Realtime Programming for the QNX® Neutrino® RTOS

AdvanTRAK Technologies Pvt Ltd

In this course you'll learn:

- the architecture of the QNX® Neutrino® RTOS
- a bit about your development tools
- how to properly utilise threads where needed
- how to write systems that consist of a number of cooperating components (processes) that talk to one another using various forms of interprocess communication (IPC)
- a bit about writing drivers
- a bit about configuring your target

Agenda

Topics:

QNX Neutrino Architecture

- Overview
- The Microkernel
- The Process Manager
- Scheduling
- SMP
- Resource Managers
- System Library
- Shared Objects

Agenda

Topics:

Processes, Threads & Synchronization

- Threads and Processes
- Thread Creation
- Detecting Thread Termination
- Thread Operations
- Synchronization

Agenda

Topics:

Interprocess Communication

- Native QNX Neutrino Messaging
- Pulses
- Signals
- Event Delivery
- Shared Memory
- Pipes & POSIX Msg Queues
- POSIX fd/fp Based Functions

Agenda

Topics:

Time

- Timing Architecture
- Getting and Setting the System Clock
- Timers
- Kernel Timeouts

Interrupts

- Concepts
- IPC from ISR
- Handler Architecture

Agenda

Topics:

I/O

- Memory Mapping
- Port I/O
- DMA
- PCI

Introduction to Resource Managers

- Overview
- A Simple Resource Manager
- Device Specific and Per-Open Data
- Out-of-Band/Control Messages

Building a Boot Image

- Images & Build files
- File Systems
- Loading

Agenda

Topics:

QNX Momentics Development & Debugging using IDE

- IDE Basics
- Managing C/C++ Projects
- Editing and Compiling
- Running and Debugging
- IDE Tools Demo

QNX® Neutrino® Architecture

Introduction

You will learn:

- the architecture of the QNX Neutrino RTOS
  - how it's different from others
  - what this means
- operating system services and what delivers them
- process and thread models
- how scheduling works

Topics:

- **Overview**
- **The Microkernel**
- **The Process Manager**
- **Scheduling**
- **SMP**
- **Resource Managers**
- **System Library**
- **Shared Objects**
- **Conclusion**

Portability

QNX Neutrino delivers a standards based system in a small form factor:

- POSIX 1003.1-2001
  - Unix, threads, timers, signals, etc
- ANSI C/C++
  - GNU(Genuinely not UNIX) Compiler Chain

Architecture - Executive

In a traditional Real-Time Executive:

– All modules share the same address space and are, effectively, one big program.

Architecture - Monolithic

In a traditional Monolithic kernel OS:

Monolithic Kernel

- The kernel contains the OS kernel functionality and all drivers, so driver development is complex and debugging can be painful.
- Applications are processes in protected memory space, so the kernel is protected from applications and applications are protected from each other.

In the QNX Neutrino OS:

- The OS consists of the microkernel (or just “kernel”) and a set of cooperating processes.
- The processes are separate from the kernel so if something goes wrong in a process it would not affect the kernel.

Processes communicate with each other:

- the OS processes and your processes cooperate using Interprocess communication. Together, the OS and your processes make up one seamless system.
- there are a large variety of types of interprocess communication

Examples of processes are:

- Disk Drivers
  devb-eide, devb-aha2
- Network Stack
  io-net, io-pkt
- Character Drivers
  devc-ser8250, devc-serppc800, devc-con
- GUI components
  Photon, phfontFA, io-graphics
- Bus managers
  pci-raven, devp-pccard
- System daemons
  cron, inetd, mqueue, qconn

So what does this mean?

Trade-offs:

- benefits:
  - resilience and reliability
  - ease of configurability and reconfiguration
  - ease of debugging
  - ease of development
- costs:
  - system overhead
    - more context switches
    - more copies of data

What is a process?

- a program loaded into memory
- identified by a process id, commonly abbreviated as **`pid`**
- owns resources:
  - memory, including code and data
  - open files
  - identity - user id, group id
  - timers
  - and more

Resources owned by one process are protected from other processes

What is a thread?

- a thread is a single flow of execution or control
- a thread has some attributes:
  - priority
  - scheduling algorithm
  - register set
  - CPU mask for SMP
  - signal mask
  - and others

- all its attributes have to do with running code

Threads run in a process:

- a process must have at least one thread
- threads in a process share all the process resources

Threads run code, processes own resources

Processes and threads:

- processes are your "building blocks" components of a system
  - visible to each other
  - communicate with each other
- threads are the implementation detail
  - hidden inside processes

Topics:

- Overview
- → The Microkernel
- The Process Manager
- Scheduling
- SMP
- Resource Managers
- System Library
- Shared Objects
- Conclusion

Kernel

The kernel is special:

- it is the glue that holds the system together
- programs deal with the kernel by using special library routines, called "**kernel calls**", that execute code in the kernel
- most the other sub-systems, including user applications, communicate with each other using the **message passing** provided by the kernel through kernel calls

Kernel

The kernel is the core of your system:

Kernel calls:

- often you'll make kernel calls
- this means you'll be executing code in the kernel for the duration of the call

Kernel calls are pre-emptable:

Kernel - Pre-emption

Kernel operations:

Kernel - Pre-emption

So what does this mean?

More trade-offs:

- benefit: reduce latency
  - respond to new events faster
  - shorter interrupt latency, scheduling latency
- cost: throughput
  - takes more time to restart an interrupted kernel call
  - takes more time to save current state & restart a pre-empted message pass

Kernel - Services

The kernel provides:

Kernel - IPC

The forms of IPC provided by the kernel:

- Messages
  - exchanging information between processes
- Pulses
  - delivering notification to a process
- Signals
  - interrupting a process and making it do something different (usually termination)

Native QNX Neutrino Messages:

Native QNX Neutrino Pulses:

– used for event notification: “something happened”

POSIX Signals:

– interrupt another process

Thread Functions:

- create / terminate threads
- wait for thread completion
- change thread attributes

Thread synchronization methods:

mutex	mutually exclude threads
condvar	wait for a change
semaphore	wait on a counter
rwlock	synchronize reader and writer threads
join	synchronize to termination of a thread
spinlock	busy wait on a memory location
sleepon	like condvars, but dynamically allocated
barrier	wait for predetermined number of threads

QNX® Neutrino®'s Concept of Time:

Interrupt Handling:

All hardware interrupts are vectored to the kernel.

