2020, QNX Software Systems Limited

Note that this example would be used if you're handling all messages yourself. Since we recommend using the resource manager library, your code would really look more like this:

```
int
main (int argc, char **argv)
{
    …
    // do the usual initializations

    pulse_attach (dpp, 0, MyCodeTimer, my_timer_pulse_handler,
                  NULL);
    pulse_attach (dpp, 0, MyCodeISR, my_isr_pulse_handler,
                  NULL);


    …
}
```

This time, we're telling the resource manager library to put the two checks that we showed in the previous example into its receive loop and call our two handling functions (*my_timer_pulse_handler()* and *my_isr_pulse_handler()*) whenever those codes show up. Much simpler.

Proxies for their contents

If you're looking at proxies for their contents (you're ignoring the fact that it's a proxy and just treating it like a message), then you already have to deal with the fact that you can't reply to a proxy under QNX 4. Under QNX Neutrino, you can't reply to a pulse. What this means is, you've already got code in place that either looks at the proxy ID returned by the receive function and determines that it shouldn't reply, or the proxy has encoded within it special indications that this is a message that shouldn't be replied to.

Unfortunately under QNX Neutrino, you can't stuff arbitrary data into a pulse. A pulse has a well-defined structure, and there's just no getting around that fact. A clever solution would be to “simulate” the message that you'd ordinarily receive from the proxy by using a pulse with a table. The table would contain the equivalent messages that would have been sent by the proxies. When a pulse arrives, you'd use the value field in the pulse as an index into this table and “pretend” that the given proxy message had arrived.

Interrupt service routines

QNX 4's interrupt service routines had the ability to either return a proxy ID (indicating that the proxy should be sent to its owner) or a zero, indicating nothing further needed to be done. Under QNX Neutrino, this mechanism is almost identical, except that instead of returning a proxy, you're returning a pointer to a `struct sigevent`. The event that you return can be a pulse, which will give you the “closest” analog to a proxy, or it can be a signal or the creation of a thread. Your choice.

Also, under QNX 4 you *had* to have an interrupt service routine, even if all that the ISR did was return a proxy and nothing else. Under QNX Neutrino, using *InterruptAttachEvent()*, you can bind a `struct sigevent` to an interrupt vector, and that event will be delivered every time the vector is activated.

Summary

Porting from QNX 4 to QNX Neutrino, or maintaining a program that must function on both, is possible, if you follow these rules:

- abstract, abstract, and abstract
- decouple, decouple, and decouple

The key is to not tie yourself to a particular “handle” that represents the “connection” between the client and the server, and to not rely on a particular mechanism for finding the server. If you abstract the connection and the detection services into a set of function calls, you can then conditionally compile the code for whatever platform you wish to port to.

The exact same discussion applies to the message transport—always abstract the client's API away from “knowing” how the messages are transported from client to server to some generic API which can then rely upon a single-point transport API; this single-point transport API can then be conditionally compiled for either platform.

Porting a *server* from QNX 4 to QNX Neutrino is more difficult, owing to the fact that QNX 4 servers were generally “hand-made” and didn't follow a rigorous structure like that imposed by the resource manager library under QNX Neutrino. Generally, though, if you're porting something hardware specific (for example, a sound card driver, or a block-level disk driver), the main “code” that you'll be porting has nothing to do with the operating system, and everything to do with the hardware itself. The approach I've adopted in these cases is to code a shell “driver” structure, and provide well-defined hardware-specific functions. The entire shell driver will be different between operating systems, but the hardware-specific functions can be amazingly portable.

© 2020, QNX Software Systems Limited

Appendix B
Calling 911

No matter how good a developer you are, there are times when you:

- get stuck with a problem you can't solve
- encounter a bug and wish to report it and/or find a workaround
- need assistance with your design.

In this chapter, we'll look at the resources available when you face these problems.

So you've got a problem...

We'll talk about the first two problems together, because it's often hard to tell which problem you're actually experiencing.

Something no longer works, or doesn't work as expected. What should you do about it?

Read the manuals

While this may seem like an obvious first step, it's *amazing* the number of people who don't do this!

All the manuals for the QNX Neutrino operating system are online:

Building Embedded Systems

This guide contains all the information you'll need to “embed” QNX Neutrino—that is, to get a QNX Neutrino system up and running. It has chapters on the development environment (how to compile, link, and debug a QNX Neutrino program), building images (how to get a system image created, how to embed this image into a deeply embedded system, how to get it “running” on a supported platform), and some design notes.

C Library Reference

The “A through Z” of the C library—use this to find information about each and every function call that's provided by QNX Neutrino's C library. This is the ultimate “authority” on function calls. Often in this book, I've referred you to this library (for example, to find out more about a particular function, such as arguments that aren't commonly used).

System Architecture

A “top-level” architecture document, the *System Architecture* guide describes the QNX Neutrino system from a high-level view, giving enough details about the implementation that you can get a good idea of what the pieces are and how they all fit together.

Technotes

The technotes describe special features of QNX Neutrino and may vary from release to release. Take a look at the online version to see what's in the release you currently have.

User's Guide

This guide is intended for *all* users of a QNX Neutrino system, from system administrators to end users. This guide tells you how to:

- Use the QNX Neutrino *runtime* environment, regardless of the kind of computer it's running on (embedded system or desktop). Think of this guide as the companion how-to doc for the *Utilities Reference*. Assuming there's a QNX Neutrino system prompt waiting for input, this guide is intended to help you learn how to interact with that prompt.
- Perform such traditional system administration topics as setting up user accounts, security, starting up a QNX Neutrino machine, etc.

Utilities Reference

This guide is the “A through Z” of the command-line utilities available. It covers all command-line utilities such as `grep`, `make`, `ls`, etc.

Audio Developer's Guide

This guide describes how to develop audio applications.

© 2020, QNX Software Systems Limited

Programmer's Guide

The QNX Neutrino *Programmer's Guide* and this book both describe how to develop applications and resource managers, but from somewhat different perspectives.

Contact technical support

Once you've determined to the best of your abilities that the problem isn't some misunderstanding of the function call or utility you're using, or a mere typo, you may enter the realm of the QNX technical support department. This is nothing to be afraid of—most customers are extremely pleased with the level of technical support they get from QNX Software Systems.

There are two ways of contacting the QNX technical support group: by phone or via the web. The QNX website (at *http://www.qnx.com*), has a Community area called Foundry27, at *http://community.qnx.com/sf/sfmain/do/home*, that's full of useful information and files.

Before we talk about which method you should use, there are a few things you can do to make the turnaround time on your bug report *much* shorter.

Describe the problem

Often customers try to fix the problems themselves by trying various things that come to mind. This is great.

Unfortunately, what tends to happen is that customers get frustrated and post messages something like:

```
I just ran the TCP/IP package connected to a Windows box
and it doesn't work.

What's going on?!?
```

The very next message from tech support looks like the following (I think they should have a standard template for it, myself):

```
Can you describe what you mean by "doesn't work"?  Do you mean
the TCP/IP on the QNX side?  Do you mean the TCP/IP on the
Windows box?  What part of TCP/IP doesn't work?  What are you
trying to do?  What versions of the OS, and TCP/IP package do
you have?  What version of Windows?  What TCP/IP package
were you using there?
```

The moral of the story: if you're having a problem, then you're probably in a hurry for the answer. If you're in a hurry for the answer, supply as much information as possible in your initial post so that someone at QNX Software Systems can try right away to reproduce the problem.

Here are the things that tech support almost always asks for:

- precise descriptions of failure
- versions
- configuration
- platform (x86, ARM, etc.)

© 2020, QNX Software Systems Limited

Precise information

To supply this information, state what you had expected to happen, and what actually happened.

In our above example, a much better problem description would have been:

```
I just ran telnet from Neutrino 2.0, patch level "A", to my
Windows box, and, immediately after the login prompt, got a
"Connection closed by foreign host".
```

Versions

The next thing that you should supply is the versions of the various commands that you may have been using. This can be done by using the `ls` and `cksum` commands. For our example above, you'll want to tell tech support which version of the `telnet` command you were using, and the version of the TCP/IP protocol stack etc.

```
# ls -l /usr/bin/telnet /lib/dll/devnp-e1000.so
-rwxrwxr-x  1 root      bin           64220 Jun 22 05:36 /usr/bin/telnet
-rwxrwxr-x  1 root      bin           27428 Jun 22 03:29 /lib/dll/devnp-e1000.so

# cksum /usr/bin/telnet /lib/dll/devnp-e1000.so
1217616014      64220 /usr/bin/telnet
  50089252      27428 /lib/dll/devnp-e1000.so
```

This gives tech support at least some idea of the dates, sizes, and checksums of some of the products that might be involved in the problem.

If you suspect your problem might be related to a platform-specific interaction, you should of course specify the name, brand, and relevant chipsets used on that particular platform.

Another thing that tech support usually requests, especially if they suspect some problems with insufficient memory, licensing, configuration, etc., is the runtime configuration of your system. You should try to give them an idea of how much memory is installed, how many processes are running, what the approximate load on the system might be, etc.

The more information you have, the faster they can help you.

If you're using a beta…

If you're using a beta version of the product (i.e., you're on QNX Software Systems' list of beta sites), all the above information is critical, because you'll typically be using different versions of the software than what is released.

Note, however, that the technical support department generally doesn't handle telephone support of beta products. The only way to get help on these is to post in the conference or, if the developer has requested direct contact, talk to the developer. Posting is generally the best solution anyway, because then other members of the beta conference can see what problems are out there and can learn what the solution is (i.e., if it's a bug, what the workaround for it is). In any event, the above information is crucial in order to determine which products you have from the beta release and which ones are stock.

Also, keep in mind that if you're talking with a developer, they often have a million things on their plates and might not be able to get back to you right away. Sending a friendly “ping” reminder after a few days doesn't hurt. Sending a demanding one after 15 minutes will not gain you any new friends!

An issue that frequently comes up with betas is that you may forget to install an update. Due to the way that the beta process works, missing an update may cause strange behavior on your system. Certain new drivers or resource managers may behave differently towards their respective client programs than they did in previous versions.

In this case, you should ensure (because the support staff will ask!) that you have indeed installed all the beta updates in the order listed.

Reproduce the problem

One of the first things that tech support usually wants to know is, Does it happen just once in a blue moon, or can you make it happen on demand?

They don't ask this question idly. If it's a problem that happens infrequently, it's just as serious as a problem that happens regularly. The point is to try to determine how to proceed.

Generally, for problems that happen infrequently, the support staff will recommend that you configure the machine with the operating system and components set up in such a way that when the problem happens again, some form of log will be left around or perhaps the debugger will be invoked so that the problem can be diagnosed later.

For a problem that's easily reproducible, they'll want to reproduce it at QNX Software Systems so that they can show the developer on a live system. “Hey, look! It dies when I …”

Narrow it down

Even if it's reproducible, tech support most likely doesn't want to see 6000 lines of C code with a problem buried in the middle of it.

In most cases that I've witnessed, a bug can usually be narrowed down to about 20 to 30 lines of C *at the most*. The only cases where a really large file is actually useful is when reporting bugs with something where you suspect it's a size problem, rather than a library or kernel problem. For example, some utilities may have a default array size that may cause trouble when it needs to resize that array for something bigger. In this case, tech support may ask you for a `tar` file with *everything* in it. Luckily, `tar` files are easy to create. For example, if you're developing your product in **/src/projects/xyzzy** and they want to see everything in that directory, you can perform the following steps:

```
# cd /src/projects
# tar cvf xyzzy.tar xyzzy
```

This will “suck” everything out of the **xyzzy** directory (and *all* subdirectories too!) into the file called **xyzzy.tar**. If this resulting `tar` file is huge, you can save some download time and disk space by compressing it with `gzip`:

```
# gzip -9v xyzzy.tar
xyzzy.tar:       60.2% -- replaced with xyzzy.tar.gz
```

You'd then send the support people the **xyzzy.tar.gz** file (generally by `ftp` rather than as an email attachment `:-)`).

© 2020, QNX Software Systems Limited

Training

Finally, several companies offer training courses for QNX products, and QNX Software Systems offers onsite as well as periodic training at their facility.

© 2020, QNX Software Systems Limited

Appendix C
Sample Programs

This appendix contains the complete versions of some of the sample programs discussed in this book.

atoz.c

For more information about this program, see the example in the "Returning directory entries" section of the Resource Managers chapter.

```
/*
 *  atoz.c
 *
 *  /dev/atoz using the resource manager library
*/

#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <errno.h>
#include <dirent.h>
#include <limits.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>

#define ALIGN(x) (((x) + 3) & ~3)
#define NUM_ENTS            26

static  iofunc_attr_t   atoz_attrs [NUM_ENTS];

static int
my_open (resmgr_context_t *ctp, io_open_t *msg, iofunc_attr_t *attr, void *extra)
{
    if (msg -> connect.path [0] == 0)
    {   // the directory (/dev/atoz)
        return (iofunc_open_default (ctp, msg, attr, extra));
    } else if (msg -> connect.path [1] == 0 &&
              (msg -> connect.path [0] >= 'a' &&
               msg -> connect.path [0] <= 'z'))
    {   // the file (/dev/atoz/[a-z])
        return (iofunc_open_default (ctp, msg,
                   atoz_attrs + msg -> connect.path [0] - 'a', extra));
    } else {
        return (ENOENT);
    }
}

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

struct dirent *
dirent_fill (struct dirent *dp, int inode, int offset, char *fname)
{
    dp -> d_ino = inode;
    dp -> d_offset = offset;
```


© 2020, QNX Software Systems Limited

```
    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));
}

static int
my_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 [_POSIX_PATH_MAX];

    // allocate a buffer for the reply
    reply_msg = calloc (1, msg -> i.nbytes);
    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 < NUM_ENTS) {

        // create the filename
        sprintf (fname, "%c", ocb -> offset + 'a');

        // 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;
        }
    }

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

```
                (char *) dp - reply_msg);

    // release our buffer
    free (reply_msg);

    // tell resource manager library we already did the reply
    return (_RESMGR_NOREPLY);
}

static int
my_read_file (resmgr_context_t *ctp, io_read_t *msg, iofunc_ocb_t *ocb)
{
    int     nbytes;
    int     nleft;
    char    string;

    // 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 -> nbytes - ocb -> offset;

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

    if (nbytes) {
        // create the output string
        string = ocb -> attr -> inode - 1 + 'A';

        // return it to the client
        MsgReply (ctp -> rcvid, nbytes, &string + ocb -> offset, nbytes);

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

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

static int
my_read (resmgr_context_t *ctp, io_read_t *msg, iofunc_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);
```


© 2020, QNX Software Systems Limited

```
    }

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

int
main (int argc, char **argv)
{
    dispatch_t              *dpp;
    resmgr_attr_t           resmgr_attr;
    dispatch_context_t      *ctp;
    resmgr_connect_funcs_t  connect_func;
    resmgr_io_funcs_t       io_func;
    iofunc_attr_t           attr;
    int                     i;

    // create the dispatch structure
    if ((dpp = dispatch_create ()) == NULL) {
        perror ("Unable to dispatch_create");
        exit (EXIT_FAILURE);
    }

    // initialize the various data structures
    memset (&resmgr_attr, 0, sizeof (resmgr_attr));
    resmgr_attr.nparts_max = 1;
    resmgr_attr.msg_max_size = 2048;

    // bind default functions into the outcall tables
    iofunc_func_init (_RESMGR_CONNECT_NFUNCS, &connect_func,
                      _RESMGR_IO_NFUNCS, &io_func);

    // create and initialize the attributes structure for the directory
    iofunc_attr_init (&attr, S_IFDIR | 0555, 0, 0);
    attr.inode = NUM_ENTS + 1;  // 1-26 are reserved for 'a' through 'z' files
    attr.nbytes = NUM_ENTS;     // 26 entries contained in this directory

    // and for the "a" through "z" names
    for (i = 0; i < NUM_ENTS; i++) {
        iofunc_attr_init (&atoz_attrs [i], S_IFREG | 0444, 0, 0);
        atoz_attrs [i].inode = i + 1;
        atoz_attrs [i].nbytes = 1;
    }

    // add our functions; we're only interested in the open connect
    // function and read I/O function handlers
    connect_func.open = my_open;
    io_func.read = my_read;
```

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

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

    // wait here forever, handling messages
    while (1) {
        if ((ctp = dispatch_block (ctp)) == NULL) {
            perror ("Unable to dispatch_block");
            exit (EXIT_FAILURE);
        }
        dispatch_handler (ctp);
    }

    // you'll never get here
    return (EXIT_SUCCESS);
}
```


© 2020, QNX Software Systems Limited

time1.c

For more information about this program, see “Server-maintained timeouts” in the Clocks, Timers, and Getting a Kick Every So Often chapter.

```
/*
 *  time1.c
 *
 *  Example of a server that receives periodic messages from
 *  a timer, and regular messages from a client.
 *
 *  Illustrates using the timer functions with a pulse.
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <signal.h>
#include <errno.h>
#include <unistd.h>
#include <sys/siginfo.h>
#include <sys/neutrino.h>

// message send definitions

// messages
#define MT_WAIT_DATA        2       // message from client
#define MT_SEND_DATA        3       // message from client

// pulses
#define CODE_TIMER          1       // pulse from timer

// message reply definitions
#define MT_OK               0       // message to client
#define MT_TIMEDOUT         1       // message to client

// message structure
typedef struct
{
    int messageType;                // contains both message to and from client
    int messageData;                // optional data, depending upon message
} ClientMessageT;

typedef union
{
    ClientMessageT  msg;            // a message can be either from a client, or
    struct _pulse   pulse;          // a pulse
} MessageT;

// client table
#define MAX_CLIENT 16               // maximum number of simultaneous clients

struct
```

```
{
    int in_use;                     // is this client entry in use?
    int rcvid;                      // receive ID of client
    int timeout;                    // timeout left for client
}   clients [MAX_CLIENT];           // client table

int     chid;                       // channel ID (global)
int     debug = 1;                  // set debug value, 1 == enabled, 0 == off
char    *progname = "time1.c";

// forward prototypes
static  void setupPulseAndTimer (void);
static  void gotAPulse (void);
static  void gotAMessage (int rcvid, ClientMessageT *msg);

int
main (void)                         // ignore command-line arguments
{
    int rcvid;                      // process ID of the sender
    MessageT msg;                   // the message itself

    if ((chid = ChannelCreate (0)) == -1) {
        fprintf (stderr, "%s:  couldn't create channel!\n", progname);
        perror (NULL);
        exit (EXIT_FAILURE);
    }

    // set up the pulse and timer
    setupPulseAndTimer ();

    // receive messages
    for (;;) {
        rcvid = MsgReceive (chid, &msg, sizeof (msg), NULL);

        // determine who the message came from
        if (rcvid == 0) {
            // production code should check "code" field...
            gotAPulse ();
        } else {
            gotAMessage (rcvid, &msg.msg);
        }
    }

    // you'll never get here
    return (EXIT_SUCCESS);
}

/*
 *  setupPulseAndTimer
 *
 *  This routine is responsible for setting up a pulse so it
 *  sends a message with code MT_TIMER.  It then sets up a periodic
 *  timer that fires once per second.
*/
```