A process can either:

- register a function to be called by the kernel when the interrupt happens
- request notification that the interrupt has happened

Kernel

The kernel:

- can be thought of as a library
  - no processing loop, no `while(1)`
- only runs if invoked by:
  - kernel call
  - interrupt
  - processor fault/exception e.g. illegal instruction, invalid address

Topics:

- Overview
- The Microkernel
- → The Process Manager
- Scheduling
- SMP
- Resource Managers
- System Library
- Shared Objects
- Conclusion

The Process Manager provides:

- packaging of groups of threads together into processes
- memory protection, address space management including shared memory for IPC
- pathname management
- process creation and termination
  - spawn / exec / fork
  - loads ELF executables

Communication with the Process Manager:

– processes communicate with the process manager using IPC

Each process is a collection of resources and one or more threads:

We use a virtual address model:

- each process runs in its **own protected virtual address space**
- **pointers** that you deal with **contain virtual addresses**, not physical
- physically they all share the **same address space**

Virtual Addresses

	System Process (procnto)	User Process #1	User Process #2	User Process #3
x86:	3.5G ↔ 4G	0 ↔ 3.5G	0 ↔ 3.5G	0 ↔ 3.5G
PPC:	0 ↔ 1G	1G ↔ 4G	1G ↔ 4G	1G ↔ 4G
MIPS:	2G ↔ 4G	0 ↔ 2G	0 ↔ 2G	0 ↔ 2G
SH4:	2G ↔ 4G	0 ↔ 2G	0 ↔ 2G	0 ↔ 2G
ARM:	2G ↔ 4G	0 ↔ 32MB	0 ↔ 32MB	0 ↔ 32MB

Process Manager - Pathname Management

When QNX Neutrino starts up, the entire pathname space is owned by **`procnto`**:

Any requests for file or device pathname resolution are handled by **`procnto`**.

Process Manager - Pathname Management

`procnto` allows resource managers to adopt a portion of the pathname space:

Topics:

- Overview
- The Microkernel
- The Process Manager
- → Scheduling
- SMP
- Resource Managers
- System Library
- Shared Objects
- Conclusion

Threads have two basic states:

blocked

- waiting for something to happen
- there are lots of different blocked states depending on what they are waiting for, e.g.:
  - **`REPLY`** blocked is waiting for a IPC reply
  - **`MUTEX`** blocked is waiting for a mutex
  - **`RECEIVE`** blocked is waiting to get a message

ready

- capable of using the CPU
- two main ready states
  - **`RUNNING`** actually using the CPU
  - **`READY`** waiting while someone else is running

Scheduling - Priority

All threads have a priority:

- the priority range is 0 (low) to 255 (high)
- priority matters for ready threads only
- the kernel always picks the highest priority **`READY`** thread to be the one that actually uses the CPU (fully pre-emptive)
  - the thread's state becomes **`RUNNING`**
  - blocked threads don't even get considered
- most threads spend most of their time blocked
  - that is how CPU is shared between threads

Priorities:

Priorities

highest

```
(hardware interrupt handlers)
-----------------------------------------
255
 .
 .
21  devb-eide (one of many of
 .                devb-eide's threads)
 .
11
10  ksh
 9
 .
 .
 1
 0  procnto (idle thread)
```

Priority

lowest

The READY queue

Scheduling algorithms: Round-robin

☞ There is an "other" algorithm that is currently the same as round-robin

Scheduling algorithms: FIFO

Scheduling algorithms: Sporadic

Interrupt Scheduling (preemptive):

Interrupt Scheduling (non-preemptive):

Topics:

- Overview
- The Microkernel
- The Process Manager
- Scheduling
- → SMP
- Resource Managers
- System Library
- Shared Objects
- Conclusion

SMP

SMP:

- is short for Symmetrical MultiProcessor
- means that you are using a board that has more than one processor/CPU tightly coupled
- you don't have to write special code
- requires a different kernel:
  - e.g. **`procnto-smp`**, **`procnto-smp-instr`**, **`procnto-600-smp`**
- on an SMP system, threads of different priorities or multiple FIFO threads of the same priority may execute at the same time

Topics:

- Overview
- The Microkernel
- The Process Manager
- Scheduling
- SMP
- → Resource Managers
- System Library
- Shared Objects
- Conclusion

What is a resource manager?

- a program that looks like it is extending the operating system by:
  - creating and managing a name in the pathname space
  - providing a POSIX interface for clients (e.g. *open()*, *read()*, *write()*, ...)
- can be associated with hardware (such as a serial port, or disk drive)
- or can be a purely software entity (such as **`mqueue`**, the POSIX queue manager)

Locating a Resource Manager

Interactions:

```
fd = open("/dev/ser1",…);
```

1. Client's library (*open()*) sends a "query" message
2. Process Manager replies with who is responsible
3. Client's library establishes a connection to the specified resource manager and sends an open message
4. Resource manager responds with status (pass/fail)

Resource Managers

Further communication is message passing directly to the resource manager:

Other notes:

- this setup allows for a lot of powerful solutions
  - debug “OS” drivers with a high-level (symbolic) debugger
  - distribute drivers across a QNX network
  - export access to your custom driver with a network file system such as NFS (***Network file system***) or CIFS (***Common Internet File System***)
  - provide resiliency or redundancy of OS services
- QSSL (*QNX Software Systems, Ltd*.) supplies a library that provides a lot of useful code to minimise the work needed to write ones

Topics:

- Overview
- The Microkernel
- The Process Manager
- Scheduling
- SMP
- Resource Managers
- → System Library
- Shared Objects
- Conclusion

Library Architecture

Many standard functions in the library are built on kernel calls

- usually this is a thin layer, that may just change the format of arguments, e.g.
  - the POSIX function *timer_create()* calls the kernel function *TimerCreate()*
    - it changes the time values from the POSIX seconds & nanoseconds to the kernel's 64-bit nanosecond representation
- we recommend using the standard calls
  - your code is more portable
  - you use calls that are going to be more familiar to and readable by your developers

But QNX is a microkernel

- so many routines that would be a kernel call, or have a dedicated kernel call in a traditional Unix become a message pass
- they build a message then call *MsgSend()* passing it to a server, e.g.
  - *read()* builds a message then sends it to a resource manager
  - *fork()* builds a message and sends it to the process manager

Library Architecture

Still more functions supply an extra layer on top of something lower level

- e.g. the stdio functions provide local buffering on top of the underlying *read()* and *write()* calls so:
  - if you were wanting to read a byte at a time, *fread(*) would be a good choice as it would locally buffer and only do the message pass every 1000 reads
  - if you were wanting read 64k at a time, though, it would break it down into 64 1K *read()*s, and you would be better calling *read()* directly

Topics:

- Overview
- The Microkernel
- The Process Manager
- Scheduling
- SMP
- Resource Managers
- System Library
- → Shared Objects
- Conclusion

Shared Objects:

- are libraries loaded and linked at run time
- one copy used (shared) by all programs using library
- also sometimes called DLLs
  - shared objects and DLLs use the same architecture to solve different problems

Topics:

- Overview
- The Microkernel
- The Process Manager
- Scheduling
- SMP
- Resource Managers
- System Library
- Shared Objects
- → Conclusion

Conclusion

You learned that:

- QNX Neutrino is a microkernel architecture OS
- most OS services are delivered by cooperating processes
- processes own resources and threads run code
- QNX Neutrino does pre-emptive scheduling
  - only **`READY`** threads are schedulable, blocked threads are not

Processes, Threads & Synchronization

Introduction

You will learn:

- what a process is and what a thread is
- why you'd use multiple threads in a process
- how to create processes and threads and how to detect when they die
- how to synchronize among threads using mutexes, condvars, semaphores, ...

Topics:

→ **Processes and Threads**

Processes

Threads

Synchronization

Conclusion

Processes

What is a process?

- a program loaded into memory
- identified by a process id, commonly abbreviated as **`pid`**
- owns resources:
  - memory, including code and data
  - open files
  - identity - user id, group id
  - timers
  - and more

Resources owned by one process are protected from other processes

What is a thread?

- a thread is a single flow of execution or control
- a thread has some attributes:
  - priority
  - scheduling algorithm
  - register set
  - CPU mask for SMP
  - signal mask
  - and others
- all its attributes have to do with running code

Threads run in a process:

- a process must have at least one thread
- threads in a process share all the process resources

Threads run code, processes own resources

Example - Assembly line

Processes

The processes that monitor and control these devices:

Another advantage to this process model:

– they can be spread out across multiple QNX nodes and still work together:

Process opacity:

- one process should not be aware of the threads in another process
  - threads are an implementation detail of the process that they are in
- why?
  - object oriented design - the process is the object.
  - flexibility in how processes are written - it might use only one thread, it might use multiple threads, the threads may be dynamically created and destroyed as needed, …
  - scalability and configurability - if clients find servers using names then servers can be moved around. Intermediate servers can be added, servers can be put on other nodes of a network, server can be scaled up or down by adding or removing threads

Some examples of multithreaded processes:

- high priority, time-critical thread dedicated to handling hardware requests as soon as they come in; other thread(s) that talk to clients

- pool of worker threads. If one or more threads are busy handling previous requests there are still other threads available for new requests

Process Virtual Address Space

Virtual address space of a process:

Each thread has its own stack. Everything else is shared

Thread stacks:

- each thread's stack has a maximum size but not all of it will necessarily be physically allocated

Topics:

Processes and Threads

Processes

- Creation
- Detecting termination

Threads

Synchronization

Conclusion

Process Creation

There are a number of process creation calls:

- *fork()*
  - create a copy of the calling process
- *exec*()*
  - load a program from storage to transform the calling process
- *spawn()*, *spawn*()*
  - load a new program creating a new process for it

Process Creation - fork()

fork() will create a copy of your process:

- the child will:
  - be an identical copy of the parent
  - start from the *fork()*
  - initially have the same data as the parent
- QNX does not support *fork()* in a multi-threaded process
- *fork()* returns child's pid for the parent and 0 for the child

parent

```
pid = fork();
if (pid > 0) {
  // parent does this section
} else if (pid == 0 ) {
  // child does this section
} else {
  // error return to parent
}
```

child

```
pid = fork();
if (pid > 0) {
  // parent does this section
} else if (pid == 0 ) {
  // child does this section
} else {
  // error return to parent
}
```

Fork() - Inherited resources

What resources get inherited?

- inherited:
  - filedescriptors (fds)
  - any thread attributes that inherit (e.g. priority, scheduling algorithm, signal mask, io privilege)
  - uid, gid, umask, process group, session
  - address space is replicated
- not inherited:
  - side channel connections (coids)
  - channels (chids)
  - timers

Process Creation - exec*()

The *exec*()* family of functions replace the current process environment with a new program loaded from storage

- process id (**`pid`**) remains the same
- inheritance is mostly same as *fork()* except:
  - address space is created new
  - inheritance of filedescriptors (fds) is configurable on a per fd basis
- arguments and environment variables may be passed to the new program
- these functions will not return unless an error occurs

Process Creation - spawn*()

To run a new program:

- use the *spawn*()* calls or *spawn()*
  - will load and run a program in a new process
  - will return the pid of the child process
  - inheritance rules follow that of *fork()* and *exec*()*
- *spawn*()* are convenience functions
- *spawn()* does the actual work
  - gives more control
  - more complex to use