© 2020, QNX Software Systems Limited

```
void
setupPulseAndTimer (void)
{
    timer_t             timerid;    // timer ID for timer
    struct sigevent     event;      // event to deliver
    struct itimerspec   timer;      // the timer data structure
    int                 coid;       // connection back to ourselves

    // create a connection back to ourselves
    coid = ConnectAttach (0, 0, chid, 0, 0);
    if (coid == -1) {
        fprintf (stderr, "%s:  couldn't ConnectAttach to self!\n", progname);
        perror (NULL);
        exit (EXIT_FAILURE);
    }

    // set up the kind of event that we want to deliver -- a pulse
    SIGEV_PULSE_INIT (&event, coid, SIGEV_PULSE_PRIO_INHERIT, CODE_TIMER, 0);

    // create the timer, binding it to the event
    if (timer_create (CLOCK_REALTIME, &event, &timerid) == -1) {
        fprintf (stderr, "%s:  couldn't create a timer, errno %d\n",
                 progname, errno);
        perror (NULL);
        exit (EXIT_FAILURE);
    }

    // setup the timer (1s delay, 1s reload)
    timer.it_value.tv_sec = 1;
    timer.it_value.tv_nsec = 0;
    timer.it_interval.tv_sec = 1;
    timer.it_interval.tv_nsec = 0;

    // and start it!
    timer_settime (timerid, 0, &timer, NULL);
}

/*
 *  gotAPulse
 *
 *  This routine is responsible for handling the fact that a timeout
 *  has occurred.  It runs through the list of clients to see
 *  which client has timed-out, and replies to it with a timed-out
 *  response.
 */

void
gotAPulse (void)
{
    ClientMessageT  msg;
    int             i;

    if (debug) {
```

```
        time_t  now;

        time (&now);
        printf ("Got a Pulse at %s", ctime (&now));
    }

    // prepare a response message
    msg.messageType = MT_TIMEDOUT;

    // walk down list of clients
    for (i = 0; i < MAX_CLIENT; i++) {

        // is this entry in use?
        if (clients [i].in_use) {

            // is it about to time out?
            if (--clients [i].timeout == 0) {

                // send a reply
                MsgReply (clients [i].rcvid, EOK, &msg, sizeof (msg));

                // entry no longer used
                clients [i].in_use = 0;
            }
        }
    }
}

/*
 *  gotAMessage
 *
 *  This routine is called whenever a message arrives.  We look at the
 *  type of message (either a "wait for data" message, or a "here's some
 *  data" message), and act accordingly.  For simplicity, we'll assume
 *  that there is never any data waiting.  See the text for more discussion
 *  about this.
*/

void
gotAMessage (int rcvid, ClientMessageT *msg)
{
    int i;

    // determine the kind of message that it is
    switch (msg -> messageType) {

    // client wants to wait for data
    case    MT_WAIT_DATA:

        // see if we can find a blank spot in the client table
        for (i = 0; i < MAX_CLIENT; i++) {

            if (!clients [i].in_use) {
```


© 2020, QNX Software Systems Limited

```
                // found one -- mark as in use, save rcvid, set timeout
                clients [i].in_use = 1;
                clients [i].rcvid = rcvid;
                clients [i].timeout = 5;
                return;
            }
        }

        fprintf (stderr, "Table full, message from rcvid %d ignored, "
                         "client blocked\n", rcvid);
        break;

    // client with data
    case    MT_SEND_DATA:

        // see if we can find another client to reply to with this
        // client's data
        for (i = 0; i < MAX_CLIENT; i++) {

            if (clients [i].in_use) {

                // found one -- reuse the incoming message as an
                // outgoing message
                msg -> messageType = MT_OK;

                // reply to BOTH CLIENTS!
                MsgReply (clients [i].rcvid, EOK, msg, sizeof (*msg));
                MsgReply (rcvid, EOK, msg, sizeof (*msg));

                clients [i].in_use = 0;
                return;
            }
        }

        fprintf (stderr, "Table empty, message from rcvid %d ignored, "
                         "client blocked\n", rcvid);
        break;
    }
}
```

tp1.c

For more information about this program, see “Controlling the number of threads” in the Processes and Threads chapter.

```
/*
 * tp1.c
 *
 * Thread Pool Example (1)
 *
 * 1999 06 26 R. Krten
*/

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/neutrino.h>
#include <sys/dispatch.h>

char *progname = "tp1";

void
tag (char *name)
{
 time_t t;
 char buffer [BUFSIZ];

 time (&t);
 strftime (buffer, BUFSIZ, "%T ", localtime (&t));
 printf ("%s %3d %-20.20s:  ", buffer, pthread_self (), name);
}

THREAD_POOL_PARAM_T *
blockfunc (THREAD_POOL_PARAM_T *ctp)
{
 tag ("blockfunc"); printf ("ctp %p\n", ctp);
 tag ("blockfunc"); printf ("sleep (%d);\n", 15 * pthread_self ());
 sleep (pthread_self () * 15);
 tag ("blockfunc"); printf ("done sleep\n");
 tag ("blockfunc"); printf ("returning 0x%08X\n", 0x10000000 + pthread_self ());
 return ((void *) (0x10000000 + pthread_self ()));  // passed to handlerfunc
}

THREAD_POOL_PARAM_T *
contextalloc (THREAD_POOL_HANDLE_T *handle)
{
 tag ("contextalloc"); printf ("handle %p\n", handle);
 tag ("contextalloc"); printf ("returning 0x%08X\n", 0x20000000 + pthread_self ());
 return ((void *) (0x20000000 + pthread_self ()));  // passed to blockfunc
}

void
contextfree (THREAD_POOL_PARAM_T *param)
```


© 2020, QNX Software Systems Limited

```
{
 tag ("contextfree"); printf ("param %p\n", param);
}

void
unblockfunc (THREAD_POOL_PARAM_T *ctp)
{
 tag ("unblockfunc"); printf ("ctp %p\n", ctp);
}

int
handlerfunc (THREAD_POOL_PARAM_T *ctp)
{
 static int i = 0;

 tag ("handlerfunc"); printf ("ctp %p\n", ctp);
 if (i++ > 15) {
  tag ("handlerfunc"); printf ("exceeded 15 operations, return 0\n");
  return (0);
 }
 tag ("handlerfunc"); printf ("sleep (%d)\n", pthread_self () * 25);
 sleep (pthread_self () * 25);
 tag ("handlerfunc"); printf ("done sleep\n");

/*
 i = 0;
 if (i++ & 1) {
  tag ("handlerfunc"); printf ("returning 0\n");
  return (0);
 } else {
*/
  tag ("handlerfunc"); printf ("returning 0x%08X\n", 0x30000000 + pthread_self ());
  return (0x30000000 + pthread_self ());
/*
 }
*/
}


main ()
{
 thread_pool_attr_t tp_attr;
 void    *tpp;

 memset (&tp_attr, 0, sizeof (tp_attr));
 tp_attr.handle = (void *) 0x12345678;  // passed to contextalloc
 tp_attr.block_func = blockfunc;
 tp_attr.unblock_func = unblockfunc;
 tp_attr.context_alloc = contextalloc;
 tp_attr.context_free = contextfree;
 tp_attr.handler_func = handlerfunc;

 tp_attr.lo_water = 3;
 tp_attr.hi_water = 7;
```

```
 tp_attr.increment = 2;
 tp_attr.maximum = 10;

    tpp = thread_pool_create (&tp_attr, POOL_FLAG_USE_SELF);
    if (tpp == NULL) {
        fprintf (stderr,
                 "%s:  can't thread_pool_create, errno %s\n",
                 progname, strerror (errno));
        exit (EXIT_FAILURE);
    }

 thread_pool_start (tpp);

 fprintf (stderr, "%s:  thread_pool_start returned; errno %s\n",
             progname, strerror (errno));
 sleep (3000);
 exit (EXIT_FAILURE);
}
```


© 2020, QNX Software Systems Limited

tt1.c

For more information about this program, see “Kernel timeouts with pthread_join()” in the Clocks, Timers, and Getting a Kick Every So Often chapter.

```
/*
 * tt1.c
*/

#include <stdio.h>
#include <pthread.h>
#include <inttypes.h>
#include <errno.h>
#include <sys/neutrino.h>

#define SEC_NSEC 1000000000LL // 1 billion nanoseconds in a second

void *
long_thread (void *notused)
{
    printf ("This thread runs for more than 10 seconds\n");
    sleep (20);
}

int
main (void) // ignore arguments
{
    uint64_t        timeout;
    struct sigevent event;
    int             rval;
    pthread_t       thread_id;

    // set up the event -- this can be done once

    // This or event.sigev_notify = SIGEV_UNBLOCK:
    SIGEV_UNBLOCK_INIT (&event);

    // create a thread
    pthread_create (&thread_id, NULL, long_thread, NULL);

    // set up for 10 second timeout
    timeout = 10LL * SEC_NSEC;

    TimerTimeout (CLOCK_MONOTONIC, _NTO_TIMEOUT_JOIN, &event, &timeout, NULL);

    rval = pthread_join (thread_id, NULL);
    if (rval == ETIMEDOUT) {
        printf ("Thread %d is still running after 10 seconds!\n",
                thread_id);
    }

    sleep (5);
```

```
    TimerTimeout (CLOCK_MONOTONIC, _NTO_TIMEOUT_JOIN, &event, &timeout, NULL);
    rval = pthread_join (thread_id, NULL);
    if (rval == ETIMEDOUT) {
        printf ("Thread %d is still running after 25 seconds (bad)!\n",
                thread_id);
    } else {
        printf ("Thread %d finished (expected!)\n", thread_id);
    }
}
```


© 2020, QNX Software Systems Limited

Appendix D
Glossary

absolute timer

A timer with an expiration point defined as a fixed time, for example, January 20, 2005 at 09:43:12 AM, EDT. Contrast with *relative timer*.

alignment

The characteristic that accessing an *N*-byte data element must be performed only on an address that is a multiple of *N*. For example, to access a 4-byte integer, the address of the integer must be a multiple of 4 bytes (e.g., 0x2304B008, and not 0x2304B009). On some CPU architectures, an alignment fault will occur if an attempt is made to perform a non-aligned access. On other CPU architectures (e.g., x86) a non-aligned access is simply slower than an aligned access.

asynchronous

Used to indicate that a given operation is not synchronized to another operation. For example, the timer tick interrupt that is generated by the system's timer chip is said to be "asynchronous" to a thread that's requesting a delay of a certain amount of time, because the thread's request is not synchronized in any way to the arrival of the incoming timer tick interrupt. Contrast with *synchronous*.

atomic (operation)

An operation that is "indivisible," that is to say, one that will not get interrupted by any other operation. Atomic operations are critical especially in interrupt service routines and multithreaded programs, as often a "test and set" sequence of events must occur in one thread without the chance of another thread interrupting this sequence. A sequence can be made atomic from the perspective of multiple threads not interfering with each other through the use of *mutexes* or via *InterruptLock()* and *InterruptUnlock()* when used with *Interrupt service routines*. See the header file **<atomic.h>** as well.

attribute (structure)

A structure used within a *resource manager* that contains information relating to the device that the resource manager is manifesting in the pathname space. If the resource manager is manifesting multiple devices in the pathname space (for example, the serial port resource manager might manifest **/dev/ser1** and **/dev/ser2**) there will be an equal number of attribute structures in the resource manager. Contrast with *OCB*.

barrier (synchronization object)

A thread-level synchronization object with an associated count. Threads that call the blocking barrier call (*pthread_barrier_wait()*) will block until the number of threads specified by the count have all called the blocking barrier call, and then they will all be released. Contrast this with the operation of *semaphores*.

blocking

A means for threads to synchronize to 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 event occurs that the thread was waiting for, the thread is *unblocked* and is able to consume CPU again.

channel

An abstract object on which a *server receives a message*. This is the same object to which a *client* creates a *connection* in order to *send a message* to the server. When the channel is created via *ChannelCreate()*, a "channel ID" is returned. This channel ID (or "chid" for short) is what a *resource manager* will advertise as part of its registered mountpoint.

client

QNX Neutrino's *message-passing* architecture is structured around a client/*server* relationship. In the case of the client, it's the one that is requesting services of a particular 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.

condition variable

A synchronization object used between multiple *threads*, characterized by acting as a rendezvous point where multiple threads can *block*, waiting for a signal (not to be confused with a UNIX-style *signal*). When the signal is delivered, one or more of the threads will *unblock*.

connection

The concept of a *client* being attached to a *channel*. A connection is established by the client either directly by calling *ConnectAttach()* or on behalf of the client by the client's C library function *open()*. In either case, the *connection ID* returned is usable as a handle for all communications between the client and the *server*.

connection ID

A "handle" returned by *ConnectAttach()* (on the *client* side) and used for all communications between the client and the *server*. The connection ID is *identical* to the traditional C library's "file descriptor." That is to say, when *open()* returns a file descriptor, it's really returning a connection ID.

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* (see below). (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.)

© 2020, QNX Software Systems Limited

FIFO (scheduling)

In FIFO scheduling, a *thread* will consume CPU until a higher priority thread is ready to run, or until the thread voluntarily *gives up CPU*. If there are no higher priority threads, and the thread does not voluntarily give up CPU, it will run forever. Contrast with *round robin* scheduling.

interrupt service routine

Code that gets executed (in privileged mode) by the kernel as a result of a hardware interrupt. This code cannot perform any kernel calls and should return as soon as possible, since it runs at a priority level effectively higher than any other thread priority in the system. QNX Neutrino's interrupt service routines can return a `struct sigevent` that indicates what event, if any, should be triggered.

IOV (I/O Vector)

A structure where each member contains a pointer and a length. Generally used as an array of IOVs, rather than as a single IOV. When used in the array form, this array of structures of pointers and lengths defines a *scatter/gather* list, which allows the *message-passing* operations to proceed much more efficiently (than would otherwise be accomplished by copying data individually so as to form one contiguous buffer).

kernel callouts

The QNX Neutrino operating system can be customized to run on various hardware, without requiring a source license, by supplying kernel callouts to the startup program. Kernel callouts let the developer supply code that knows how to deal with the specifics of the hardware. For example, how to ask an interrupt controller chip about which interrupt fired, or how to interface to the timer chip to be able to arrange for periodic interrupts, etc. This is documented in great depth in the *Building Embedded Systems* book.

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.

MMU (Memory Management Unit)

A piece of hardware (usually embedded within the CPU) that provides for *virtual address* to *physical address* translation, and can be used to implement a *virtual memory* system. Under QNX Neutrino, the primary benefit of an MMU is the ability to detect when a *thread* has accessed a virtual address that is not mapped into the *process's* address space.

mutex

A *Mut*ual *Ex*clusion object used to serialize a number of threads so that only one thread at a time has access to the resources defined by the mutex. By using a mutex every time (for example) that you access a given variable, you're ensuring that only one thread at a time has access to that variable, preventing race conditions. See also *atomic (operation)*.

Neutrino

Quoting from the Sudbury Neutrino Observatory web pages (found at *http://www.sno.phy.queensu.ca/*):

> Neutrinos are tiny, possibly massless, neutral elementary particles which interact with matter via the weak nuclear force. The weakness of the weak force gives neutrinos the property that matter is almost transparent to them. The sun, and all other stars, produce neutrinos copiously due to nuclear fusion and decay processes within the core. Since they rarely interact, these neutrinos pass through the sun and the earth (and you) unhindered. Other sources of neutrinos include exploding stars (supernovae), relic neutrinos (from the birth of the universe) and nuclear power plants (in fact a lot of the fuel's energy is taken away by neutrinos). For example, the sun produces over two hundred trillion trillion trillion neutrinos every second, and a supernova blast can unleash 1000 times more neutrinos than our sun will produce in its 10-billion year lifetime. Billions of neutrinos stream through your body every second, yet only one or two of the higher energy neutrinos will scatter from you in your lifetime.

OCB (open context block)

A data structure used by a *resource manager* that contains information for each *client's open()* call. If a client has opened several files, there will exist a corresponding OCB for each file descriptor that the client has in the respective resource managers. Contrast with the *attribute (structure)*.

PDP-8

An antique computer, "Programmable Data Processor," manufactured between 1965 and the mid 1970's by Digital Equipment Corporation (now Compaq) with the *coolest* front panel. Also, the first computer I ever programmed. Unfortunately, this wonderful 12-bit machine does *not* run QNX Neutrino `:-(`!

periodic timer

See *Repeating timer*

physical address

An address that is emitted by the CPU onto the bus connected to the memory subsystem. Since QNX Neutrino runs in *virtual address* mode, this means that an *MMU* must translate

© 2020, QNX Software Systems Limited

the virtual addresses used by the *threads* into physical addresses usable by the memory subsystem. Contrast with *virtual address* and *virtual memory*.

process

A non-schedulable entity that occupies memory, effectively acting as a container for one or more *threads*.

pthreads

Common name given to the set of function calls of the general form *pthread_*()*. The vast majority of these function calls are defined by the POSIX committee, and are used with *threads*.

pulse

A non-blocking message which is *received* in a manner similar to a *regular message*. It is non-blocking for the sender, and can be waited upon 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* (breaking which would cause *deadlock*). Contrast with *signal*.

QNX Software Systems

The company responsible for the QNX family of operating systems.