Process Creation - The QNX way

Fork & exec vs spawn

- fork & exec is traditional Unix way
  - portable
  - inefficient
- spawn does this as a single operation
  - avoids the copy of data segment,
  - avoids a lot of setup and initialization that will immediately get torn down again
  - fewer calls
  - can be done from a multi-threaded process

Topics:

Processes and Threads

Processes

- Creation
- Detecting termination

Threads

Synchronization

Conclusion

Detecting Process Termination

We'll consider two cases:

- detecting the termination of a child
  - this is the only situation with POSIX support
- client-server relationship

Process Termination - Child death

When a child dies:

- the parent will be sent a **`SIGCHLD`** signal
  - **`SIGCHLD`** does not terminate a process
- the parent can determine why the child died by calling *waitpid()* or other *wait*()* functions

When a Parent dies :

- if the parent does not wait on the child, the child will become a zombie
  - a zombie uses no CPU, most resources it owns are freed, but an entry remains in the process table to hold its exit status
  - **`signal(SIGCHLD, SIG_IGN)`** in the parent will prevent the notification of death and creation of zombies

Process Termination - Client - server

If you have a client-server relationship:

- if a client dies, all of its connections are detached
- if a server dies, all of its channels are destroyed
- these notifications are also delivered in case of severed network connection
- these notifications happen on death, but can happen otherwise as well
- can be used to detect death explicitly if:
  - a client only detaches a connection at death, or
  - a server only destroys its channel at death

Process Termination - Client side

Client side:

- a client can get notification of the destruction of a server's channel:
  - servers usually only do this at termination
  - client must have a channel
  - must specify **`_NTO_CHF_COID_DISCONNECT`** flag for the channel when creating it
  - will get a pulse as notification

Process Termination - Server side

If the server is not a resource manager:

- can get notification when all connections from a client are detached, the server:
  - must specify **`_NTO_CHF_DISCONNECT`** flag for its channel when creating it
  - will receive a pulse as notification

If the server is a resource manager:

- will get a close message for each of the client's open file descriptors

Topics:

Processes and Threads

Processes

Threads

- Creation
- Detecting termination
- Operations

Synchronization

Conclusion

Thread Creation - pthread_create()

To create a thread, use:

```
pthread_create (pthread_t *tid, pthread_attr_t *attr,
                void *(*func) (void *), void *arg);
```

Example:

```
pthread_create (&tid, &attr, &func, &arg);
```

- the thread will start execution in *func()*. *func()* is the "main" for the thread. All other parameters can be **`NULL`**
- on return from *pthread_create()*, the **`tid`** parameter will contain the tid (thread id) of the newly created thread
- **`arg`** is miscellaneous data of your choosing to be passed to *func()*
- **`attr`** allows you to specify thread attributes such as what priority to run at, ...

Setting up thread attributes

```
pthread_attr_t attr;

pthread_attr_init(&attr);
... /* set up the pthread_attr_t structure */
pthread_create (&tid, &attr, &func, &arg);
```

- *pthread_attr_init()* sets the **`pthread_attr_t`** members to their default values
- we'll talk about some of the things you might set in the attribute structure

Thread Attributes - Functions for dealing with attributes

Functions for setting attributes

- initializing, destroying

  pthread_attr_init(),

  pthread_attr_destroy()

- setting it up

  pthread_attr_setdetachstate(),

  pthread_attr_setinheritsched(),

  pthread_attr_setschedparam(),

  pthread_attr_setschedpolicy(),

  pthread_attr_setstackaddr(),

  pthread_attr_setstacksize()

Setting priority and scheduling algorithm

– setting both:

```
struct sched_param param;
pthread_attr_setinheritsched (&attr, PTHREAD_EXPLICIT_SCHED);
param.sched_priority = 15;
pthread_attr_setschedparam (&attr, &param);
pthread_attr_setschedpolicy (&attr, SCHED_RR);
pthread_create (NULL, &attr, func, arg);
```

– setting priority only:

```
struct sched_param param;
pthread_attr_setinheritsched (&attr, PTHREAD_EXPLICIT_SCHED);
param.sched_priority = 15;
pthread_attr_setschedparam (&attr, &param);
pthread_attr_setschedpolicy (&attr, SCHED_NOCHANGE);
pthread_create (NULL, &attr, func, arg);
```

Thread Attributes - Stack Allocation

You can control the thread's stack allocation:

– to set the maximum size:

```
pthread_attr_setstacksize (&attr, size);
```

– to provide your own buffer for the stack:

```
pthread_attr_setstackaddr (&attr, addr);
```

Thread Attributes - Stack Allocation

Thread stack allocation can be automatic:

```
size = 0;              // default size
addr = NULL;           // OS allocates
```

partly automatic:

```
size = desired_size;
addr = NULL;           // OS allocates
```

or totally manual:

```
size = sizeof (*stack_ptr);
addr = stack_ptr;
```

Your stack size should be the sum of:

```
PTHREAD_STACK_MIN +
platform_required_amount_for_code;
```

Topics:

Processes and Threads

Processes

Threads

- Creation
- Detecting termination
- Operations

Synchronization

Conclusion

Waiting for threads to die & finding out why

- if a thread is “joinable” then you can wait for it to die

```
pthread_create (&tid, …, worker_thread, …);
// at this point, worker_thread is running
// ... do stuff
// now check if worker_thread died or wait for
// it to die if it hasn’t already
pthread_join (tid, &return_status);
```

- if it dies before the call to *pthread_join()* then *pthread_join()* returns immediately and **`return_status`** contains the thread’s return value or the value passed to *pthread_exit()*
- once a *pthread_join*() is done, the information about the dead thread is gone

Thread Termination - Getting Notification

The kernel can send you a pulse message:

- to get a pulse message when any thread in your process dies, set the **`_NTO_CHF_THREAD_DEATH`** flag when you create your channel

```
chid = ChannelCreate (_NTO_CHF_THREAD_DEATH);
```

- in the pulse message, the code will be **`_PULSE_CODE_THREADDEATH`** and the value will be the tid of the thread that died

Topics:

Processes and Threads

Processes

Threads

- Creation
- Detecting termination
- Operations

Synchronization

Conclusion

Some thread operations:

pthread_exit (retval)	terminate the calling thread
pthread_kill (tid, signo)	set signal **`signo`** on thread **`tid`**
pthread_detach (tid)	make the thread detached (i.e. unjoinable)
tid = pthread_self ()	find out your thread id
pthread_equal (tid1, tid2)	compare two thread ids

Set/get priority and scheduling algorithm:

– setting:

```
struct sched_param param;

param.sched_priority = new_value;
pthread_setschedparam (tid, policy, &param);
```

see the documentation for other members for sporadic scheduling

– getting:

```
int policy;
struct sched_param param;

pthread_getschedparam (tid, &policy, &param);
```

but what priority does this get? ...

Thread Operations - Priorities

Two of the priorities it gets are:

- **`param.sched_priority`** contains the *defined* priority - the priority assigned to the thread (e.g. at thread creation time)
- **`param.sched_curpriority`** contains the *current* priority

Topics:

Processes and Threads

Processes

Threads

→ **Synchronization**

- mutexes
- condvars
- semaphores
- atomic operations

Conclusion

Synchronization - Resource Inheritance

Threads within a process share:

- Timers
- Channels
- Connections
- Memory Access
- File pointers / descriptors
- Signal Handlers

Synchronization - Problems

Threads introduce new solutions, but new problems as well:

Common memory areas:

- multiple writers can overwrite each other's values,
- readers don't know when data is stable or valid,

Similar problems occur with other shared resources...

Synchronization - Problems and Solutions

These are problems with:

SYNCHRONIZATION

In this section, we'll see some tools for solving these problems:

- mutexes,
- condvars,
- semaphores,
- atomic operations

Syncrhonization - Volatile

Any variables that are being shared:

- should be declared as **`volatile`**

```
volatile unsigned flags;
...
atomic_clr (&flags, A_FLAG);
```

- **`volatile`** is an ANSI C keyword that tells the compiler not to optimize the variable

  Example:

  The compiler may optimize the code by having it store the value in a register, and refer to the register all the time instead of going back to the memory. Meanwhile, the another thread's code would be accessing the memory!

Topics:

Processes and Threads

Processes

Threads

Synchronization

→ – mutexes
– condvars
– semaphores
– atomic operations

Conclusion

Mutual Exclusion

"Mutual exclusion" means only *one* thread:

- is allowed into a critical section of code at a time
- is allowed to access a particular piece of data at a time

Mutual Exclusion

POSIX provides the following calls:

- administration

```
pthread_mutex_init (pthread_mutex_t *,
                    pthread_mutexattr_t *);
pthread_mutex_destroy (pthread_mutex_t *);
```

- usage

```
pthread_mutex_lock (pthread_mutex_t *);
pthread_mutex_trylock (pthread_mutex_t *);
pthread_mutex_unlock (pthread_mutex_t *);
```

A simple example:

```
pthread_mutex_t myMutex;

init () {
    ...
    // create the mutex for use
    pthread_mutex_init (&myMutex, NULL);
    ...
}
thread_func () {
    ...
    // obtain the mutex, wait if necessary
    pthread_mutex_lock (&myMutex);
    // critical data manipulation area
    // end of critical region, release mutex
    pthread_mutex_unlock (&myMutex);
    ...
}
cleanup () {
    pthread_mutex_destroy (&myMutex);
}
```

use default attributes

Consider malloc():

A number of threads request memory from *malloc()*. Internally, *malloc()* maintains a “free list” of memory blocks that are available for allocation.
All threads in the process use the same free list.

Mutual Exclusion

Simplified malloc() source looks something like this:

```
void *
malloc (int nbytes)
{
   …
   while (freeList && freeList -> size != nbytes) {
     freeList = freeList -> next;
   }
   if (freeList) {
     … // mark block as used, and return block address to caller
     return (freeList -> memory_block);
   }
   …
}
```

Mutual Exclusion

Now consider a number of threads that use malloc():

```
thread1 ()
{
  char    *data;

  data = malloc (64);
}

thread2 ()
{
  char    *other_data;

  other_data = malloc (64);
}
```

Something bad happens!

Mutual Exclusion

The problem is, multiple threads can get in each other's way!

What we need is *exclusive* access to the "**`freeList`**" data structure!

We'll use a MUTEX to do this...

Let's fix the **malloc** routine:

```
pthread_mutex_t                 malloc_mutex;

void *
malloc (int nbytes)
{
   …
   pthread_mutex_lock (&malloc_mutex);
   while (freeList && freeList -> size != nbytes) {
     freeList = freeList -> next;
   }
   if (freeList) {
     … // mark block as used, and return block
     block = freeList -> memory_block;
     pthread_mutex_unlock (&malloc_mutex);
     return (block);
   }
   pthread_mutex_unlock (&malloc_mutex);
   …
}
```

critical section

Mutex Initialization

To explicitly initialize the mutex:

```
pthread_mutex_init (&malloc_mutex, NULL);
```

If successful, this ensures that all appropriate resources have been allocated for the mutex.

Mutex Initialization

A simple method for mutex initialization:

```
// static initialization of Mutex

pthread_mutex_t malloc_mutex =
  PTHREAD_MUTEX_INITIALIZER;

void *
malloc (int nbytes)
{
  . . .
  // MUTEX will be initialized the first time
  // it is used …
  pthread_mutex_lock (&malloc_mutex);
  . . .
```

Mark as: “not in use” and “to be initialized the first time that it is used”.