QSS

An abbreviation for *QNX Software Systems*.

receive a message

A thread can receive a message by calling *MsgReceive()* or *MsgReceivev()*. If there is no message available, the thread will *block*, waiting for one. See *Message passing*. A thread that receives a message is said to be a *server*.

receive ID

When a *server receives a message* from a *client*, the server's *MsgReceive()* or *MsgReceivev()* function returns a “receive ID” (often abbreviated in code as *rcvid*). This *rcvid* then acts as a handle to the *blocked* client, allowing the server to *reply with the data* back to the client, effectively unblocking the client. Once the *rcvid* has been used in a reply operation, the *rcvid* ceases to have any meaning for all function calls, except *MsgDeliverEvent()*.

relative timer

A timer that has an expiration point defined as an offset from the current time, for example, 5 minutes from now. Contrast with *absolute timer*.

repeating timer

An *absolute* or *relative* timer that, once expired, will automatically reload with another relative interval and will keep doing that until it is canceled. Useful for receiving periodic notifications.

reply to a message

A *server* will reply to a *client's* message in order to deliver the results of the client's request back to the client.

resource manager

Also abbreviated "resmgr." This is a *server* process which provides certain well-defined file-descriptor-based services to arbitrary *clients*. A resource manager supports a limited set of messages, which correspond to standard client C library functions such as *open()*, *read()*, *write()*, *lseek()*, *devctl()*, etc.

round robin (scheduling)

In Round Robin (or "RR") scheduling, a *thread* will consume CPU until a higher priority thread is ready to run, until the thread voluntarily *gives up CPU*, or until the thread's timeslice expires. If there are no higher priority threads, the thread doesn't voluntarily give up CPU, and there are no other threads at the same priority, it will run forever. If all the above conditions are met except that a thread at the same priority is ready to run, then this thread will give up CPU after its timeslice expires, and the other thread will be given a chance to run. Contrast with *FIFO* scheduling.

scatter/gather

Used to define the operation of *message passing* where a number of different pieces of data are "gathered" by the kernel (on either the *client* or *server* side) and then "scattered" into a (possibly) different number of pieces of data on the other side. This is extremely useful when, for example, a header needs to be prepended to the client's data before it's sent to the server. The client would set up an *IOV* which would contain a pointer and length of the header as the first element, and a pointer and length of the data as the second element. The kernel would then "gather" this data as if it were one contiguous piece and *send* it to the server. The server would operate analogously.

semaphore

A *thread* synchronization primitive characterized by having a count associated with it. Threads may call the *sem_wait()* function and not *block* if the count was nonzero at the time of the call. Every thread that calls *sem_wait()* decrements the count. If a thread calls *sem_wait()* when the count is zero, the thread will block until some other thread calls *sem_post()* to increment the count. Contrast with *barrier*.

send a message

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*. See *Message passing*. A thread that sends a message is said to be a *Client*.

send hierarchy

A design paradigm whereby *messages sent* flow in one direction, and *messages replied to* flow in another 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

© 2020, QNX Software Systems Limited

hierarchy—one thread should not have sent to the other thread, as that other thread must have been at a lower level.

server

A server is a regular, user-level *process* that provides certain types of functionality (usually file-descriptor-based) to *clients*. Servers are typically *Resource Managers*, and there's an extensive library provided by QNX Software Systems 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* can be both a client and a server at the same time.

signal

A mechanism dating back to early UNIX systems that is used to send *asynchronous* notification of events from one *thread* to another. Signals are non-blocking for the sender. The receiver of the signal may decide to treat the signal in a *synchronous* manner by explicitly waiting for it. Contrast with *pulse*.

sporadic

A scheduling policy whereby a thread's priority can oscillate dynamically between a “foreground” or normal priority and a “background” or low priority. A thread is given an execution *budget* of time to be consumed within a certain *replenishment* period. See also *FIFO* and *round robin*.

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*), the unblocking of the client is said to be synchronous to the reply operation. Contrast with *Asynchronous*.

thread

A single, schedulable, flow of execution. Threads are implemented directly within the microkernel and correspond to 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 *FIFO*, *Round Robin*, or *sporadic* scheduling mode.

unblock

A thread that had been *blocked* will be unblocked when the condition it has been blocked on is met. For example, a thread might be blocked waiting to *receive a message*. When the *message is sent*, the thread will be unblocked.

virtual address

An address that's not necessarily equivalent to a *physical address*. Under QNX Neutrino, all *threads* operate in virtual addressing mode, where, through the magic of an *MMU*, the virtual addresses are translated into physical addresses. Contrast with *physical address* and *virtual memory*.

virtual memory

A “virtual memory” system is one in which the *virtual address* space may not necessarily map on a one-to-one basis with the *physical address* space. The typical example (which QNX Neutrino doesn't support as of this writing) is a “paged” system where, in the case of a lack of RAM, certain parts of a *process's* address space may be swapped out to disk. What QNX Neutrino does support is the dynamic mapping of stack pages.

© 2020, QNX Software Systems Limited

Index

_FTYPE_ANY *234*
_FTYPE_MQUEUE *234*
_IO_ACL *253*
_IO_CHMOD *255*
_IO_CHOWN *256*
_IO_CLOSE *259*, *328*
_IO_CONNECT *261*, *273*, *277*, *279*, *285*, *287*, *299*, *328*, *353*
_IO_CONNECT_COMBINE *279*
_IO_CONNECT_COMBINE_CLOSE *238*, *279*
_IO_CONNECT_LINK *261*
_IO_CONNECT_MKNOD *273*
_IO_CONNECT_MOUNT *277*
_IO_CONNECT_OPEN *279*
_IO_CONNECT_READLINK *285*
_IO_CONNECT_RENAME *287*
_IO_CONNECT_UNLINK *299*
_IO_DEVCTL *226*, *263*, *317*
_IO_DUP *267*
_IO_FDINFO *268*
_IO_FLAG_RD *241*
_IO_FLAG_WR *241*
_IO_LSEEK *293–294*, *328*
_IO_LSEEK_IGNORE_NON_SEEKABLE *294*
_IO_MAX *118*
_IO_MMAP *275*
_IO_MSG *226*, *302*, *348–350*, *352–353*
_IO_NOTIFY *289*
_IO_NOTIFY64 *289*
_IO_OPEN *246*
_IO_OPENFD *281*
_IO_PATHCONF *282*
_IO_READ *283–284*, *308*, *328*
_IO_READ_GET_NBYTES() 284
_IO_READ64 *283*
_IO_SET_CONNECT_RET() 278, *280*
_IO_SET_FDINFO_LEN() 269
_IO_SET_PATHCONF_VALUE() 282
_IO_SET_READ_NBYTES() 284
_IO_SET_WRITE_NBYTES() 304, *315–316*
_IO_SPACE *292*
_IO_STAT *295*
_IO_SYNC *271*
_IO_UTIME *257*
_IO_UTIME64 *257*
_IO_WRITE *303*, *313*
_IO_WRITE_GET_NBYTES() 304
_IO_WRITE64 *303*
_IO_XTYPE_NONE *311*
_IO_XTYPE_OFFSET *284*, *311–313*, *316*
_IO_XTYPE_READDIR *284*
_IOFUNC_NFUNCS *247*
_IOMGR_PRIVATE_BASE *353*
_IOMGR_PRIVATE_MAX *353*
_MEM_MMAP *276*
_NTO_CHF_COID_DISCONNECT *140*
_NTO_CHF_DISCONNECT *140*
_NTO_CHF_FIXED_PRIORITY *139*, *153*
_NTO_CHF_THREAD_DEATH *140*
_NTO_CHF_UNBLOCK *140–142*, *164*, *185*, *298*
  and kernel timeouts *185*
  modifying client's behavior *142*
_NTO_INTR_FLAGS_END *196*
_NTO_INTR_FLAGS_EXCLUSIVE *197*
_NTO_INTR_FLAGS_NO_UNMASK *197*
_NTO_INTR_FLAGS_PROCESS *197*
_NTO_INTR_FLAGS_TRK_MSK *197*
_NTO_MI_BITS_64 *121*
_NTO_MI_BITS_DIFF *121*
_NTO_MI_ENDIAN_BIG *121*
_NTO_MI_ENDIAN_DIFF *121*
_NTO_MI_NET_CRED_DIRTY *121*
_NTO_MI_UNBLOCK_REQ *121*, *143–144*, *297–298*, *322*
_POSIX_DEVDIR_FROM *264*
_POSIX_DEVDIR_TO *264*
_PULSE_CODE_UNBLOCK *137*
_RESMGR_CONNECT_NFUNCS *228*
_RESMGR_DEFAULT *249*, *265*, *320*
_RESMGR_ERRNO() 250
_RESMGR_FLAG_AFTER *233–234*
_RESMGR_FLAG_BEFORE *233–234*
_RESMGR_FLAG_DIR *233*, *327*, *330*
_RESMGR_FLAG_FTYPEALL *233*
_RESMGR_FLAG_FTYPEONLY *233*
_RESMGR_FLAG_OPAQUE *233*
_RESMGR_FLAG_SELF *233*
_RESMGR_IO_NFUNCS *229*
_RESMGR_NOREPLY *249*, *312*, *326*
_RESMGR_NPARTS() 249, *321*

_RESMGR_PTR() 250, *321*
_RESMGR_STATUS() 231, *254–256*, *258–259*, *261*, *263*, *266–267*, *270*, *272–274*, *276*, *281*, *286*, *288*, *291*, *293–294*, *296–298*, *300–303*
_SC_PAGESIZE *51*
`/dev/null` resource manager *220*
`<sys/netmgr.h>` *149*
`<sys/neutrino.h>` *114–116*, *130*, *135–136*, *168*, *183*
`<sys/siginfo.h>` *168*
`<time.h>` *166*

A

absolute timer *160*, *166*, *185*
  converting time formats *167*
  defined *381*
  example *167*
access control lists (ACLs) *246*
ACL I/O function handler *253*
address space *35*
adjusting time of day
  abruptly *178*
  gradually *178*
adjusting timebase *179*
alignment
  defined *381*
arming timeouts *183*, *185*
asctime() 167
asynchronous *381*
  defined *381*
    See also synchronous
asynchronous messaging, *See* pulses
atomic operation *64*
  defined *381*
atomic_*() 206
atomic_set() 206
attribute structure
  defined *381*
  thread *49*

B

barrier
  analogy *58*
  and threads *58*
  defined *381*
base timing resolution
  getting and setting *179*
  limits *179*
basename() 41
Bell, Gordon *16*
block_func() 82, *87*
blocking
  defined *382*
  in client due to message passing *102*
blocking state *29*

C

cancellation point *134*
cautions
  about timers and creating threads on trigger *177*
change file mode I/O function handler *255*
Change ownership I/O function handler *256*
change timestamps I/O function handler *257*
channel
  abstraction *120*
  as class of service *119*
  constants
    _NTO_CHF_COID_DISCONNECT *140*
    _NTO_CHF_DISCONNECT *140*
    _NTO_CHF_FIXED_PRIORITY *139*
    _NTO_CHF_THREAD_DEATH *140*
    _NTO_CHF_UNBLOCK *140*, *298*
  creation by server *114*
  defined *382*
  with multiple threads *120*
channel ID *112*, *119*, *121*
  how to find *123*
  process manager *213*
ChannelCreate() 111, *114*, *139*, *153*, *170*, *298*, *382*
  example *170*
  flags *139*
    _NTO_CHF_COID_DISCONNECT *140*
    _NTO_CHF_DISCONNECT *140*
    _NTO_CHF_FIXED_PRIORITY *139*
    _NTO_CHF_THREAD_DEATH *140*
    _NTO_CHF_UNBLOCK *140–141*
  priority inheritance *153*
ChannelDestroy() 111, *114*
CHIDs
  message passing *124*
chips
  82C54 *158*

© 2020, QNX Software Systems Limited

chmod() 221
chown() 221
`cksum` *361*
class of service
    via channels *119*
clearing timeouts *183*
client
    and not replying to them *176*
    assumptions about data area *126*
    basic operation *114*
    behavior modified by _NTO_CHF_UNBLOCK *142*
    being notified by server *138*
    blocked during *MsgSend() 116*
    busy server *102*
    connecting to server *112*
        diagram *114*
    defined *382*
    establishing a connection *112*
    informing server of unblock *185*
    limiting transfer size *117*
    multithreaded server *110*
    node descriptor *121*
    operation of *112*
    reply blocked and server *103*
    reply-blocked state *102*
    send-blocked state *102*
    server/subserver *108*
    servers with mismatched buffer sizes *125*
    specifying event to server *139*
    state
        diagram *102*
    timeouts *168*
    unblocked by server *116*
    unblocked by timeout *140*
    unblocking
        due to signal *140*
    unblocking server *116*
client/server *108*
    analogy *108*
    example *109*
    message passing *102*
    problems with single threaded *108*
clock
    drift *158*
    hardware *157*
    how maintained *157*
clock *(continued)*
    jitter *159–160*
        diagram *159*
clock tick
    adjusting time gradually *178*
clock_getres() 177
clock_gettime() 177
CLOCK_MONOTONIC *165*, *180–182*
    characteristics *181*
clock_nanosleep() 182
CLOCK_REALTIME *165–166*, *178*, *180–183*
    used with *ClockAdjust() 178*
clock_settime() 177
CLOCK_SOFTTIME *165*, *180–182*
ClockAdjust() 177
    CLOCK_REALTIME *178*
    `struct _clockadjust` *178*
ClockCycles() 177, *179*
ClockPeriod() 30, *177–178*
    `struct _clockperiod` *179*
ClockTime() 177
close duplicate OCB I/O function handler *258*, *267*
close OCB I/O function handler *251*, *259–260*
close() 104, *147*
CODE_TIMER *171*
command
    `cksum` *361*
    `devc-pty` *102*
    `esh` *103*
    `export` *39*
    `fs-cache` *216*
    `fs-qnx6` *126*, *128*, *132*
    `grep` *358*
    `gunzip` *39*
    `gzip` *39*, *362*
    `ls` *38*, *41*, *358*, *361*
    `make` *358*
    `mqueue` *234*
    `nice` *35–36*
    `pidin` *53*, *102–103*
    `procnto` *103*
    `tar` *362*
    `telnet` *361*
condition variable *68*
    defined *382*
    See also synchronization
condvar, *See* synchronization

ConnectAttach() *111–112*, *114*, *123–124*, *147*, *149*, *171*, *382*
example *112*, *171*
networked case *147*
ConnectDetach() *111–112*
connection
defined *382*
connection ID *121*, *213*
as equivalent to file descriptor *125*
defined *382*
obtaining *112*
resource manager *211*
constants
_FTYPE_ANY *234*
_FTYPE_MQUEUE *234*
_NTO_INTR_FLAGS_END *196*
_NTO_INTR_FLAGS_EXCLUSIVE *197*
_NTO_INTR_FLAGS_NO_UNMASK *197*
_NTO_INTR_FLAGS_PROCESS *197*
_NTO_INTR_FLAGS_TRK_MSK *197*
_NTO_MI_BITS_64 *121*
_NTO_MI_BITS_DIFF *121*
_NTO_MI_ENDIAN_BIG *121*
_NTO_MI_ENDIAN_DIFF *121*
_NTO_MI_NET_CRED_DIRTY *121*
_NTO_MI_UNBLOCK_REQ *121*, *143–144*
_SC_PAGESIZE *51*
channel
_NTO_CHF_COID_DISCONNECT *140*
_NTO_CHF_DISCONNECT *140*
_NTO_CHF_FIXED_PRIORITY *139*, *153*
_NTO_CHF_THREAD_DEATH *140*
_NTO_CHF_UNBLOCK *140–142*, *164*, *185*, *298*
clock
CLOCK_MONOTONIC *165*, *180–182*
CLOCK_REALTIME *165–166*, *178*, *180–183*
CLOCK_SOFTTIME *165*, *180–182*
CODE_TIMER *171*
error
EINTR *140*
ENOSYS *118*
EOK *123*, *213*, *216*
EROFS *123*, *213*, *231*
ETIMEDOUT *184–185*
FD_CLOEXEC *42*
message passing
_IO_MAX *118*
constants *(continued)*
message passing *(continued)*
_NTO_CHF_UNBLOCK *298*
_NTO_MI_UNBLOCK_REQ *297–298*, *322*
MT_TIMEDOUT *173*
ND_LOCAL_NODE *150*
open mode
O_RDONLY *241*
O_RDWR *241*
O_WRONLY *213*, *241*
P_NOWAIT *41–42*
P_NOWAITO *41*
P_OVERLAY *41*
P_WAIT *41*
POOL_FLAG_EXIT_SELF *83*
POOL_FLAG_USE_SELF *84–85*
POSIX_SPAWN_EXPLICIT_CPU *45*
POSIX_SPAWN_SETPGROUP *45*
process
SPAWN_NOZOMBIE *41*
PTHREAD_EXPLICIT_SCHED *51–52*
PTHREAD_STACK_LAZY *51*
PTHREAD_STACK_NOTLAZY *51*
pulse
_PULSE_CODE_UNBLOCK *137*
resource manager
_IO_ACL *253*
_IO_CHMOD *255*
_IO_CHOWN *256*
_IO_CLOSE *259*, *328*
_IO_CONNECT *261*, *273*, *277*, *279*, *285*, *287*, *299*, *328*, *353*
_IO_CONNECT_COMBINE *279*
_IO_CONNECT_COMBINE_CLOSE *238*, *279*
_IO_CONNECT_LINK *261*
_IO_CONNECT_MKNOD *273*
_IO_CONNECT_MOUNT *277*
_IO_CONNECT_OPEN *279*
_IO_CONNECT_READLINK *285*
_IO_CONNECT_RENAME *287*
_IO_CONNECT_UNLINK *299*
_IO_DEVCTL *226*, *263*, *317*
_IO_DUP *267*
_IO_FDINFO *268*
_IO_FLAG_RD *241*
_IO_FLAG_WR *241*
_IO_LSEEK *293–294*, *328*
_IO_MMAP *275*

© 2020, QNX Software Systems Limited

constants *(continued)*
resource manager *(continued)*
_IO_MSG *226*, *302*, *348–350*, *352–353*
_IO_NOTIFY *289*
_IO_NOTIFY64 *289*
_IO_OPENFD *281*
_IO_PATHCONF *282*
_IO_READ *283–284*, *308*, *328*
_IO_READ64 *283*
_IO_SPACE *292*
_IO_STAT *295*
_IO_SYNC *271*
_IO_UTIME *257*
_IO_UTIME64 *257*
_IO_WRITE *303*, *313*
_IO_WRITE64 *303*
_IO_XTYPE_NONE *311*
_IO_XTYPE_OFFSET *284*, *311–313*, *316*
_IO_XTYPE_READDIR *284*
_IOFUNC_NFUNCS *247*
_IOMGR_PRIVATE_BASE *353*
_IOMGR_PRIVATE_MAX *353*
_POSIX_DEVDIR_FROM *264*
_POSIX_DEVDIR_TO *264*
_RESMGR_CONNECT_NFUNCS *228*
_RESMGR_DEFAULT *249*, *265*, *320*
_RESMGR_FLAG_AFTER *233*
_RESMGR_FLAG_BEFORE *233*
_RESMGR_FLAG_DIR *233*, *327*, *330*
_RESMGR_FLAG_FTYPEALL *233*
_RESMGR_FLAG_FTYPEONLY *233*
_RESMGR_FLAG_OPAQUE *233*
_RESMGR_FLAG_SELF *233*
_RESMGR_IO_NFUNCS *229*
_RESMGR_NOREPLY *249*, *312*, *326*
DCMD_ALL_SETFLAGS *246*
F_ALLOCSP *293*
F_FREESP *293*
IOFUNC_ATTR_ATIME *311*, *316*
IOFUNC_ATTR_DIRTY_TIME *311*
IOFUNC_ATTR_MTIME *316*
IOFUNC_MOUNT_32BIT *246*
IOFUNC_MOUNT_FLAGS_PRIVATE *246*
IOFUNC_OCB_FLAGS_PRIVATE *242*
IOFUNC_OCB_MMAP *242*
IOFUNC_OCB_MMAP_UNIQUE *242*
IOFUNC_OCB_PRIVILEGED *242*
IOFUNC_OCB_T *322*

constants *(continued)*
resource manager *(continued)*
IOFUNC_PC_ACL *246*
IOFUNC_PC_CHOWN_RESTRICTED *246*, *256*
IOFUNC_PC_LINK_DIR *246*
IOFUNC_PC_NO_TRUNC *246*
IOFUNC_PC_SYNC_IO *246*
O_DSYNC *246*
O_RSYNC *246*
O_SYNC *246*
scheduling
SCHED_FIFO *52*
SCHED_OTHER *52*
SCHED_RR *52*
sharing flags
SH_COMPAT *242*
SH_DENYNO *242*
SH_DENYRD *242*
SH_DENYRW *242*
SH_DENYWR *242*
signal
SIGALRM *162*, *177*
SIGEV_INTR *161*, *164*, *201–202*, *205*
SIGEV_PULSE *161–163*, *171*, *201*
SIGEV_PULSE_PRIO_INHERIT *163*, *171*
SIGEV_SIGNAL *161–163*, *201*
SIGEV_SIGNAL family *162*
SIGEV_SIGNAL_CODE *161–163*
SIGEV_SIGNAL_THREAD *161–163*
SIGEV_THREAD *161–163*, *201*
SIGEV_UNBLOCK *161*, *163*, *183–184*
SIGEV_UNBLOCK example *183*
SIGSEGV *51*
SIGUSR1 *177*
thread
STATE_CONDVAR *32*, *74*
STATE_DEAD *32*
STATE_INTR *32–33*
STATE_JOIN *32*
STATE_MUTEX *32*
STATE_NANOSLEEP *32–33*
STATE_NET_REPLY *32*
STATE_NET_SEND *32*
STATE_READY *32*, *96*, *102*, *151–152*, *157*, *159–160*, *188*, *194*, *202*
STATE_RECEIVE *32–33*, *102*, *153*, *164*
STATE_REPLY *32–33*, *141*, *176*
STATE_RUNNING *32*, *96*, *188*

constants *(continued)*
thread *(continued)*
STATE_SEM *32*
STATE_SEND *32–33*, *138*, *141*
STATE_SIGSUSPEND *32*
STATE_SIGWAITINFO *32*
STATE_STACK *32*
STATE_STOPPED *32*
STATE_WAITCTX *32*
STATE_WAITPAGE *32*
STATE_WAITTHREAD *32*
timer
TIMER_ABSTIME *166*
consumer
and producer *71*
state analysis *73*
and producer using condvars
example *74*
context switch *28*, *35*
context_alloc() 82, *87*
context_free() 82, *87*
cookbook *306*
counter
high accuracy *179*
high frequency *179*
CPU
hog *156*
usage, monitoring *180*
create file link connect function handler *261*
Creceive() (QNX 4) *342*, *354*
ctime() 167

D

data structure, *See* structure
data type, *See* structure
data types
`forksafe_mutex_t` *44*
`struct _clockadjust` *178*
`struct _clockperiod`
members *179*
`struct itimerspec` *166*
`struct sigevent` *139*, *183*
and SIGEV_UNBLOCK *183*
shortcut initialization *184*
`struct sigevent` example *183*
`struct timespec` *166*
DCMD_ALL_SETFLAGS *246*
deadlock
defined *382*
with message passing *119*
decoupling *66*
via message passing *100*, *105–106*
delay() 159
detaching interrupt handlers *196*
`devc-pty` *102*
devctl() 211, *246*, *348–350*, *352*, *386*
device control I/O function handler *235–236*, *263*, *316–317*, *320–321*
diagram
big picture of timer chain *158*
clock jitter *159*
InterruptAttach() and wakeups *203*
InterruptAttachEvent() and wakeups *203*
interrupts and waking up only when required *203*
interrupts with unnecessary wakeups *203*
server/subserver *109*
using *InterruptAttach() 202*
`dirent` *284*, *327*
disabling interrupts *188*
discontinuities in time flow *178*
dispatch_block() 222
dispatch_context_alloc() 222
dispatch_create() 221, *307*
dispatch_handler() 222
`dispatch.h`, include after `resmgr.h` *87*
Dodge, Dan *16*
domain of authority *218*
duplicate a file descriptor I/O function handler *267*

E

edge-sensitive interrupt *191*
diagram *191*
EINTR *140*
message passing *140*
enabling interrupts *188*
endian
server flags *121*
ENOSYS *118*
environment variable *39*
PATH 40–41
EOK *123*, *213*, *216*
EROFS *123*, *213*, *231*
errno
MsgError() 123, *250*

© 2020, QNX Software Systems Limited

errno (continued)
MsgReply() 123
`esh` *103*
ETIMEDOUT *184–185*
event
and interrupt handlers *199*
and interrupts *191*
and ISRs *199*
example
`/dev/null` resource manager *220*
absolute timers *167*
barriers *58*
car using timers *156*
ChannelCreate() 170
ConnectAttach() 112, *171*
connecting to a server *112*
creating a `tar` file *362*
creating a thread *52*, *54*
demultiplexing pulse versus message *135*
demultiplexing the pulse code *136*
detaching interrupt handler *196*
file
`time1.c` *168*
`tt1.c` *183*
filling `struct itimerspec` *167*
interrupt handler *195*, *197*
InterruptAttach() 195, *197*
InterruptAttachEvent() 199
InterruptWait() 199
IOV *130*
ISR *195*, *197*
ISR pseudo-code *190*
kernel timeout *183–184*
message passing *103*, *113*
`fs-qnx6` *126*
multipart messages *130*, *132–133*
replying with no data *123*
server *116*
messages *168*, *170–171*, *173–174*
MsgReadv() 133
MsgReceive() 116, *126*, *133*, *170*
MsgReply() 116, *173–174*
MsgSend() 113, *129*
MsgSendv() 130
mutex *90*
netmgr_remote_nd() 150
netmgr_strtond() 150
example *(continued)*
networked message passing *105*
node descriptors *150*
non-blocking *pthread_join() 184*
one-shot timers *167*
periodic timers *167*
priority inversion *151*
producer *72*
producer and consumer *71*, *74*
pthread_attr_init() 52
pthread_attr_setdetachstate() 52
pthread_attr_setinheritsched() 52
pthread_attr_setschedpolicy() 52
`pthread_attr_t` *52*
pthread_barrier_init() 58
pthread_barrier_wait() 58
pthread_cond_signal() 74
pthread_cond_wait() 74
pthread_create() 52, *54*, *58*, *74*
pthread_join() 56, *183*
pthread_mutex_destroy() 90
pthread_mutex_init() 90
pthread_mutex_lock() 74, *90*
pthread_mutex_unlock() 74, *90*
pthread_rwlock_destroy() 93
pthread_rwlock_init() 93
pthread_rwlock_rdlock() 93
pthread_rwlock_unlock() 93
pthread_rwlock_wrlock() 93
pthread_sleepon_lock() 71–72
pthread_sleepon_signal() 72
pthread_sleepon_unlock() 71–72
pthread_sleepon_wait() 72
pulses *168*, *170–171*, *173–174*
receive ID *116*
receiving a message *135*
receiving a pulse *135*
relative timers *167–168*, *170–171*, *173–174*
replying with an error code *123*
replying with just EOK *123*
resource manager
device control I/O function handler *316*
open connect function handler *235*
read I/O function handler *308–309*
returning data to a client *308–309*
write I/O function handler *313*
rwlock *93*

example *(continued)*
SETIOV() macro *132*
SIGEV_PULSE_INIT() (macro) *171*
SIGEV_THREAD_INIT() (macro) *177*
`struct itimerspec` *171*
`struct sigevent` *171*
SIGEV_UNBLOCK *183*
synchronization *88*
the pulse value *136*
thread *58*
thread pool *84*
thread_pool_create() 84
thread_pool_start() 84
thread_pool_start() pseudo code *87*
timeouts *168*, *170–171*, *173–174*
timer_create() 171
timer_create() and signals *177*
timer_settime() 171
TimerTimeout() 183–184
and multiple states *185*
where to use *pthread_cond_broadcast() 77*
where to use *pthread_cond_signal() 76*
where to use *pthread_sleepon_broadcast() 77*
where to use *pthread_sleepon_signal() 76*
write() 129
exceptions and scheduling *96–97*
exec() 46
exec() family *36–38*, *40–41*
execl() 37, *41*
execle() 37
execlp() 37, *41*
execlpe() 37
execution time, monitoring *180*
execv() 37
execve() 37
execvp() 37
execvpe() 37
exit function *48*
exit() 196
and interrupts *196*
`export` *39*

F

F_ALLOCSP *293*
F_FREESP *293*
faults and scheduling *96–97*
fcntl() 42, *246*
FD_CLOEXEC *42*
fgets() 210, *217*
FIFO scheduling *29*
defined *383*
`FILE` *211*, *217*
file descriptor
and Resource managers *211*
connection ID *125*, *213*
resource manager *213*
file descriptor information I/O function handler *268*
File locking I/O function handler *269*
file stream
and Resource managers *211*
filesystem
`chown` restricted *246*
server example *125*
union *215*, *217*
floating-point operations
not safe to use in ISRs *189*
not safe to use in signal handlers *164*
flush data I/O function handler *271*
fopen() 212, *214*, *217*
fork() 36–37, *42–43*, *46*, *97*
and resource managers *46*
and threads *46*
avoid *46*
forksafe_mutex_*() 44
`forksafe_mutex_t` *44*
fprintf() 210
`fs-cache` *216*
`fs-qnx6` *126*, *128*, *132*, *212*
function
atomic
atomic_*() 206
atomic_set() 206
basename() 41
block_func() 82, *87*
channel
ChannelCreate() 111, *114*, *139*, *141*, *170*, *382*
ChannelCreate() example *170*
ChannelDestroy() 111, *114*
ConnectAttach() 111–112, *114*, *124*, *147*, *149*, *171*, *382*
ConnectAttach() example *171*
ConnectAttach() prototype *112*
ConnectDetach() 111–112

© 2020, QNX Software Systems Limited

function *(continued)*
chmod() 221
chown() 221
clock_nanosleep() 182
close() 104, *147*
context_alloc() 82, *87*
context_free() 82, *87*
Creceive() 342, *354*
delay() 159
devctl() 211, *348–350*, *352*, *386*
dispatch_create() 221
event related
SIGEV_INTR_INIT() (macro) *164*
SIGEV_PULSE_INIT() (macro) *163*
SIGEV_SIGNAL_CODE_INIT() (macro) *163*
SIGEV_SIGNAL_INIT() (macro) *163*
SIGEV_SIGNAL_THREAD_INIT() (macro) *163*
SIGEV_THREAD_INIT() (macro) *164*
SIGEV_UNBLOCK_INIT() (macro) *163*
exit() 196
fcntl() 42
fgets() 210, *217*
file descriptor information I/O handler *268*
fopen() 212, *214*, *217*
fork() 36–37, *42–43*, *46*, *97*
forksafe_mutex_*() 44
fprintf() 210
getppid() 124
gotAMessage() 170, *174*
gotAPulse() 170, *173*
handler_func() 82, *87*
in() 206*
in8() 199
interrupt
Interrupt() family *206*
InterruptAttach() 194–197, *201–203*, *205*, *208*
InterruptAttach() diagram *203*
InterruptAttachEvent() 194–197, *199*, *201–206*, *208*, *355*
InterruptAttachEvent() diagram *203*
InterruptAttachEvent() example *199*
InterruptAttachEvent() versus *InterruptAttach() 201–203*
InterruptDetach() 196
InterruptDisable() 188, *206–207*
InterruptEnable() 188, *206–207*
InterruptLock() 188, *206–208*, *381*

function *(continued)*
interrupt *(continued)*
InterruptMask() 206
InterruptUnlock() 188, *206–208*, *381*
InterruptUnmask() 197, *206*
InterruptWait() 33, *199*, *201–202*, *205*
InterruptWait() example *199*
ISR-safe *205*
atomic() family *206*
in() family *206*
InterruptDisable() 206
InterruptEnable() 206
InterruptLock() 206
InterruptMask() 206
InterruptUnlock() 206
InterruptUnmask() 206
mem() family *206*
out() family *206*
str() family *206*
kill() 343
lseek() 211, *217*, *338*, *382*, *386*
malloc() 129–131, *206*
mem() 206*
memcpy() 129–130, *206*
message passing
ChannelCreate() 111, *114*, *139*, *141*, *153*, *298*
ChannelDestroy() 111, *114*
ConnectAttach() 111–112, *114*, *123–124*, *147*, *149*
ConnectDetach() 111–112
MsgDeliverEvent() 111, *119*, *122*, *138–139*, *147–148*, *291*, *353*, *385*
MsgError() 111, *118*, *123*, *250*
MsgInfo() 120, *143*
MsgRead() 111, *126*, *128*, *134*, *147–148*, *315*
MsgReadv() 111
MsgReceive() 111, *115–117*, *120*, *122–123*, *125–126*, *134–135*, *137–138*, *140*, *142–143*, *145*, *147–148*, *151*, *153*, *170*, *176*, *201*, *219*, *342*, *385*
MsgReceive() example *170*
MsgReceivePulse() 111, *134–135*, *137–138*, *385*
MsgReceivev() 111, *137*, *230*, *385*
MsgReply() 111, *115–117*, *122–123*, *128*, *135*, *143*, *147–148*, *311–312*

function *(continued)*
message passing *(continued)*
MsgReply() example *173–174*
MsgReplyv() 111, *133*, *249*, *312*
MsgSend() 111, *113–114*, *116–117*, *122–123*, *125*, *131*, *133–135*, *140*, *185*, *302*
MsgSend() example *113*
MsgSend() family *113*, *133*
MsgSendnc() 111, *133*
MsgSendsv() 111, *133–134*
MsgSendsvnc() 111, *133–134*
MsgSendv() 96, *111*, *133*, *140*, *311*, *353*
MsgSendvnc() 111, *133*
MsgSendvs() 111, *133*, *140*
MsgSendvsnc() 111, *133*
MsgVerifyEvent() 139
MsgWrite() 111, *123*, *126*, *128*, *134*, *147–148*, *312*
MsgWritev() 111, *133*, *312*
message queue
mq_open() 234
mq_receive() 234
message-sending, *See* message passing
mktime() 167
MsgReceive() 132–133
MsgReceivev() 132
MsgSend() 131
MsgSendv() 131
MsgVerifyEvent() 139
name_attach() 124, *345*
name_close() 124, *345*
name_detach() 124, *345*
name_open() 124, *345*
nanospin() 157
netmgr_remote_nd() 149
netmgr_strtond() 149
network
netmgr_remote_nd() 149
netmgr_strtond() 149
open() 104–105, *124–125*, *146–147*, *149*, *211–217*, *219*, *229*, *345*, *353*, *382*, *384*, *386*
implementation *212*
out() 206*
POSIX
mq_open() 234
mq_receive() 234

function *(continued)*
POSIX threads *385*, *387*
pread() 311
printf() 43
process
exec() 46
exec() family *36–38*, *40–41*
execl() 37, *41*
execle() 37
execlp() 37, *41*
execlpe() 37
execv() 37
execve() 37
execvp() 37
execvpe() 37
getppid() 124
posix_spawn() family *36–37*, *45–46*
spawn() 37–38, *42*
spawn() family *36–38*, *40–41*
spawnl() 37–38
spawnle() 37–38
spawnlp() 37–38
spawnlpe() 37–38
spawnp() 37–38
spawnv() 37–38
spawnve() 37–38
spawnvp() 37–38
spawnvpe() 37–38
process creation *36–37*, *42*
process transformation *37*
pthread_atfork() 44
pthread_mutex_destroy() 90
pthread_mutex_init() 90
pthread_mutex_lock() 90
pthread_mutex_timedlock() 181
pthread_mutex_unlock() 90
pthread_rwlock_destroy() 93
pthread_rwlock_init() 93
pthread_rwlock_rdlock() 93
pthread_rwlock_unlock() 93
pthread_rwlock_wrlock() 93
pulse
MsgReceive() 135
MsgReceivePulse() 135
pulse_attach() 226
pulses
pulse_attach() 223
pulse_detach() 223

© 2020, QNX Software Systems Limited

function *(continued)*
QNX 4
Creceive() 342, *354*
qnx_name_attach() 345
qnx_name_locate() 345
qnx_proxy_attach() 353
Receive() 342, *354*
Reply() 342
Send() 342, *348*
tfork() 341
Trigger() 353
read I/O handler *308*, *313*
example *308*
read() 125, *147*, *211*, *217*, *219*, *221*, *348–349*, *352*, *386*
rename() 219
Reply() 342
resmgr_attach() 231
resource manager
_IO_SET_WRITE_NBYTES() 315–316
ACL I/O handler *253*
change file mode I/O handler *255*
Change ownership I/O handler *256*
change timestamps I/O handler *257*
close duplicate OCB I/O handler *258*, *267*
close OCB I/O handler *251*, *259–260*
create file link connect handler *261*
devctl() 246
device control I/O handler *235–236*, *263*, *316–317*, *320–321*
dispatch_block() 222
dispatch_context_alloc() 222
dispatch_create() 307
dispatch_handler() 222
duplicate a file descriptor I/O handler *267*
fcntl() 246
file descriptor information I/O handler *268*
File locking I/O handler *269*
flush data I/O handler *271*
io_open_default() 236
iofunc_ability_check() 255–256, *279–280*
iofunc_acl_default() 253
iofunc_acl() 253
iofunc_attr_init() 221
iofunc_chmod_default() 255
iofunc_chmod() 255
iofunc_chown_default() 256
iofunc_chown() 256

function *(continued)*
resource manager *(continued)*
iofunc_close_dup_default() 259
iofunc_close_dup() 259
iofunc_devctl_default() 263, *320*
iofunc_devctl_verify() 263
iofunc_devctl() 263, *265*
iofunc_func_init() 221, *227–229*, *235*, *253*, *306*, *308*, *320*
iofunc_link() 261
iofunc_lock_default() 244, *269*
iofunc_lock_ocb_default() 251, *272*
iofunc_lseek_default() 293
iofunc_lseek() 293
iofunc_mknod() 273
iofunc_mmap_default() 244, *275*
iofunc_mmap() 275
iofunc_notify_remove_strict() 289
iofunc_notify_remove() 289
iofunc_notify_trigger_strict() 289
iofunc_notify_trigger() 289, *291*
iofunc_notify() 289
iofunc_ocb_attach() 279–280
iofunc_ocb_calloc() 276, *322*
iofunc_ocb_free() 322
iofunc_open_default() 236, *279*
iofunc_open() 279
iofunc_openfd_default() 280
iofunc_openfd() 280
iofunc_pathconf_default() 282
iofunc_pathconf() 282
iofunc_read_default() 283
iofunc_read_verify() 283–284, *310*, *314*
iofunc_readlink() 285
iofunc_rename() 287
iofunc_space_verify() 292
iofunc_stat_default() 295, *312*
iofunc_stat_format() 295–296
iofunc_stat() 295
iofunc_sync_default() 271
iofunc_sync_verify() 246, *271*, *284*, *304*
iofunc_sync() 271
iofunc_unblock_default() 298
iofunc_unblock() 297–298
iofunc_unlink() 299
iofunc_unlock_ocb_default() 251, *301*
iofunc_utime_default() 257
iofunc_utime() 257