Sharing Mutexes between Processes

By default, mutexes cannot be shared between processes

- to make them shared, set the **PTHREAD_PROCESS_SHARED** flag for the mutex
- the mutex should be in shared memory
- e.g.:

```
pthread_mutexattr_t mutex_attr;
pthread_mutex_t *mutex;
pthread_mutexattr_init( &mutex_attr );
pthread_mutexattor_setpshared( &mutex_attr,
  PTHREAD_PROCESS_SHARED);
mutex = (pthread_mutex_t *)shmem_ptr;
pthread_mutex_init( mutex, &mutex_attr );
```

Designing with Mutexes - Mutex Locked Time

Keep mutexes locked for short periods

- keep a mutex locked for as short a time as possible
- while one thread has the mutex locked, other threads that want the resource protected by the mutex have to wait
- with QNX there is the added benefit that if the mutex is not locked, and you try to lock it, the amount of code you'll end up doing is very short and very fast...

Topics:

Processes and Threads

Processes

Threads

Synchronization

- mutexes
- condvars
- semaphores
- atomic operations

Conclusion

Condvars

Consider a simple case where:

– we need to block, waiting for another thread to change a variable:

```
int state;

thread_1 ()
{
  while (1) {
    // wait until "state" changes,
    // then, perform some work
  }
}
```

– condvars provide a mechanism for doing this

The condvar calls:

```
pthread_cond_init (pthread_cond_t *,
                   pthread_condattr_t *);
pthread_cond_wait (pthread_cond_t *,
                   pthread_mutex_t *);
pthread_cond_signal (pthread_cond_t *);
pthread_cond_broadcast (pthread_cond_t *);
```

Condvars - Example

An example:

1. a data provider thread gets some data, likely from a client process, and adds it to a queue
2. it tells the hardware handling thread that data is there
3. the hardware writing thread wakes up, removes the data from the queue and writes it to the hardware

To do all this we need two things:

- a mutex to make sure that the two threads don't access the queue data structure at the same time
- a mechanism for the data provider thread to tell the hardware writing thread to wake up

Hardware handling thread's code:

```
while (1) {
  pthread_mutex_lock (&mutex);          // get exclusive access
  if (!data_ready)
    pthread_cond_wait (&cond, &mutex);  // we wait here

  /* get and decouple data from the queue */
  while ((data = get_data_and_remove_from_queue ()) != NULL) {
    pthread_mutex_unlock (&mutex);
    write_to_hardware (data); // pretend to do this
    free (data);              // we don't need it after this
    pthread_mutex_lock (&mutex);
  }
  data_ready = 0;               // reset flag
  pthread_mutex_unlock (&mutex);
}
```

Data providing thread's code:

```
pthread_mutex_lock (&mutex);      // get exclusive access
add_to_queue (buf);
data_ready = 1;                   // set the flag
pthread_cond_signal (&cond);      // notify a waiter
pthread_mutex_unlock (&mutex);    // release exclusivity
```

Let's zoom in on the "wait":

```
pthread_cond_wait (&condvar, &mutex);
```

- **unlock**
- **wait**
- becomes **READY**
- **lock**

unlock

- allows other threads to get access

wait

- this is the actual "waiting" part

becomes READY

- just because the thread is no longer waiting, that doesn't mean that it gets the CPU. The lock doesn't happen until the function is returning.

lock

- ensures that we once again have access

Signalling vs broadcasting:

- Threads 1, 2 and 3 (all at the same priority) are waiting for a change via *pthread_cond_wait()*,
- Thread 4 makes a change, signals the variable via *pthread_cond_signal()*,
- The longest waiting thread (let's say "2") is informed of the change, and tries to acquire the mutex (done automatically by the *pthread_cond_wait()*),
- Thread 2 tests against its condition, and either performs some work, or goes back to sleep

Condvars - Signalling vs broadcasting

What about threads 1 and 3?

- they never see the change!

If we change the example:

- use *pthread_cond_broadcast()* instead of *pthread_cond_signal()*,
- then <u>all three</u> threads will get the signal
  - they will all become READY, but only one can lock the mutex at a time so they will end up taking turns.

Condvars - Signalling vs broadcasting

We use one over the other?

- use a signal if:
  - you have only one waiting thread, or
  - you need only one thread to do the work and you don't care which one
- use a broadcast if you have multiple threads and:
  - they all need to do something, or
  - they don't all need to do something but you don't know which one(s) to wake up

Sharing condvars between processes

By default, condvars cannot be shared between processes

- to make them shared, set the **PTHREAD_PROCESS_SHARED** flag for the condvar
- the condvar should be in shared memory
- e.g.:

```
pthread_condattr_t cond_attr;
pthread_cond_t *cond;
pthread_condattr_init( &cond_attr );
pthread_condattr_setpshared( &cond_attr,
  PTHREAD_PROCESS_SHARED);
cond = (pthread_cond_t *)shmem_ptr;
pthread_cond_init(cond, &cond_attr );
```

Producer / Consumer example:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond = PTHREAD_COND_INITIALIZER;
volatile int    state = 0;
volatile int    product = 0;

void *consume (void *arg) {
    while (1) {
        pthread_mutex_lock (&mutex);
        while (state == 0) {
            pthread_cond_wait (&cond, &mutex);
        }
        printf ("Consumed %d\n", product);
        state = 0;
        pthread_cond_signal (&cond);
        pthread_mutex_unlock (&mutex);
        do_consumer_work ();
    }
    return (0);
}
```

Condvars

```
void *produce (void *arg) {
    while (1) {
        pthread_mutex_lock (&mutex);
        while (state == 1) {
            pthread_cond_wait (&cond, &mutex);
        }
        printf ("Produced %d\n", product++);
        state = 1;
        pthread_cond_signal (&cond);
        pthread_mutex_unlock (&mutex);
        do_producer_work ();
    }
    return (0);
}

int main () {
    pthread_create (NULL, NULL, &produce, NULL);
    consume (NULL);
    return (EXIT_SUCCESS);
}
```

Topics:

Processes and Threads

Processes

Threads

Synchronization

- mutexes
- condvars
- semaphores
- atomic operations

Conclusion

Semaphores for access control:

administration

unnamed semaphores