function *(continued)*
resource manager *(continued)*
iofunc_write_default() 303
iofunc_write_verify() 303–304, *314*
lock OCB I/O handler *272*, *301*
make filesystem node connect handler *273*
map memory I/O handler *275*
mount connect handler *277*
open connect handler *231*, *235–236*, *259*, *278*
open connection handler *323*
open file descriptor I/O handler *280*
pathconf I/O handler *282*
power I/O handler *283*
read I/O function handler *312*
read I/O handler *235–236*, *283–284*, *308*, *312–314*, *316*, *326–328*
read link connect handler *285*
register for notifiction I/O handler *289*
rename connect function handler *287*
resize file I/O handler *292*
resmgr_attach() 222–223, *226*, *231–233*, *235*, *253*, *308*
resmgr_bind_ocb() 280
resmgr_detach() 223
resmgr_msgread() 223, *315*
resmgr_msgreadv() 223, *231*, *265*, *304*, *315*
resmgr_msgwrite() 223
resmgr_msgwritev() 223, *265*
resmgr_open_bind() 223, *237*
set file position I/O handler *293*, *328*
shutdown I/O handler *295*
stat I/O handler *295*, *311*
unblock connect handler *296*, *298*, *322*
unblock I/O handler *297*
unlink connection handler *299*
unlock OCB I/O handler *301*
user defined message I/O handler *302*
write I/O handler *235–236*, *284*, *303*, *313*
rsrcdbmgr_devno_attach() 247
scheduling
ClockPeriod() 30
sched_get_priority_max() 29
sched_get_priority_min() 29
sched_rr_get_interval() 30
sched_yield() 30
SchedYield() 30
setuid() 194
setupPulseAndTimer() 170–171

function *(continued)*
signal
SIGEV_INTR_INIT() (macro) *164*
sigev_notify_function() 163
SIGEV_PULSE_INIT() 171
SIGEV_PULSE_INIT() (macro) *163*
SIGEV_SIGNAL_CODE_INIT() (macro) *163*
SIGEV_SIGNAL_INIT() (macro) *163*
SIGEV_SIGNAL_THREAD_INIT() (macro) *163*
SIGEV_THREAD_INIT() (macro) *164*
SIGEV_UNBLOCK_INIT() (macro) *163*
sigwait() 164
sleep() 33, *96*, *156–157*, *180*
bad implementation *156*
stat() 221
strcmp() 206
strdup() 206
strftime() 167
synchronization
pthread_barrier_init() 58
pthread_barrier_wait() 56, *58*, *60*, *381*
sem_post() 386
sem_wait() 386
sysconf() 51
system() 36
tfork() (QNX 4) *341*
thread
pthread_atfork() 46
pthread_attr_destroy() 48–49
pthread_attr_getdetachstate() 48
pthread_attr_getguardsize() 48
pthread_attr_getinheritsched() 48
pthread_attr_getschedparam() 48
pthread_attr_getschedpolicy() 48
pthread_attr_getscope() 48
pthread_attr_getstack() 48
pthread_attr_getstackaddr() 48
pthread_attr_getstacklazy() 48
pthread_attr_getstackprealloc() 48
pthread_attr_getstacksize() 48
pthread_attr_getsuspendstate_np() 48
pthread_attr_init() 48–49
pthread_attr_set() family *49*
pthread_attr_setdetachstate() 48–49
pthread_attr_setguardsize() 48
pthread_attr_setinheritsched() 48–49, *51*
pthread_attr_setschedparam() 48–49, *51*
pthread_attr_setschedpolicy() 48–49, *51*

© 2020, QNX Software Systems Limited

function *(continued)*
thread *(continued)*
pthread_attr_setscope() 48, 50
pthread_attr_setstack() 48, 50
pthread_attr_setstackaddr() 48, 50
pthread_attr_setstacklazy() 48, 50
pthread_attr_setstackprealloc() 48, 50
pthread_attr_setstacksize() 48, 50
pthread_attr_setsuspendstate_np() 48–49
pthread_cancel() 96, 134
pthread_cond_broadcast() 76
pthread_cond_signal() 76
pthread_cond_wait() 74, 76, 79
pthread_create() 46, 49, 52, 57, 341
pthread_join() 49, 56–58, 60, 96, 183–184
pthread_join() example *183*
pthread_mutex_lock() 76, 80, 341
pthread_mutex_unlock() 76, 80
pthread_rwlock_destroy() 69
pthread_rwlock_init() 69
pthread_rwlock_rdlock() 69
pthread_rwlock_tryrdlock() 69
pthread_rwlock_unlock() 70
pthread_rwlock_wrlock() 69
pthread_rwlockattr_destroy() 69
pthread_rwlockattr_getpshared() 69
pthread_rwlockattr_init() 69
pthread_rwlockattr_setpshared() 69
pthread_setschedparam() 26, 153
pthread_setschedprio() 26, 153
pthread_sleepon_broadcast() 74, 76
pthread_sleepon_lock() 71, 76
pthread_sleepon_signal() 73–74, 76
pthread_sleepon_unlock() 71, 76
pthread_sleepon_wait() 71–72, 74, 76
thread_pool_control() 81
thread_pool_create() 81, 83–84
thread_pool_destroy() 81
thread_pool_limits() 81
thread_pool_start() 81, 83–85
thread_pool() family *107*
time
asctime() 167
clock_getres() 177
clock_gettime() 177
clock_settime() 177
ClockAdjust() 177–178
ClockCycles() 177, 179

function *(continued)*
time *(continued)*
ClockPeriod() 177
ClockTime() 177
ctime() 167
mktime() 167
strftime() 167
time() 167
timer_create() 165–166, 171, 177
timer_create() example *171*
timer_settime() 166–167, 171
timer_settime() example *171*
timer
ClockCycles() 179
ClockPeriod() 178
delay() 159
timer_create() 165
timer_settime() 166–167
TimerTimeout() 122, 140, 183–185, 342
TimerTimeout() example *183–184*
timer_create() 181
timing
nanospin() 157
sleep() 156–157
unblock_func() 87
waitpid() 41
write I/O handler *313*
example *313*
write() 104, 125–126, 129, 349, 353, 386

G

gather/scatter, *See* scatter/gather
getppid() 124
getting help *357, 360*
beta versions *361*
updates *362*
contacting technical support *360*
describing the problem *360–361*
be precise *361*
narrow it down *362*
read the manuals *358*
reproduce the problem *362*
training *363*
getting the time *177*
gotAMessage() 170, 174
gotAPulse() 170, 173
`grep` *358*

`gunzip` *39*
`gzip` *39*, *362*

H

handler routines, *See* resource managers, handler routines
handler_func() *82*, *87*
hardware
    82C54 component *158*
    and polling using timers *176*
    asynchronous nature of timers *159*
    changing timer rate *159*
    clock tick and timers *185*
    divider, used with timers *158*
    impact of integer divisor on timer *158*
    inactivity shutdown *176*
    used with timers *158*
    warm-up timer *177*
header files, order of inclusion *87*, *228–229*
high accuracy counter *179*
high frequency interrupts *205*
high-resolution timers *159*, *179*

I

I/O Vector, *See* IOV
IBM PC *15*
in()* *206*
in8() *199*
inactivity shutdown *176*
initializing
    `struct sigevent` *184*
        shortcut *184*
interrupt
    _NTO_INTR_FLAGS_END *196*
    _NTO_INTR_FLAGS_EXCLUSIVE *197*
    _NTO_INTR_FLAGS_NO_UNMASK *197*
    _NTO_INTR_FLAGS_PROCESS *197*
    _NTO_INTR_FLAGS_TRK_MSK *197*
    10 millisecond *157*
    8259 chip *190*
    analogy *188*
    associated thread *190–191*
    associating with thread or process *197*
    attaching handler *194–195*
    BSP specific *193*
    causes *188*
    chained *193*
interrupt *(continued)*
    chaining *196*, *205*
    clearing *189–190*, *201*
    clock
        diagram *158*
    complex *189*
    context switches *203*
    defined *188*
    detach on death of thread or process *196*
    disabling *188*, *207*
        `cli` *188*
    doing nothing in the ISR *188*
    doing the work in a thread *190*
    edge-sensitive *191*, *193*
        and cleanup *196*
        diagram *191*
        problems *191–193*
    enabling *188*, *207*
        `sti` *188*
    end of interrupt (EOI) *191*
    environment *205*
    EOI *193*
    ethernet device *192–193*
    events *191*
    exit() *196*
    filtering out unnecessary *205*
    floppy disk controller *190–191*
    functions
        InterruptAttach() diagram *202*
        InterruptAttachEvent() diagram *202*
        InterruptAttachEvent() example *199*
        InterruptWait() example *199*
    handlers
        detaching *196*
        detaching the last handler *196*
        events *199*
        example *195*, *197*
        goals *205*
        safe functions *205–206*
        `volatile` *199*
    handling *188*
    hardware *96*
    hardware acknowledgment *189*
    high frequency *205*
    ignoring *190*
    impact on scheduling *189–190*
    informing the kernel *188*

© 2020, QNX Software Systems Limited

interrupt *(continued)*
interrupt identifier *201*
InterruptAttach() 194
InterruptAttachEvent() 194
InterruptAttachEvent() versus *InterruptAttach() 201–203*
InterruptDetach() 196
ISRs *188–189*
detaching *196*
detaching the last handler *196*
events *199*
example *195*, *197*
floating-point operations aren't safe to use in *189*
goals *205*
pseudo-code example *190*
safe functions *205–206*
`volatile` *199*
kernel *202*
kernel ISR for timer *157*
latency *188*
leaving masked *197*
level-sensitive *191*, *193*
and cleanup *196*
diagram *191*
low frequency *205*
managing *188*
masking *190*
masking after detaching *196*
masking on detach of last handler *196*
minimizing latency *205*
minimizing time spent in *189*
not clearing the source *190*
optimal performance from exclusive attachment to interrupt vector *197*
order of processing *193*
permissions required *194*
PIC *190*, *193*
edge-sensitive mode *191*
edge-sensitive mode diagram *191*
level-sensitive mode *191*
level-sensitive mode diagram *191*
priority *189*
processing *190*
programmable interrupt controller *190*
edge-sensitive mode *191*
edge-sensitive mode diagram *191*

interrupt *(continued)*
programmable interrupt controller *(continued)*
level-sensitive mode *191*
level-sensitive mode diagram *191*
pulses *137*, *191*
readying a thread *188*, *190–191*, *194–195*, *199*, *202–203*
realtime *188*
realtime clock *96*
relationship to thread *190*
responsibilities *189*
returning `struct sigevent` *199*
role of ISR *189*
scheduling *96*, *157*, *188*
SCSI device *192–193*
serial handler *189*
servers *196*
sharing *192–193*, *196*
diagram *192*
problem *193*
problems *193*
SIGEV_INTR *161*, *201*, *205*
SIGEV_SIGNAL *201*
SIGEV_THREAD *201*
signals *191*
SMP *64*, *207*
source *189*
specifying order *196*
spending a long time in ISR *190*
startup code *194*
`struct sigevent` *191*, *194–195*, *199*, *201*
tail-end polling *189*
thread interaction *207*
thread-level interactions *199*
threads *190–191*
timing *157*
tracking mask/unmasks *197*
unmasking *190*
unmasking when attaching *196–197*
unnecessary wakeups
diagram *203*
used for timing *157*
used with timers *158*
using *InterruptWait() 201*
`volatile` *199*, *208*
waiting in thread *201*
waking up only when required *203*

interrupt *(continued)*
waking up unnecessarily *203*
writing *194*
interrupt service routine *188*
defined *383*
See also Interrupts
Interrupt() family *206*
InterruptAttach() 194–197, *201–203*, *205*, *208*
diagram *202–203*
example *197*
flags parameter *196*
InterruptAttachEvent() 194–197, *199*, *201–206*, *208*, *355*
diagram *203*
example *199*
flags parameter *196*
pseudo-code *204*
returning interrupt identifier *201*
InterruptAttachEvent() versus *InterruptAttach() 201–203*
InterruptDetach() 196
example *196*
InterruptDisable() 188, *206–207*
InterruptEnable() 188, *206–207*
InterruptLock() 188, *206–208*, *381*
InterruptMask() 206
InterruptUnlock() 188, *206–208*, *381*
InterruptUnmask() 197, *206*
InterruptWait() 33, *199*, *201–202*, *205*
and SIGEV_INTR *202*, *205*
example *199*
`io_acl_t` *253*
`io_chmod_t` *255–256*
`io_close_t` *259–260*
`io_devctl_t` *263*
`io_dup_t` *267*
`io_link_extra_t` *262*
`io_link_t` *262*
`io_lock_t` *270*
`io_lseek_t` *294*
`io_mknod_t` *274*
`io_mmap_t` *275*, *289*
`io_mount_t` *277*
`io_msg_t` *302*
io_open_default() 236
`io_open_t` *279*
`io_openfd_t` *281*
`io_pathconf_t` *282*
`io_read_t` *283*
`io_readlink_t` *285*
`io_rename_extra_t` *287*
`io_rename_t` *287*
`io_space_t` *292*
`io_stat_t` *295*
`io_sync_t` *271*
`io_unlink_t` *299*
`io_utime_t` *257*
`io_write_t` *303*
iofunc_ability_check() 255–256, *279–280*
iofunc_acl_default() 253
iofunc_acl() 253
IOFUNC_ATTR_ATIME *311*, *316*
IOFUNC_ATTR_DIRTY_TIME *311*
iofunc_attr_init() 221
IOFUNC_ATTR_MTIMEdl *316*
`iofunc_attr_t` *240*, *242–245*
iofunc_chmod_default() 255
iofunc_chmod() 255
iofunc_chown_default() 256
iofunc_chown() 256
iofunc_close_dup_default() 259
iofunc_close_dup() 259
iofunc_devctl_default() 263, *320*
iofunc_devctl_verify() 263
iofunc_devctl() 263, *265*
iofunc_func_init() 221, *227–229*, *235*, *253*, *306*, *308*, *320*
iofunc_link() 261
iofunc_lock_default() 244, *269*
iofunc_lock_ocb_default() 251, *272*
iofunc_lseek_default() 293
iofunc_lseek() 293
iofunc_mknod() 273
iofunc_mmap_default() 244, *275*
iofunc_mmap() 275
IOFUNC_MOUNT_32BIT *246*
IOFUNC_MOUNT_FLAGS_PRIVATE *246*
`iofunc_mount_t` *240*, *243*, *246–247*
iofunc_notify_remove_strict() 289
iofunc_notify_remove() 289
iofunc_notify_trigger_strict() 289
iofunc_notify_trigger() 289, *291*
iofunc_notify() 289
IOFUNC_NS_TIMESTAMP_SUPPORT *245*
iofunc_ocb_attach() 279–280

© 2020, QNX Software Systems Limited

iofunc_ocb_calloc() 276, *322*
IOFUNC_OCB_FLAGS_PRIVATE *242*
iofunc_ocb_free() 322
IOFUNC_OCB_MMAP *242*
IOFUNC_OCB_MMAP_UNIQUE *242*
IOFUNC_OCB_PRIVILEGED *242*
`iofunc_ocb_t` *240–242*
IOFUNC_OCB_T *322*
iofunc_open_default() 236, *279*
iofunc_open() 279
iofunc_openfd_default() 280
iofunc_openfd() 280
iofunc_pathconf_default() 282
iofunc_pathconf() 282
IOFUNC_PC_ACL *246*
IOFUNC_PC_CHOWN_RESTRICTED *246*, *256*
IOFUNC_PC_LINK_DIR *246*
IOFUNC_PC_NO_TRUNC *246*
IOFUNC_PC_SYNC_IO *246*
iofunc_read_default() 283
iofunc_read_verify() 283–284, *310*, *314*
iofunc_readlink() 285
iofunc_rename() 287
iofunc_space_verify() 292
iofunc_stat_default() 295, *312*
iofunc_stat_format() 295–296
iofunc_stat() 295
iofunc_sync_default() 271
iofunc_sync_verify() 246, *271*, *284*, *304*
iofunc_sync() 271
iofunc_unblock_default() 298
iofunc_unblock() 297–298
iofunc_unlink() 299
iofunc_unlock_ocb_default() 251, *301*
iofunc_utime_default() 257
iofunc_utime() 257
iofunc_write_default() 303
iofunc_write_verify() 303–304, *314*
`iofunc.h`, include before `resmgr.h` *228–229*
IOV *130–131*
    defined *383*
    See also Message passing
        See also Message passing
`iov_t` *130–131*
    defined *130*
ISR, *See* interrupt service routine

K

kernel
    as arbiter *28*
    base timing resolution *179*
    context switch *28*
    context-switch *35*
    preempting thread *29*
    readying a thread *157*, *185*
    resuming thread *29*
    special pulse *185*
    suspending a thread *157*
    synthesizing unblock pulse *141*
    timeouts *183*
        SIGEV_UNBLOCK *161*
    timer implementation *157*, *160*
    view of data in message pass *131*
kernel callouts
    defined *383*
kernel state
    blocking *32*
    complete list *32*
    STATE_CONDVAR *32*, *74*
    STATE_DEAD *32*
    STATE_INTR *32–33*
    STATE_JOIN *32*
    STATE_MUTEX *32*
    STATE_NANOSLEEP *32–33*
    STATE_NET_REPLY *32*
    STATE_NET_SEND *32*
    STATE_READY *32*, *96*, *102*, *151–152*, *157*, *159–160*, *188*, *194*, *202*
    STATE_RECEIVE *32–33*, *102*, *153*, *164*
        diagram *102*
    STATE_REPLY *32–33*, *102–103*, *141*, *176*, *185*
        diagram *102*
    STATE_RUNNING *32*, *96*, *188*
    STATE_SEM *32*
    STATE_SEND *32–33*, *102–103*, *138*, *141*, *185*
        diagram *102*
        when abnormal *103*
        when normal *103*
    STATE_SIGSUSPEND *32*
    STATE_SIGWAITINFO *32*
    STATE_STACK *32*
    STATE_STOPPED *32*
    STATE_WAITCTX *32*
    STATE_WAITPAGE *32*

kernel state *(continued)*
STATE_WAITTHREAD *32*
triggering timeout *183*
kernel timeout *185*
_NTO_CHF_UNBLOCK *185*
arming *185*
example *184*
message passing *185*
servers *185*
specifying multiple *185*
with *pthread_join() 183*
kill() 343
Krten, Rob *20*

L

latency, interrupt *188*
level-sensitive interrupts *191*
diagram *191*
limits
multipart messages *132*
range of pulse code values *136*
local node descriptor *112*
lock OCB I/O function handler *272*, *301*
low frequency interrupts *205*
`ls` *38*, *41*, *358*, *361*
lseek() 211, *217*, *338*, *382*, *386*

M

macros
for filling `struct sigevent` *163*
SIGEV_INTR_INIT() 164
SIGEV_PULSE_INIT() 163
SIGEV_SIGNAL_CODE_INIT() 163
SIGEV_SIGNAL_INIT() 163
SIGEV_SIGNAL_THREAD_INIT() 163
SIGEV_THREAD_INIT() 164
SIGEV_UNBLOCK_INIT() 163
resource manager
_IO_SET_CONNECT_RET() 278, *280*
_IO_SET_FDINFO_LEN() 269
_IO_SET_PATHCONF_VALUE() 282
_IO_SET_READ_NBYTES() 284
_IO_SET_WRITE_NBYTES() 304
_RESMGR_ERRNO() (deprecated) *250*
_RESMGR_NPARTS() 249, *321*
_RESMGR_PTR() 250, *321*
macros *(continued)*
resource manager *(continued)*
_RESMGR_STATUS() 231, *254–256*, *258–259*, *261*, *263*, *266–267*, *270*, *272–274*, *276*, *281*, *286*, *288*, *291*, *293–294*, *296–298*, *300–303*
SETIOV() 130–131
SIGEV_PULSE_INIT()
example *171*
SIGEV_THREAD_INIT()
example *177*
SIGEV_UNBLOCK_INIT() 183–184
example *183*
`make` *358*
make filesystem node connect function handler *273*
malloc() 129–131, *206*
map memory I/O function handler *275*
masking interrupts *190*, *196*
meet-me synchronization, *See* synchronization
mem() 206*
memcpy() 129–130, *206*
memory
manager *276*, *323*
physical, defined *385*
virtual, defined *387–388*
memory management unit, *See* MMU
memory protection *35*
message
combined *237–239*
why they work *239*
connect *213*, *219*
constants
_NTO_CHF_UNBLOCK *298*
_NTO_MI_UNBLOCK_REQ *297–298*, *322*
determining if pulse or message *176*
functions
ChannelCreate() 298
MsgDeliverEvent() 291
MsgRead() 315
MsgReply() 311–312
MsgReplyv() 312
MsgSend() 302
MsgSendv() 311
MsgWrite() 312
MsgWritev() 312
how to tell from pulses *176*
I/O *219*

© 2020, QNX Software Systems Limited

message *(continued)*
- identifying *118*
- not replying to client *176*
- other *219*
- receive ID, defined *385*
- receiving, defined *385*
- replying to multiple clients *176*
- replying, defined *386*
- resource manager *219*
  - _IO_DEVCTL *226*
  - _IO_MSG *226*
  - combine *250*
  - connect *237*
  - processing *250*
- send hierarchy, defined *387*
- sending
  - functions *386*
- types *118*

message passing *100*
- `<sys/neutrino.h>` *130*
- advantages *106*
- as decoupling *153*
- as object oriented design *106*
- as synchronization scheme *153*
- avoiding unnecessary copying *130*
- blocking client *102*, *116*
- buffer sizes *116*
- cancellation points *134*
- channel ID *112*, *121*
- *ChannelCreate() 139*, *141*, *153*
- client *112*
- client/server *102*
- confusion with timeouts *140*
- *ConnectAttach() 147*, *149*
- *ConnectDetach() 112*
- connection ID *121*
- data flow *115*
- deadlock *119*
- dealing with large buffers *129*
- decoupling of design *100*, *105–106*
- deferring data transfer *138–139*
- defined *383*
- diagram *115*
- distributing work over a network *109*
- done by C library *104*
- double standard in conventional OS *105*
- establishing client to server connection *112*

message passing *(continued)*
- example *103*, *105*
- excluding messages *138*
- filesystem example *126*
- finding a server *124*
  - ND/PID/CHID *123*
  - using a global variable *124*
  - using a resource manager *124*
  - using global variables *124*
  - using well-known files *124*
- finding the server's ND/PID/CHID *124*
- `fs-qnx6` message example *126*
- handling big messages in server *125*
- how to handle large transfers *128*
- interrupts *137*
- `iov_t` *130*
- kernel timeouts *185*
- limiting transfer size *117–118*, *122*
- modularity *100*
- *MsgDeliverEvent() 138–139*, *147–148*
- *MsgError()* versus *MsgReply() 123*
- *MsgInfo() 143*
- *MsgRead() 126*, *128*, *134*, *147–148*
- *MsgReceive() 115*, *126*, *132–135*, *137–138*, *140*, *142–143*, *145*, *147–148*, *151*, *153*
- *MsgReceive()* versus *MsgReceivev() 133*
- *MsgReceivePulse() 134*, *137–138*
- *MsgReceivev() 132*, *137*
- *MsgReply() 115*, *128*, *143*, *147–148*
- *MsgSend() 113*, *131*, *133–135*, *140*
  - example *113*
- *MsgSend()* family *133*
- *MsgSendnc() 133*
- *MsgSendsv() 133–134*
- *MsgSendsvnc() 133–134*
- *MsgSendv() 131*, *133*, *140*
- *MsgSendvnc() 133*
- *MsgSendvs() 133*, *140*
- *MsgSendvsnc() 133*
- *MsgWrite() 126*, *128*, *134*, *147–148*
- multipart messages *129–130*
  - example *130*, *132–133*
  - IOV *130–132*
  - kernel's view *131*
  - limitations *132*
- multipart versus linear *134*
- multiple threads *107*

message passing *(continued)*
multithreaded server *110*
ND/PID/CHIDs *124*
network
detailed analysis *146*
differences from local *147*
network implementation *145*
network transparent *153*
network-distributed *105*
networked *145*
networked case
determining how much data should have been transferred *148*
determining how much data was transferred *148*
networked overhead *147*
node descriptor *112*
not replying to the client *122*
notifying client *138*
obtaining a connection ID *112*
offsetting into the client's data *128*, *133*
peeking into a message *126*
phases *115*
priority *125*
process ID *112*
pulse
MsgReceive() *135*
MsgReceivePulse() *135*
receiving *135*
race condition with unblock *142*
reading from the client's address space *126*
readying a thread *102*
receive ID *122*
and reply *116*
receive-blocked *102*
diagram *102*
receiving only pulses *137*
receiving pulses only *138*
reply-blocked *102–103*
diagram *102*
REPLY-blocked *185*
reply-driven model *108*, *123*
example *109*
important subtlety *109*
replying to the client *122*
replying with no data *123*
example *123*
resource manager *104*
run time installability of components *100*

message passing *(continued)*
scatter/gather
defined *386*
SEND state
diagram *102*
send-blocked *102–103*
diagram *102*
SEND-blocked *185*
send-driven model *108*
example *109*
important subtlety *109*
server *115*
example *116*
server connection ID *121*
server replying to client *116*
server/subserver *107*
delegation of work *108*
SETIOV() (macro) *130*
SMP *107*
STATE_RECEIVE state *102*
diagram *102*
STATE_REPLY state *102–103*
diagram *102*
STATE_SEND state *102–103*
summary *153*
synthetic unblock pulse *141*
thread and channels *120*
thread pool *138*
timeouts
informing server *185*
timeouts and _NTO_CHF_UNBLOCK *185*
timer *137*
tracking owner of message *120*
transmit buffer *116*
transparency over network *105*
unblock *142*
unblocking
_NTO_MI_UNBLOCK_REQ *143–144*
client *140*
server *116*
unit testing *106*
useful minimal set of functions *111*
using IOV (vectored) functions *134*
using the _NTO_MI_UNBLOCK_REQ flag *144*
validity of receive ID *122*
vs. traditional OS *104–105*
with pool of threads *107*
write() example *129*

© 2020, QNX Software Systems Limited

message passing *(continued)*
writing a header later *128*
writing to the client's address space *128*
microkernel *100*
mktime() 167
MMU *35*
defined *383*
modularity due to message passing *100*
mount connect function handler *277*
mountpoint
creating *232*, *234*
registering *232*, *234*
union *217*
mq_open() 234
mq_receive() 234
`mqueue` *234*
MsgDeliverEvent() 111, *119*, *122*, *138–139*, *148*, *291*, *353*, *385*
breaking send hierarchy *119*
networked case *147–148*
special use of receive ID *139*
MsgError() 111, *118*, *123*
errno 123, *250*
MsgInfo() 120, *143*
MsgRead() 111, *126*, *128*, *134*, *148*, *315*
networked case *147–148*
offset parameter *128*
MsgReadv() 111
example *133*
MsgReceive() 111, *115–117*, *120*, *122–123*, *125–126*, *132–135*, *137–138*, *140*, *142–143*, *145*, *148*, *151*, *153*, *170*, *176*, *201*, *219*, *342*, *385*
example *116*, *126*, *133*, *170*
networked case *147–148*
priority inheritance *153*
relationship of parameters to *MsgReply() 115*
MsgReceivePulse() 111, *134–135*, *137–138*, *385*
MsgReceivev() 111, *132*, *137*, *230*, *385*
MsgReply() 111, *115–117*, *122–123*, *128*, *135*, *143*, *148*, *311–312*
errno 123
example *116*, *173–174*
networked case *147–148*
relationship of parameters to *MsgReceive() 115*
MsgReplyv() 111, *133*, *249*, *312*
MsgSend() 111, *113–114*, *116–117*, *122–123*, *125*, *131*, *133–135*, *140*, *185*, *302*
EINTR *140*
example *113*, *129*
MsgSend() family *113*
guide to variants *133*
MsgSendnc() 111, *133*
MsgSendsv() 111, *133–134*
MsgSendsvnc() 111, *133–134*
MsgSendv() 96, *111*, *133*, *140*, *311*, *353*
example *130*
MsgSendvnc() 111, *133*
MsgSendvs() 111, *133*, *140*
MsgSendvsnc() 111, *133*
MsgVerifyEvent() 139
MsgWrite() 111, *123*, *126*, *128*, *134*, *148*, *312*
networked case *147–148*
offset parameter *128*
MsgWritev() 111, *133*, *312*
offset parameter *133*
MT_TIMEDOUT *173*
multipart messages*, See* Message passing
mutex
analogy *25*
defined *383*
example *90*
forksafe *44*
MUTEX *32*
mutual exclusion*, See* mutex

N

name space*, See* pathname space
name_attach() 124, *345*
name_close() 124, *345*
name_detach() 124, *345*
name_open() 124, *345*
nanospin() 157
ND*, See* node descriptor
ND_LOCAL_NODE *150*
netmgr_remote_nd() 149
example *150*
netmgr_strtond() 149
example *150*
network
data transfer *118*
determining how much data should have been transferred *148*

network *(continued)*
determining how much data was transferred *148*
distributed architecture *66*
message passing *105*, *145*
ConnectAttach() differences *147*
detailed analysis *146*
differences from local *147*
MsgDeliverEvent() differences *147–148*
MsgRead() differences *147–148*
MsgReceive() differences *147–148*
MsgReply() differences *147–148*
MsgWrite() differences *147–148*
name resolution *146*
overhead *147*
remote name resolution *146*
message passing transparency *105*
netmgr_remote_nd() 149
netmgr_strtond() 149
node descriptor *112*
of local node *112*
node descriptor of client *121*
of SMP systems *107*
server *108*
using message passing to distribute work *109*
versus shared memory *66*
Neutrino
defined *384*
`nice` *35–36*
node descriptor *112*
`<sys/netmgr.h>` *149*
characteristics *149*
contained in `struct _msg_info` *150*
conversion from symbolic name *149*
example *150*
how to find *123*
how to pass within network *149*
message passing *124*
obtaining remote *149*
of local node *112*
process manager *213*
receiving node's for transmitting node's *150*
representation of remote *150*
transmitting node's for receiving node's *150*
node ID *148*
defined *149*
not network unique *149*
of self *149*

O

O_DSYNC *246*
O_RDONLY *241*
O_RDWR *241*
O_RSYNC *246*
O_SYNC *246*
O_WRONLY *213*, *241*
object oriented design via message passing *106*
OCB *229*
allocating *324*
defined *384*
extended *322*
monitoring *323*
one-shot timers *160*, *167*
example *167*
open connect function handler *231*, *235–236*, *259*, *278*
open connection function handler *323*
open context block, *See* OCB
open file descriptor I/O function handler *280*
open() 104–105, *124–125*, *146–147*, *149*, *211–217*, *219*, *229*, *345*, *353*, *382*, *384*, *386*
implementation *212*
operating system
double standard in conventional *105*
message passing vs. traditional *104–105*
microkernel *100*
process
background *35*
creating *36–37*, *42*
out() 206*

P

P_NOWAIT *41–42*
P_NOWAITO *41*
P_OVERLAY *41*
P_WAIT *41*
PATH 40–41
pathconf I/O function handler *282*
pathname
pollution *124*
registering *218*
resolving *213*
pathname space *212*
and `procnto` *212*
defined *212*

© 2020, QNX Software Systems Limited

PDP-8
  and QNX Neutrino *384*
  defined *384*
periodic timer *160*, *167*, *384*
  example *167*
  power saving *185*
  server maintenance *176*
  servers *168*
    See also repeating timer
philosophy of QNX Neutrino *106*
physical address
  defined *385*
`pidin` *53*, *102–103*
platforms
  PDP-8 *384*
polling
  for completion of thread *184*
  timer *176*
POOL_FLAG_EXIT_SELF *83*
POOL_FLAG_USE_SELF *84–85*
pool, threads, *See* thread
POSIX
  signals *177*
POSIX thread, *See* thread
POSIX_SPAWN_EXPLICIT_CPU *45*
`posix_spawn_file_actions_t` *45*
POSIX_SPAWN_SETPGROUP *45*
posix_spawn() family *36–37*, *45–46*
`posix_spawnattr_t` *45*
power I/O function handler *283*
power saving *185*
pread() 311
printf() 43
priority
  boosting *152*
  inversion *151*
  message passing *125*
  thread analogy *25*
priority inheritance *151*
  undoing *153*
priority inversion *151*
  defined *151*
  example *151*
  fixed by priority inheritance *151*
  solution *152*
  starving CPU *152*
process
  abstraction *65*
  aid to maintainability *34*
  aid to reliability *35*
  and threads *65*
  associating with interrupt handler *197*
  background *35*
  child *43*
  consisting of threads *34*
  context-switch *35*
  coupling *65*
  creating *36–37*, *42*
  creating from program *36*
    exec() family *36–37*
    fork() 36, *42*
    posix_spawn() family *36–37*, *45*
    spawn() family *36–37*
    system() 36
  decoupling *66*
  decoupling of design *34*
  defined *385*
  distributability *66*
  execution time, monitoring *180*
  fork() 43
  in system *34*
  multithreaded *24*
  mutex *80*
  network distributed *66*
  parent *42–43*
  scalability *66*
  shared memory *65*
  single-threaded *24*
  starting *35*
  starting from shell *35*
  thread *24*, *65*
process ID *112*
  getppid() 124
  how to find *123*
  message passing *124*
  process manager *213*
process IDs, *See* PIDs
process manager
  channel ID *213*
  finding *213*
  node descriptor *213*
  process ID *213*
processing interrupts *190*
procmgr_ability() 234

PROCMGR_AID_INTERRUPT *194*
PROCMGR_AID_INTERRUPTEVENT *194*
PROCMGR_AID_PATHSPACE *234*
PROCMGR_AID_PUBLIC_CHANNEL *234*
PROCMGR_AID_SIGEV_THREAD *163*, *201*
`procnto` *103*, *212*
producer
  and consumer *71*
    state analysis *73*
  and consumer using condvars
    example *74*
pthread_atfork() 44, *46*
pthread_attr_destroy() 48–49
pthread_attr_getdetachstate() 48
pthread_attr_getguardsize() 48
pthread_attr_getinheritsched() 48
pthread_attr_getschedparam() 48
pthread_attr_getschedpolicy() 48
pthread_attr_getscope() 48
pthread_attr_getstack() 48
pthread_attr_getstackaddr() 48
pthread_attr_getstacklazy() 48
pthread_attr_getstackprealloc() 48
pthread_attr_getstacksize() 48
pthread_attr_getsuspendstate_np() 48
pthread_attr_init() 48–49
pthread_attr_set() family *49*
pthread_attr_setdetachstate() 48–49
pthread_attr_setguardsize() 48
pthread_attr_setinheritsched() 48–49, *51*
pthread_attr_setschedparam() 48–49, *51*
pthread_attr_setschedpolicy() 48–49, *51*
pthread_attr_setscope() 48, *50*
pthread_attr_setstack() 48, *50*
pthread_attr_setstackaddr() 48, *50*
pthread_attr_setstacklazy() 48, *50*
pthread_attr_setstackprealloc() 48, *50*
pthread_attr_setstacksize() 48, *50*
pthread_attr_setsuspendstate_np() 48–49
`pthread_attr_t` *47*
  defined *47*
pthread_barrier_init() 58
  example *58*
pthread_barrier_wait() 56, *58*, *60*, *381*
  example *58*
pthread_cancel() 96, *134*
pthread_cond_broadcast() 76
pthread_cond_signal() 76
  example *74*
`pthread_cond_t` *76*
pthread_cond_wait() 74, *76*, *79*
  example *74*
pthread_create() 46, *49*, *52*, *57*, *341*
  example *52*, *54*, *58*, *74*
PTHREAD_EXPLICIT_SCHED *51–52*
pthread_join() 49, *56–58*, *60*, *96*, *183–184*
  example *56*, *183*
  explanation *57*
  non-blocking *184*
  timeout *184*
  with timeout *183*
pthread_mutex_destroy() 90
pthread_mutex_init() 90
pthread_mutex_lock() 76, *80*, *341*
  example *74*, *90*
pthread_mutex_timedlock() 181
pthread_mutex_unlock() 76, *80*
  example *74*, *90*
pthread_rwlock_destroy() 69, *93*
pthread_rwlock_init() 69, *93*
pthread_rwlock_rdlock() 69, *93*
`pthread_rwlock_t` *69*
pthread_rwlock_tryrdlock() 69
pthread_rwlock_unlock() 70, *93*
pthread_rwlock_wrlock() 69, *93*
pthread_rwlockattr_destroy() 69
pthread_rwlockattr_getpshared() 69
pthread_rwlockattr_init() 69
pthread_rwlockattr_setpshared() 69
pthread_setschedparam() 26, *153*
pthread_setschedprio() 26, *153*
pthread_sleepon_broadcast() 74, *76*
pthread_sleepon_lock() 71, *76*
  example *71–72*
pthread_sleepon_signal() 73–74, *76*
  example *72*
pthread_sleepon_unlock() 71, *76*
  example *71–72*
pthread_sleepon_wait() 71–72, *74*, *76*
  example *72*
PTHREAD_STACK_LAZY *51*
PTHREAD_STACK_NOTLAZY *51*
`pthread_t` *46*, *57*
pthreads, defined *385*