```
sem_init (sem_t *semaphore, int pshared, unsigned int val);
sem_destroy (sem_t *semaphore);

sem_t *sem_open (char *name, int oflag, [int sharing,
                 unsigned int val]);
sem_close (sem_t *semaphore);
sem_unlink (char *name);
```

named semaphores

usage:

```
sem_post (sem_t *semaphore);
sem_trywait (sem_t *semaphore);
sem_wait (sem_t *semaphore);
sem_getvalue (sem_t *semaphore, int *value);
```

Unnamed vs named semaphores

- with unnamed, *sem_post()* and *sem_wait()* call the semaphore kernel calls directly whereas...
- with named semaphores, *sem_post()* and *sem_wait()* send messages to mqueue who then makes the semaphore kernel call
- so unnamed semaphores are faster than named semaphores
- if you are using semaphores within a multithreaded process, unnamed semaphores are easy since the semaphore can simply be a global variable

Topics:

Processes and Threads

Processes

Threads

Synchronization

- mutexes
- condvars
- semaphores
- atomic operations

Conclusion

Atomic Operations

For short operations, such as incrementing a variable:

`atomic_add`	does += some value
`atomic_add_value`	*atomic_add()*, but returns result
`atomic_clr`	does &= ~some value
`atomic_clr_value`	*atomic_clr()*, but returns result
`atomic_set`	does |= some value
`atomic_set_value`	*atomic_set()*, but returns result
`atomic_sub`	does -= some value
`atomic_sub_value`	*atomic_sub()*, but returns result
`atomic_toggle`	does ^= some value
`atomic_toggle_value`	*atomic_toggle()*, but returns result

These functions:

- are guaranteed to complete without being preempted
- can be use between two threads (even on SMP)
- can be used between a thread and an ISR

Topics:

- **Processes and Threads**
- **Processes**
- **Threads**
- **Synchronization**
- → **Conclusion**

Conclusion

You learned:

- what a process is and what a thread is
- why you'd use threads
- how to create processes and threads
- how to detect when processes and threads die
- how to synchronize among threads using mutexes, condvars, and other methods

Reference

David R. Butenhof, *Programming with POSIX Threads*, Addison-Wesley, 1997, ISBN 0-201-63392-2

Kay A. Robbins and Steven Robbins, *Practical Unix Programming*, Prentice Hall, 1996, ISBN 0-13-443706-3

Bill O. Gallmeister, *POSIX.4 - Programming for the Real World*, O'Reilly & Associates, 1995

Rob Krten, *Getting Started with QNX Neutrino 2*, Parse Software Devices, 1999, ISBN 0-9682501-1-4

Interprocess Communication

Introduction

QNX Neutrino supports a wide variety of IPC:

- Native QNX Neutrino Messaging
- QNX Neutrino Pulses
- POSIX Signals
- Shared Memory
- Pipes (requires **`pipe`** process)
- POSIX Message Queues (requires **`mqueue`** process)
- POSIX fd/fp Based Functions (by writing resource managers)

Topics:

→ **Native QNX Neutrino Messaging**

Pulses

Signals

Event Delivery

Shared Memory

Pipes, POSIX Msg Queues

POSIX fd/fp Based Functions

Conclusion

Let's look at the core IPC services:

1. client sends request to server
2. server receives, and processes
3. server replies to client, client gets response from server, and continues

Since the *MsgSend()* does not return until Thread 2 calls *MsgReply()* it looks as if the *MsgSend()* function itself had done the work

Synchronization

Receive before Send

Synchronization

Send before Receive

Message Related Thread States

Thread States - Client side

Done by our thread

Done by another thread

Thread States - Server side

Done by our thread

Done by another thread

Synchronization and Thread States

Synchronization and Thread States

Send before Receive

Behind the scenes:

- Server:
  - creates a channel (*ChannelCreate()*)
  - waits for a message (*MsgReceive()*)
  - performs processing
  - sends reply (*MsgReply()*)
  - goes back for more
- Client:
  - attaches to channel (*ConnectAttach()*)
  - sends message (*MsgSend()*)
  - processes reply
  -

Connections and channels in diagram form:

- Servers receive on channels,
- Clients connect to channels and send to the channels over the connections

The server creates a channel using:

```
chid = ChannelCreate (flags);
```

- A channel is attached to a process. Any thread in that process can receive messages from that channel. When a message arrives the kernel simply picks a thread to receive it.
- You could have multiple channels associated with a process. This might be used in the case where the process consists of a number of threads, and provides multiple types of services. Clients could connect to one channel for one type of service, and another channel for another type of service.
- **`flags`** is a bitfield, we will look at some of the flags later.

The client connects to the server's channel:

```
coid = ConnectAttach (nd, pid, chid, _NTO_SIDE_CHANNEL, flags);
```

- **nd**, **pid**, **chid** uniquely identifies the server's channel:
  nd is the node that the server is running on. Use **0** or
  ND_LOCAL_NODE if the server is on the client's node,
  pid is the process id of the process that the channel belongs to,
  chid is the channel id
- Always pass **_NTO_SIDE_CHANNEL** for the **index** parameter.
  Why? ...

Connection ids come in two types:

```
fd = open (filename, O_RDONLY);

coid = ConnectAttach (nd, pid, chid, _NTO_SIDE_CHANNEL, flags);
```

When calling *ConnectAttach()* yourself, you do not want your coid to be in the file descriptor range. Passing **`_NTO_SIDE_CHANNEL`** prevents it.

The MsgSend() call:

```
status = MsgSend (coid, smsg, sbytes, rmsg, rbytes);
```

`coid` is the connection ID,

`smsg` is the message to send,

`sbytes` is the number of bytes of **`smsg`** to send,

`rmsg` is a reply buffer for the reply message to be put into,

`rbytes` is the size of **`rmsg`**,

`status` is what will be passed as the *MsgReply*()*'s **`status`** parameter

The MsgReceive() call:

```
rcvid = MsgReceive (chid, rmsg, rbytes, info);
```

`chid` is the channel ID,

`rmsg` is a buffer in which the message is to be received into,

`rbytes` is the number of bytes to receive in **`rmsg`**,

`info` allows us to get additional information,

`rcvid` allows us to *MsgReply*()* to the client

The MsgReply() call:

```
MsgReply (rcvid, status, msg, bytes);
```

`rcvid` is the receive ID returned from the server's *MsgReceive*()* call,

`status` is the value for the *MsgSend*()* to return, do not use a negative value

`msg` is reply message to be given to the sender,

`bytes` is the number of bytes of **`msg`** to reply with

The MsgError() call:

– will cause the *MsgSend*()* to return -1 with **`errno`** set to whatever is in **`error`**.