© 2020, QNX Software Systems Limited

pulse
_PULSE_CODE_UNBLOCK *137*
content *135*
defined *135*, *385*
example *135*, *168*, *170–171*, *173–174*
excluding messages *137*
functions
pulse_attach() 226
how to tell from messages *176*
MsgReceive() 135
MsgReceivePulse() 135
payload content *136*
POSIX *171*
range of *code* member *136*
receiving *135*
receiving pulses only *138*
special *185*
`struct sigevent` *185*
synthetic unblock *141*
timeout example *168*, *170–171*, *173–174*
timers *160*
using the *code* member *136*
using the *value* member *136*
versus signals *164*
pulse_attach() 223, *226*
pulse_detach() 223

Q

Qnet *20*, *119*, *146–148*, *226*
QNX
advantages of architecture *15*
anecdote *15*
applications *15*
history of *16*
on 8088 CPUs *16*
QNX 2 *16*
QNX 4 *16*
QNX Neutrino
philosophy *106*
QNX Software Systems *385*
website *360*
qnx_name_attach() (QNX 4) *345*
qnx_name_locate() (QNX 4) *345*
qnx_proxy_attach() (QNX 4) *353*
QSS, defined *385*
Quantum Software Systems Ltd. *16*
queue
RUNNING *160*
timer *157*
timer queue *185*
QUNIX *15–16*

R

read I/O function handler *235–236*, *283–284*, *308*, *312–314*, *316*, *326–328*
example *308*
read link connect function handler *285*
read() 125, *147*, *211*, *217*, *219*, *221*, *231*, *348–349*, *352*, *386*
readers/writer locks, *See* synchronization
READY *32*
realtime
interrupts *188*
priority inversion *151*
realtime clock *96*
getting and setting *177*
interrupts *96*
receive ID *116*, *122*
content *122*
defined *385*
duplication *140*
example of use *116*
MsgReply() 116
special use *139*
when valid *122*
receive-blocked *102*
diagram *102*
Receive() (QNX 4) *342*, *354*
receiving a message
defined *385*
register for notification I/O function handler *289*
registering
pathname *218*
relative timer *160*, *166*, *185*
defined *385*
example *167–168*, *170–171*, *173–174*
rename connect function handler *287*
rename() 219
rendezvous
and thread synchronization *58*
repeating timer, defined *385*
reply-blocked *102–103*
diagram *102*

reply-driven model *108*, *123*
    example *109*
    important subtlety *109*
Reply() (QNX 4) *342*
replying to a message, defined *386*
resize file I/O function handler *292*
resmgr, *See* resource manager
resmgr_attach() *222–223*, *226*, *231–233*, *235*, *253*, *308*
`resmgr_attr_t` *225–226*, *231*
resmgr_bind_ocb() *280*
`resmgr_connect_funcs_t` *225*, *227*
`resmgr_context_t` *225–230*, *249*
resmgr_detach() *223*
`RESMGR_HANDLE_T` *227–228*
`resmgr_io_funcs_t` *225*, *227–228*
resmgr_msgread() *223*, *315*
resmgr_msgreadv() *223*, *231*, *265*, *304*, *315*
resmgr_msgwrite() *223*
resmgr_msgwritev() *223*, *265*
`RESMGR_OCB_T` *229*
resmgr_open_bind() *223*, *237*
`resmgr.h`
    include after `iofunc.h` *228–229*
    include before `dispatch.h` *87*
resolution of timebase
    adjusting *179*
resource manager *210*
    `/dev/null` *220*
    access control lists (ACLs) *246*
    advanced topics *322*
    allocating OCBs *324*
    and *fork()* *46*
    as a means of advertising ND/PID/CHID *124*
    binding mount structure *323*
    blocking *222*, *326*
    characteristics *211*
    client *212*
        summary *217*
    clients *211*
    combined messages *237–239*
    connecting *212*
    connection ID *211*, *213*
    constants
        _FTYPE_ANY *234*
        _FTYPE_MQUEUE *234*
        _IO_ACL *253*
resource manager *(continued)*
    constants *(continued)*
        _IO_CHMOD *255*
        _IO_CHOWN *256*
        _IO_CLOSE *259*, *328*
        _IO_CONNECT *261*, *273*, *277*, *279*, *285*, *287*, *299*, *328*
        _IO_CONNECT_COMBINE *279*
        _IO_CONNECT_COMBINE_CLOSE *238*, *279*
        _IO_CONNECT_LINK *261*
        _IO_CONNECT_MKNOD *273*
        _IO_CONNECT_MOUNT *277*
        _IO_CONNECT_OPEN *279*
        _IO_CONNECT_READLINK *285*
        _IO_CONNECT_RENAME *287*
        _IO_CONNECT_UNLINK *299*
        _IO_DEVCTL *263*, *317*
        _IO_DUP *267*
        _IO_FDINFO *268*
        _IO_FLAG_RD *241*
        _IO_FLAG_WR *241*
        _IO_LSEEK *293–294*, *328*
        _IO_MMAP *275*
        _IO_MSG *302*
        _IO_NOTIFY *289*
        _IO_NOTIFY64 *289*
        _IO_OPENFD *281*
        _IO_READ *283–284*, *308*, *328*
        _IO_READ64 *283*
        _IO_SPACE *292*
        _IO_STAT *295*
        _IO_SYNC *271*
        _IO_UTIME *257*
        _IO_UTIME64 *257*
        _IO_WRITE *303*
        _IO_WRITE64 *303*
        _IO_XTYPE_NONE *311*
        _IO_XTYPE_OFFSET *284*, *311–313*, *316*
        _IO_XTYPE_READDIR *284*
        _IOFUNC_NFUNCS *247*
        _POSIX_DEVDIR_FROM *264*
        _POSIX_DEVDIR_TO *264*
        _RESMGR_DEFAULT *249*, *265*, *320*
        _RESMGR_FLAG_AFTER *233–234*
        _RESMGR_FLAG_BEFORE *233–234*
        _RESMGR_FLAG_DIR *233*, *327*, *330*
        _RESMGR_FLAG_FTYPEALL *233*
        _RESMGR_FLAG_FTYPEONLY *233*

© 2020, QNX Software Systems Limited

resource manager *(continued)*
constants *(continued)*
_RESMGR_FLAG_OPAQUE *233*
_RESMGR_FLAG_SELF *233*
_RESMGR_NOREPLY *249*, *312*, *326*
DCMD_ALL_SETFLAGS *246*
F_ALLOCSP *293*
F_FREESP *293*
IOFUNC_ATTR_ATIME *311*, *316*
IOFUNC_ATTR_DIRTY_TIME *311*
IOFUNC_ATTR_MTIME *316*
IOFUNC_MOUNT_32BIT *246*
IOFUNC_MOUNT_FLAGS_PRIVATE *246*
IOFUNC_OCB_FLAGS_PRIVATE *242*
IOFUNC_OCB_MMAP *242*
IOFUNC_OCB_MMAP_UNIQUE *242*
IOFUNC_OCB_PRIVILEGED *242*
IOFUNC_OCB_T *322*
IOFUNC_PC_ACL *246*
IOFUNC_PC_CHOWN_RESTRICTED *246*, *256*
IOFUNC_PC_LINK_DIR *246*
IOFUNC_PC_NO_TRUNC *246*
IOFUNC_PC_SYNC_IO *246*
O_DSYNC *246*
O_RSYNC *246*
O_SYNC *246*
SH_COMPAT *242*
SH_DENYNO *242*
SH_DENYRD *242*
SH_DENYRW *242*
SH_DENYWR *242*
context blocks *223*
cookbook *306*
custom *225*
defined *210*, *386*
design *214*
device numbers and inodes *247–248*
domain of authority *218*
example *306*
device control I/O function handler *316*
open connect function handler *235*
read I/O function handler *308–309*
returning data to a client *308–309*
write I/O function handler *313*
extended OCB *322*
extending attributes *325*
file descriptor *213*
file descriptors *211*

resource manager *(continued)*
file streams *211*
filesystem example *210*
finding *212–213*
functions
_IO_SET_WRITE_NBYTES() 315–316
ACL I/O handler *253*
change file mode I/O handler *255*
Change ownership I/O handler *256*
change timestamps I/O handler *257*
close duplicate OCB I/O handler *258*, *267*
close OCB I/O handler *251*, *259–260*
connect *227*
create file link connect handler *261*
custom handlers *235*
default *229*
default handlers *227*, *235*
devctl() 246
device control I/O handler *235–236*, *263*, *316–317*, *320–321*
dispatch_create() 307
duplicate a file descriptor I/O handler *267*
fcntl() 246
file descriptor information I/O handler *268*
File locking I/O handler *269*
flush data I/O handler *271*
I/O *228*
iofunc_ability_check() 255–256, *279–280*
iofunc_acl_default() 253
iofunc_acl() 253
iofunc_chmod_default() 255
iofunc_chmod() 255
iofunc_chown_default() 256
iofunc_chown() 256
iofunc_close_dup_default() 259
iofunc_close_dup() 259
iofunc_devctl_default() 263, *320*
iofunc_devctl_verify() 263
iofunc_devctl() 263, *265*
iofunc_func_init() 227, *229*, *235*, *253*, *306*, *308*, *320*
iofunc_link() 261
iofunc_lock_default() 244, *269*
iofunc_lock_ocb_default() 251, *272*
iofunc_lseek_default() 293
iofunc_lseek() 293
iofunc_mknod() 273
iofunc_mmap_default() 244, *275*

resource manager *(continued)*
functions *(continued)*
iofunc_mmap() 275
iofunc_notify_remove_strict() 289
iofunc_notify_remove() 289
iofunc_notify_trigger_strict() 289
iofunc_notify_trigger() 289, *291*
iofunc_notify() 289
iofunc_ocb_attach() 279–280
iofunc_ocb_calloc() 276, *322*
iofunc_ocb_free() 322
iofunc_open_default() 279
iofunc_open() 279
iofunc_openfd_default() 280
iofunc_openfd() 280
iofunc_pathconf_default() 282
iofunc_pathconf() 282
iofunc_read_default() 283
iofunc_read_verify() 283, *310*, *314*
iofunc_readlink() 285
iofunc_rename() 287
iofunc_space_verify() 292
iofunc_stat_default() 295, *312*
iofunc_stat_format() 295–296
iofunc_stat() 295
iofunc_sync_default() 271
iofunc_sync_verify() 246, *271*, *304*
iofunc_sync() 271
iofunc_unblock_default() 298
iofunc_unblock() 297–298
iofunc_unlink() 299
iofunc_unlock_ocb_default() 251, *301*
iofunc_utime_default() 257
iofunc_utimes() 257
iofunc_write_default() 303
iofunc_write_verify() 303–304, *314*
lock OCB I/O handler *272*, *301*
make filesystem node connect handler *273*
map memory I/O handler *275*
mount connect handler *277*
open connect handler *231*, *235–236*, *259*, *278*
open connection handler *323*
open file descriptor I/O handler *280*
pathconf I/O handler *282*
power I/O handler *283*
read I/O handler *235–236*, *283–284*, *308*, *312–314*, *316*, *326–328*
read link connect handler *285*

resource manager *(continued)*
functions *(continued)*
register for notification I/O handler *289*
rename connect function handler *287*
resgmr_open_bind() 237
resize file I/O handler *292*
resmgr_attach() 226, *231–233*, *235*, *308*
resmgr_bind_ocb() 280
resmgr_msgread() 315
resmgr_msgreadv() 231, *265*, *304*, *315*
resmgr_msgwritev() 265
set file position I/O handler *293*, *328*
shutdown I/O handler *295*
stat I/O handler *295*, *311*
unblock connect handler *296*, *298*, *322*
unblock I/O handler *297*
unlink connection handler *299*
unlock OCB I/O handler *301*
user-defined message I/O handler *302*
write I/O handler *235–236*, *284*, *303*, *313*
future expansion capabilities *228*
gate keeper *236*
handler routines *249*
context *249*
messages *251*
handlers
connect functions *253*
I/O functions *253*
unblocking *253*
handling directories *214*
header as first part of message *127*
initializing a connection *236–237*
internal context *222*
library *220–221*, *225*
base layer *223*
POSIX layer *223*
macros
_IO_SET_CONNECT_RET() 278, *280*
_IO_SET_FDINFO_LEN() 269
_IO_SET_PATHCONF_VALUE() 282
_IO_SET_READ_NBYTES() 284
_IO_SET_WRITE_NBYTES() 304
_RESMGR_ERRNO() (deprecated) *250*
_RESMGR_NPARTS() 249, *321*
_RESMGR_PTR() 250, *321*

© 2020, QNX Software Systems Limited

resource manager *(continued)*
macros *(continued)*
_RESMGR_STATUS() *254–256*, *258–259*, *261*, *263*, *266–267*, *270*, *272–274*, *276*, *281*, *286*, *288*, *291*, *293–294*, *296–298*, *300–303*
message passing *104*
messages *211*, *226*
_IO_DEVCTL *226*
_IO_MSG *226*
combine *273*
combined *250*
connect *213*, *219*, *237*
creating custom *226*
handling *222*
I/O *219*
other *219*
processing *250*
mountpoints *232*, *234*
multiple entries *218*
OCB monitoring *323*
ordering *234*
ordering in pathname space *234*
outcalls *236*
overriding allocation functions *322*
pathname *218*
POSIX layer *224*, *240*
program flow *236*
receiving messages *219*
registering *218*, *232*, *234*
resolving *216*, *218*
returning directory entries *327*
serial port example *210*
setting the *iov* size *249*
skeleton *306*
structure *231*
structures *225*
io_acl_t *253*
io_chmod_t *255*
io_chown_t *256*
io_close_t *259–260*
io_devctl_t *263*
io_dup_t *267*
io_link_extra_t *262*
io_link_t *262*
io_lock_t *270*
io_lseek_t *294*

resource manager *(continued)*
structures *(continued)*
io_mknod_t *274*
io_mmap_t *275*
io_mount_t *277*
io_msg_t *302*
io_notify_t *289*
io_open_t *279*
io_openfd_t *281*
io_pathconf_t *282*
io_read_t *283*
io_readlink_t *285*
io_rename_extra_t *287*
io_rename_t *287*
io_spaced_t *292*
io_stat_t *295*
io_sync_t *271*
io_unlink_t *299*
io_utime_t *257*
io_write_t *303*
iofunc_attr_t *240*, *242–245*
iofunc_mount_t *240*, *243*, *246–247*
iofunc_ocb_t *240–242*
POSIX layer *240*
resmgr_attr_t *225–226*, *231*
resmgr_connect_funcs_t *225*, *227*
resmgr_context_t *225–231*, *249*
RESMGR_HANDLE_T *227–228*
resmgr_io_funcs_t *225*, *227–228*
RESMGR_OCB_T *229*
struct _io_acl *253*
struct _io_acl_reply *253*
struct _io_chmod *255*
struct _io_chown *256*
struct _io_close *259–260*
struct _io_connect *251–252*, *262*, *274*, *277*, *279*, *285*, *287*, *299*
struct _io_connect_ftype_reply *262*, *274*, *277*, *279*, *285*, *287*, *299*
struct _io_connect_link_reply *262*, *274*, *277*, *279*, *285*, *287*, *299*
struct _io_devctl *263*
struct _io_devctl_reply *263*
struct _io_dup *267*
struct _io_lock *270*
struct _io_lock_reply *270*
struct _io_lseek *294*

resource manager *(continued)*
structures *(continued)*
`struct _io_mmap` *275*
`struct _io_mmap_reply` *275*
`struct _io_msg` *302*
`struct _io_notify` *289*
`struct _io_notify_reply` *289*
`struct _io_openfd` *281*
`struct _io_pathconf` *282*
`struct _io_read` *283*
`struct _io_space` *292*
`struct _io_stat` *295*
`struct _io_sync` *271*
`struct _io_utime` *257*
`struct _io_write` *303*
`struct dirent` *284*, *327*
writing *225*
round robin
defined *386*
round-robin scheduling *29*
RR, *See* round-robin
rsrcdbmgr_devno_attach() 247
RTC
getting and setting values *177*
synchronizing to current time of day *178*
RUNNING *32*
and SMP *32*
rwlock
example *93*

S

scalability *66*
due to modularity *100*
over network of SMP *110*
scatter/gather
defined *386*
operation *131*
SCHED_FIFO *52*
sched_get_priority_max() 29
sched_get_priority_min() 29
SCHED_OTHER *52*
SCHED_RR *52*
sched_rr_get_interval() 30
sched_yield() 30
scheduling *96*
algorithms *29*
FIFO *29*
scheduling *(continued)*
algorithms *(continued)*
RR *29*
events in the future using timers *160*
faults *96–97*
FIFO
defined *383*
hardware interrupts *96*, *157*
impact of interrupts *189*
interrupts *96*, *188*
kernel calls *96*, *157*
one shot events in the future *160*
other hardware *96*
periodic events using timers *160*
priority zero *52*
round robin
defined *386*
SCHED_FIFO *52*
SCHED_OTHER *52*
SCHED_RR *52*
the realtime clock *96*
thread creation *48*
timers *96*
scheduling scope *50*
SchedYield() 30
SEM *32*
sem_post() 386
sem_wait() 386
semaphore
defined *386*
in analogy *26*
send hierarchy
avoiding deadlock *119*
breaking *119*, *138*
implementation *138*
with *MsgDeliverEvent() 139*
defined *387*
designing *119*
`struct sigevent` *138*
thread *138*
send-blocked *102–103*
diagram *102*
send-blocked state
when abnormal *103*
when normal *103*
send-driven model *108*
example *109*

© 2020, QNX Software Systems Limited

send-driven model *(continued)*
important subtlety *109*
Send() (QNX 4) *342*, *348*
sending a message
defined *386*
server
acting on unblock pulse *141*
assumptions about client data area *126*
authentication of client *120*
basic operation *114*
being informed of client unblock *185*
binding of client *122*
boosting priority *152*
busy *102*
channels *119*
class of service *120*
client buffer size *125*
client connecting to
diagram *114*
client priority *121*
clients with mismatched buffer sizes *125*
creating a channel *114*
defined *387*
delivering event to client *139*
endian flags *121*
filesystem example *125*
finding *124*
by name *124*
global variable *124*
global variables *124*
resource manager *124*
well-known files *124*
finding out who sent message *120*
framework *116*
general flow *122*
handling big messages *125*
how to handle large transfers *128*
identifying messages *118*
ignoring unblock pulse *141*
limiting transfer size *117*
logging of client *120*
mixing multithreaded and server/subserver *108*
multiple requests *122*
multithreaded *110*
network distributed *108*
node descriptor of client *121*
not replying to client *122*, *176*
notifying client *138*
server *(continued)*
periodic timers *168*
phases of message passing *115*
receive ID *116*
reply blocked client *103*
replying to multiple clients *176*
server connection ID *121*
server/subserver *107–108*
delegation of work *108*
SMP *108*, *110*
state transition
diagram *102*
state transitions *102*
storing the `struct sigevent` *139*
thread pool *80*
thread pools *107*
timeouts *168*
unblock pulse handling *141–142*
unblocked by client *116*
unblocking
client *116*
using *MsgInfo()* *120*
using *MsgReceive()* *120*
using pulses for timeouts *164*
using signals for timeouts *164*
verifying validity of event *139*
writing a header later *128*
server/subserver *123*
analogy *108*
diagram *109*
example *109*
implementation description *109*
message passing *107*
set file position I/O function handler *293*, *328*
SETIOV() macro
defined *130*
example *132*
setting the time *177*
setuid() *194*
setupPulseAndTimer() *170–171*
SH_COMPAT *242*
SH_DENYNO *242*
SH_DENYRD *242*
SH_DENYRW *242*
SH_DENYWR *242*
shared memory *65*
versus network *66*

sharing interrupts *192–193*
    diagram *192*
shutdown I/O function handler *295*
SIGALRM *162*, *177*
SIGEV_INTR *161*, *164*, *201–202*, *205*
    and interrupts *201*, *205*
    and *InterruptWait() 202*
    and `struct sigevent` *161*
SIGEV_INTR_INIT() (macro) *164*
sigev_notify_function() 163
SIGEV_PULSE *161–163*, *171*, *201*
    and `struct sigevent` *161*, *171*
SIGEV_PULSE_INIT() 171
SIGEV_PULSE_INIT() (macro) *163*
    example *171*
SIGEV_PULSE_PRIO_INHERIT *163*, *171*
SIGEV_SIGNAL *161–163*, *201*
    and interrupts *201*
    and `struct sigevent` *161–162*
SIGEV_SIGNAL family *162*
SIGEV_SIGNAL_CODE *161–163*
    and `struct sigevent` *161–162*
SIGEV_SIGNAL_CODE_INIT() (macro) *163*
SIGEV_SIGNAL_INIT() (macro) *163*
SIGEV_SIGNAL_THREAD *161–163*
    and `struct sigevent` *161–162*
SIGEV_SIGNAL_THREAD_INIT() (macro) *163*
SIGEV_THREAD *161–163*, *201*
    and interrupts *201*
    and `struct sigevent` *161*
SIGEV_THREAD_INIT() (macro) *164*
    example *177*
SIGEV_UNBLOCK *161*, *163*, *183–184*
    and `struct sigevent` *161*, *183*
    example *183*
SIGEV_UNBLOCK_INIT() (macro) *163*, *183–184*
    example *183*
signal
    defined *387*
    floating-point operations aren't safe to use in handlers *164*
    SIGALRM *177*
    SIGUSR1 *177*
    `struct sigevent` *185*
    timers *160*, *177*
    versus pulses *164*
SIGSEGV *51*
SIGUSR1 *177*
sigwait() 164
sleep() 33, *96*, *156–157*, *180*
    bad implementation *156*
sleepon locks, *See* synchronization
slowing down time *178*
SMP *28*, *100*
    application *60*
    atomic operations *64*
    coding for SMP or single processor *56*
    concurrency *64*
    condvars *79*
    constraints *61*
    creating enough threads *54*
    in a networked system *107*
    interrupts *64*, *207*
    message passing *107*
    multiple threads *60*
    scalability *110*
    soaker thread *62*
    STATE_RUNNING *32*
    thread pools *107*
    threads *54*
    timing
        diagram *61–62*
    tips *64*
    underutilization *63*
    utilization *62*
soaker thread *62*
SPAWN_NOZOMBIE *41*
spawn() 37–38, *42*
spawn() family *36–38*, *40–41*
spawnl() 37–38
spawnle() 37–38
spawnlp() 37–38
spawnlpe() 37–38
spawnp() 37–38
spawnv() 37–38
spawnve() 37–38
spawnvp() 37–38
spawnvpe() 37–38
speeding time up *178*
stack
    for thread *47*
    postmortem analysis *51*
stat I/O function handler *295*, *311*
stat() 221
STATE_CONDVAR *32*, *74*

© 2020, QNX Software Systems Limited

STATE_DEAD *32*
STATE_INTR *32–33*
STATE_JOIN *32*
STATE_MUTEX *32*
STATE_NANOSLEEP *32–33*
STATE_NET_REPLY *32*
STATE_NET_SEND *32*
STATE_READY *32*, *96*, *102*, *151–152*, *157*, *159–160*, *188*, *194*, *202*
STATE_READY state *102*
STATE_RECEIVE *32–33*, *102*, *153*, *164*
STATE_RECEIVE state
  diagram *102*
STATE_REPLY *32–33*, *141*, *176*, *185*
STATE_REPLY state *102–103*
  diagram *102*
STATE_RUNNING *32*, *96*, *188*
STATE_SEM *32*
STATE_SEND *32–33*, *138*, *141*, *185*
STATE_SEND state *102–103*
  diagram *102*
STATE_SIGSUSPEND *32*
STATE_SIGWAITINFO *32*
STATE_STACK *32*
STATE_STOPPED *32*
STATE_WAITCTX *32*
STATE_WAITPAGE *32*
STATE_WAITTHREAD *32*
strcmp() 206
strdup() 206
strftime() 167
`struct _clockadjust` *178*
`struct _clockperiod` *179*
  members *179*
`struct _io_acl` *253*
`struct _io_acl_reply` *253*
`struct _io_chmod` *255*
`struct _io_chown` *256*
`struct _io_close` *259–260*
`struct _io_connect` *251–252*, *262*, *274*, *277*, *279*, *285*, *287*, *299*
`struct _io_connect_ftype_reply` *262*, *274*, *277*, *279*, *285*, *287*, *299*
`struct _io_connect_link_reply` *262*, *274*, *277*, *279*, *285*, *287*, *299*
`struct _io_devctl` *263*
`struct _io_devctl_reply` *263*
`struct _io_dup` *267*
`struct _io_lock` *270*
`struct _io_lock_reply` *270*
`struct _io_lseek` *294*
`struct _io_mmap` *275*
`struct _io_mmap_reply` *275*
`struct _io_msg` *302*
`struct _io_notify` *289*
`struct _io_notify_reply` *289*
`struct _io_openfd` *281*
`struct _io_pathconf` *282*
`struct _io_read` *283*
`struct _io_space` *292*
`struct _io_stat` *295*
`struct _io_sync` *271*
`struct _io_utime` *257*
`struct _io_write` *303*
`struct _msg_info` *120*, *143*, *148*
  declaration *150*
  fields in *121*
  flags *148*
  node descriptors *150*
`struct _pulse` *135*, *137*
  declaration *135*
`struct _thread_pool_attr`
  defined *81*
  members *87*
`struct dirent` *284*, *327*
`struct itimerspec` *166*
  defined *166*
  example *171*
  example of filling *167*
  it_interval member *166*
  it_value member *166*
`struct sigevent` *139*, *161–162*, *165–166*, *177*, *183–185*, *191*, *194–195*, *199*, *201–203*, *343*, *353*, *355*, *383*
  creating threads *177*
  defined *161*
  example *171*, *183*
  how to initialize *161*
  interrupts *194–195*, *199*, *201*
  macros for filling *163*
  persistency *191*
  pulse *185*
  pulses *171*
  send hierarchy *138*

struct sigevent *(continued)*
shortcut initialization *184*
sigev_code member *162*
sigev_coid member *162*
SIGEV_INTR *161*
sigev_notify member *161–162*
sigev_notify_attributes member *163*
sigev_notify_function member *163*
sigev_priority member *162*
SIGEV_PULSE *161*, *171*
SIGEV_SIGNAL *161–162*
SIGEV_SIGNAL_CODE *161–162*
SIGEV_SIGNAL_THREAD *161–162*
sigev_signo
and pulses *171*
sigev_signo member *162*
SIGEV_THREAD *161*
SIGEV_UNBLOCK *161*, *183*
sigev_value member *162–163*
signal *185*
timers *161*
struct timespec *166–167*
defined *166*
structures
attribute
defined *381*
FILE *211*, *217*
iov_t *130–131*
posix_spawn_file_actions_t *45*
posix_spawnattr_t *45*
pthread_attr_t *47*
pthread_cond_t *76*
pthread_rwlock_t *69*
pthread_t *46*, *57*
resmgr_attr_t *225–226*, *231*
resmgr_connect_funcs_t *225*, *227*
resmgr_context_t *225–230*
RESMGR_HANDLE_T *227–228*
resmgr_io_funcs_t *225*, *227–228*
RESMGR_OCB_T *229*
resource manager
io_acl_t *253*
io_chmod_t *255*
io_chown_t *256*
io_close_t *259–260*
io_devctl_t *263*
io_dup_t *267*
structures *(continued)*
resource manager *(continued)*
io_link_extra_t *262*
io_link_t *262*
io_lock_t *270*
io_lseek_t *294*
io_mknod_t *274*
io_mmap_t *275*
io_mount_t *277*
io_msg_t *302*
io_notify_t *289*
io_open_t *279*
io_openfd_t *281*
io_pathconf_t *282*
io_read_t *283*
io_readlink_t *285*
io_rename_extra_t *287*
io_rename_t *287*
io_space_t *292*
io_stat_t *295*
io_sync_t *271*
io_unlink_t *299*
io_utime_t *257*
io_write_t *303*
iofunc_attr_t *240*, *242–245*
iofunc_mount_t *240*, *243*, *246–247*
iofunc_ocb_t *240–242*
resmgr_context_t *249*
struct _io_acl *253*
struct _io_acl_reply *253*
struct _io_chmod *255*
struct _io_chown *256*
struct _io_close *259–260*
struct _io_connect *251–252*, *262*, *274*, *277*, *279*, *285*, *287*, *299*
struct _io_connect_ftype_reply *262*, *274*, *277*, *279*, *285*, *287*, *299*
struct _io_connect_link_reply *262*, *274*, *277*, *279*, *285*, *287*, *299*
struct _io_devctl *263*
struct _io_devctl_reply *263*
struct _io_dup *267*
struct _io_lock *270*
struct _io_lock_reply *270*
struct _io_lseek *294*
struct _io_mmap *275*
struct _io_mmap_reply *275*

© 2020, QNX Software Systems Limited

structures *(continued)*
resource manager *(continued)*
`struct _io_msg` *302*
`struct _io_notify` *289*
`struct _io_notify_reply` *289*
`struct _io_openfd` *281*
`struct _io_pathconf` *282*
`struct _io_read` *283*
`struct _io_space` *292*
`struct _io_stat` *295*
`struct _io_sync` *271*
`struct _io_utime` *257*
`struct _io_write` *303*
`struct dirent` *284*, *327*
`struct _msg_info` *120*, *143*, *148*, *150*
chid *121*
coid *121*
flags *121*
msglen *121*
nd *121*
pid *121*
priority *121*
scoid *121*
srcnd *121*
tid *121*
`struct _pulse` *135*, *137*
`struct itimerspec` *166*
it_interval member *166*
it_value member *166*
`struct sigevent` *161–162*, *165–166*, *177*, *184–185*, *191*, *194–195*, *199*, *201–203*, *343*, *353*, *355*, *383*
`struct timespec` *167*
`thread_pool_attr_t` *82*
`union sigval` *136*
sub-second timers *167*
synchronization
association of mutex and condvar *79*
barrier *97*
defined *381*
condition variable *68*, *74*
defined *382*
condvar *97*
versus sleepon *76*
condvar versus sleepon *78*
deadlock
defined *382*
synchronization *(continued)*
example *88*
joining *97*
mutex *97*
across process boundaries *80*
defined *383*
reader/writer lock *68*
rendezvous *58*
semaphore *97*
defined *386*
signal versus broadcast *76*
sleepon
versus condvar *76*
sleepon lock *68*, *70*, *97*
sleepon versus condvar *78*
to termination of thread *56*
using a barrier *58*
synchronizing time of day *178*
synchronous *387*
defined *387*
See also asynchronous
`sys/*.h`, *See* individual files
sysconf() *51*
system
as consisting of processes and threads *34*
system() *36*

T

`tar` *362*
example *362*
TDP (Transparent Distributed Processing) *119*
See also Qnet
technical support *12*, *357*, *360*
beta versions *361*
updates *362*
contacting *360*
describing the problem *360*
be precise *361*
narrow it down *362*
read the manuals *358*
reproduce the problem *362*
training *363*
`telnet` *361*
termination synchronization *56*
tfork() *341*
thread
associating with interrupt handler *197*

thread *(continued)*
barriers *58*
blocking states *29*
concurrent *28*
context switch *28*
coupling *65*
created by timer trigger *177*
creating
attribute structure initialization *49*
detached *47*
example *54*
joinable *47*
on timer expiry *160*
registering exit function *48*
scheduling parameters *48*
specifying scheduling policy *52*
specifying stack *47*
creating on timer expiration *163*
creating via `struct sigevent` *163*
deadlock
defined *382*
defined *387*
design abstraction *28*
example of creation *52*
execution time, monitoring *180*
FIFO scheduling *29*
fork() 46
fundamentals *24*
in mathematical operations *53*
in process *34*
interrupt interaction *207*
interrupt interactions *199*
interrupts *190–191*
message passing *107*, *120*
multiple threads *24*
mutex *25*
operating periodically *156*
`pidin` *53*
polling for completion *184*
pool *80*, *107*
analysis *85*
and SMP *107*
example *84*
functions *82*
message passing *107*
POOL_FLAG_EXIT_SELF *83*
POOL_FLAG_USE_SELF *84*

thread *(continued)*
POSIX *385*, *387*
postmortem stack analysis *51*
preemption *29*
priorities *25*
processes *65*
Processes *24*
pthread_join() 183–184
readers/writer locks *68*
readied by timer *185*
readying via message pass *102*
resumption *29*
RR scheduling *29*
scheduling policies *29*
scheduling policy *29*
semaphore *26*
single threads *24*
SMP *54*
and interrupts *64*
concurrency *64*
determining how many threads to create *54*
timing diagram *61–62*
soaker *62*
stack *51*
states
receive-blocked *102*
receive-blocked diagram *102*
reply-blocked *102–103*
reply-blocked diagram *102*
send-blocked *102–103*
send-blocked diagram *102*
STATE_READY *102*
STATE_RECEIVE *102*
STATE_RECEIVE diagram *102*
STATE_REPLY *102–103*
STATE_REPLY diagram *102*
STATE_SEND *102–103*
STATE_SEND diagram *102*
synchronizing to termination of *56*
utilizing SMP *62*
where to use *53*, *64*
thread pool
message passing *138*
`thread_pool_attr_t` *82*
thread_pool_control() 81
thread_pool_create() 81, *83–84*
example *84*

© 2020, QNX Software Systems Limited

thread_pool_destroy() 81
thread_pool_limits() 81
THREAD_POOL_PARAM_T *87*
thread_pool_start() 81, *83–85*
    example *84*
thread_pool() family *107*
ticksize *30*
time
    adjusting forwards or backwards *178*
    adjusting gradually *178*
    discontinuities in flow *178*
    retarding flow of *178*
    synchronizing current time of day *178*
time() 167
timebase *179*
timeout *183*
    and kernel states *183*
    arming *183*, *185*
    clearing *183*
    kernel timeouts *176*
    pthread_join() 184
    server-driven *176*
    server-maintained *168*
        example *168*, *170–171*, *173–174*
    triggering *183*
    unblocking
        client *140*
    with *pthread_join() 183*
timeout notification *160*
    pulse *164*
    signal *164*
timer
    <time.h> *166*
    10 millisecond *157*
    absolute *160*, *166*, *185*
        defined *381*
        example *167*
    accuracy *158–159*
    adjusting base timing resolution *179*
    asynchronous nature *159*
    behavior if expired *159*
    changing resolution *159*
    CLOCK_MONOTONIC *165*
    CLOCK_REALTIME *165*
    CLOCK_SOFTTIME *165*
    ClockPeriod() 178
    converting time formats *167*
timer *(continued)*
    creating *165*
    creating a thread *161*, *163*
    creating threads on expiry *160*
    creating threads on trigger *177*
    delivering a pulse *161*
    delivering a signal *161*
    diagram showing big picture *158*
    drift *158*
    flags *166*
    getting and setting the realtime clock *177*
    hardware divider *158*
    hardware divisor *158*
    high-resolution *159*, *179*
    implementation *157*, *160*, *168*, *185*
    improving accuracy *179*
    inactivity shutdown *176*
    jitter *159–160*
        diagram *159*
    kernel timeouts *185*
    limits on base timing resolution *179*
    one-shot *160*, *167*
        example *167*
    periodic *160*, *167*
        and server maintenance *176*
        and servers *168*
        example *167*
    polling *176*
    preemption *159*
    pulse versus signal *164*
    pulses *137*, *160*, *162*, *168*, *170–171*, *173–174*
    putting a thread on hold *157*
    readying a thread *159*, *185*
    relative *160*, *166*, *185*
        defined *385*
        example *167–168*, *170–171*, *173–174*
    repeating
        defined *385*
    resolving *158–159*
    scheduling an event in the future *160*
    sending a signal *162*
        specifying code number *162*
        specifying signal number *162*
    setting type *166*
    SIGALRM *177*
    SIGEV_THREAD *161*
    signals *160*, *177*
        specifying a signal *177*

© 2020, QNX Software Systems Limited

timer *(continued)*
SIGUSR1 *177*
specifying sub-second values *167*
starting *166*
`struct itimerspec` *166*
`struct sigevent` *161*
`struct timespec` *166–167*
timeout notification *160*
by pulse *164*
by signal *164*
TIMER_ABSTIME *166*
timer_create() 165
timer_settime() 166
types *160*, *166–167*
usage examples *168*
using *165*
using pulses with servers *164*
warm-up timer *177*
TIMER_ABSTIME *166*
timer_create() 165–166, *171*, *181*
example *171*
flags argument *166*
signal example *177*
timer_settime() 166–167, *171*
and TIMER_ABSTIME *166*
example *171*
TimerTimeout() 122, *140*, *183–185*, *342*
and CLOCK_REALTIME *183*
example *183–185*
specifying multiple kernel states *185*
timeslice *30*
timestamps *179*
timing
busy wait *157*
fine grained *157*
high accuracy *179*
hogging CPU *157*
using *ClockCycles() 179*
tips
broadcast versus signal *76*
SMP gotchas *64*
when to use condvar *79*
when to use sleepon *79*
where to use a thread *53*, *64*
Transparent Distributed Processing (TDP) *119*
See also Qnet
Trigger() (QNX 4) *353*
triggering timeouts *183*
Trunley, Paul *15*
typographical conventions *10*

U

unblock
defined *387*
using a timeout *183*
unblock connect function handler *296*, *322*
unblock connect handler *298*
unblock I/O function handler *297*
unblock_func() 87
union filesystem
defined *217*
union mountpoint
defined *217*
`union sigval` *136*
declaration *136*
unit testing
message passing *106*
unlink connection function handler *299*
unlock OCB I/O function handler *301*
user-defined message I/O function handler *302*

V

virtual address
defined *387*
virtual memory
defined *388*
`volatile` *199*
and interrupts *199*, *208*

W

waitpid() 41
warm-up timer *177*
website, QNX *360*
write I/O function handler *235–236*, *284*, *303*, *313*
example *313*
write() 104, *125–126*, *129*, *349*, *353*, *386*
`www.qnx.com` *360*

© 2020, QNX Software Systems Limited