```
MsgError (rcvid, error);
```

`rcvid` is the receive ID returned from the server's *MsgReceive*()* call

`error` is the error value to be put in **`errno`** for the sender

Messaging - An example

The server:

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

int  chid;          // channel ID

main ()
{
    int  rcvid;     // receive ID

    chid = ChannelCreate (0 /* or flags */);
    /* create client thread */
    ...
    while (1) {
        rcvid = MsgReceive (chid, &recvmsg, rbytes, NULL);
        // process message from client here...
        MsgReply (rcvid, 0, &replymsg, rbytes);
    }
}
```

The client:

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

extern int chid;
int coid;    // connection id
int status;

client_thread() {
  // create the connection, typically done only once
  coid = ConnectAttach (0, 0, chid, _NTO_SIDE_CHANNEL, 0);
  ...
  // at some point later we decide we want to send a message
  status = MsgSend (coid, &sendmsg, sbytes, &replymsg, rbytes);
}
```

Messaging - The message data

Message data is always copied:

- the kernel does not pass pointers
- there is no practical maximum message size

Examining Synchronization and Thread States

To get a list of threads running on your processor and to see their states enter:

```
pidin

  pid tid name                 prio STATE       Blocked
    1   1 /boot/sys/procnto     0f READY
    1   2 /boot/sys/procnto    10r RECEIVE      1
    1   3 /boot/sys/procnto    15r RECEIVE      1
    1   4 /boot/sys/procnto    10r RECEIVE      1
    1   5 /boot/sys/procnto    15r RECEIVE      1
    1   6 /boot/sys/procnto     6r NANOSLEEP
    1   7 /boot/sys/procnto    10r RUNNING
    1   8 /boot/sys/procnto    10r RECEIVE      1
 4100   1 /tinit/x86/o/tinit   10o REPLY        1
94219   1 devc-pty             20o RECEIVE      1
446488  1 r/photon/bin/pterm   10o RECEIVE      1
450595  1 bin/sh               10o SIGSUSPEND
573478  1 /pidin/x86/o/pidin   10o REPLY        1
536617  1 r/photon/bin/pterm   10o RECEIVE      1
540714  1 bin/sh               10o REPLY        94219
```

For RECEIVE blocked state, channel id.

For the REPLY blocked state, pid of process that the channel we are blocked on belongs to.

Designing For Message Passing - Message types/structures

How do you design a message passing interface?

- define message types and structures in a common message header
- start all messages with a message type
- have a structure matching each message type
  - if messages are related or they use a common structure, consider using message types & subtypes
  - define matching reply structures, if appropriate
- avoid overlapping message types for different types of servers

Designing For Message Passing - Server code

On the server side

- generally contains setup code then an infinite loop around a *MsgReceive()*
- branch based on message type, e.g.:

```
while(1) {
  recvid = MsgReceive( chid, &msg, sizeof(msg), NULL );
  switch( msg.hdr.type ) {
     case MSG_TYPE_1:
       handle_msg_type_1(rcvid, &msg);
       break;
     case MSG_TYPE_2:
     …
   }
}
```

Messaging - Using IOVs

What if you wanted to send the following three buffers in one message?

– You could do three *memcpy()*s to produce one big buffer:

But there is a more convenient way...

Messaging - Using IOVs

Instead of giving the kernel the address of one buffer using *MsgSend()* ...

one buffer

```
MsgSend      (int coid, void *smsg, int sbytes,
              void *rmsg, int rbytes);
```

one buffer

... give the kernel an array of pointers to buffers using *MsgSendv()*:

array of pointers to multiple buffers

```
MsgSendv     (int coid, iov_t *siov, int sparts,
              iov_t *riov, int rparts);
```

array of pointers to multiple buffers

What does **iov_t** look like?

```
typedef struct {
  void          *iov_base;
  size_t        iov_len;
} iov_t;
```

Most useful as an array:

```
iov_t           iovs [3];
```

Messaging - Using IOVs

Using an IOV:

```
SETIOV (&iovs [0], &part1, sizeof (part1));
SETIOV (&iovs [1], &part2, sizeof (part2));
SETIOV (&iovs [2], &part3, sizeof (part3));
```

When sent or received, these parts are considered as one contiguous sequence of bytes. This is ideal for scatter/gather buffers & caches...

☞ IOVs are used in the Messaging functions that contain a “v” near the end of their name: (MsgReadv/MsgReceivev/MsgReplyv/MsgSendsv/MsgSendsvnc/MsgSendv/MsgSendvnc/ MsgSendvs/MsgSendvsnc/MsgWritev)

Client side:

```
write (fd, buf, size);
```

effectively does:

```
hdr.nbytes = size;
SETIOV (&siov[0], &header, sizeof (header));
SETIOV (&siov[1], buf, size);
MsgSendv (fd, siov, 2, NULL, 0);
```

What actually gets sent:

A contiguous stream of bytes

On the server side, what gets received:

```
// assume riov has been setup
MsgReceivev (chid, riov, 4, NULL);
```

Messaging - Using IOVs

In reality, though, we don't know how many bytes to receive, until we've looked at the header:

```
rcvid = MsgReceive (chid, &header,
                    sizeof (header), NULL);
```

header

Then, we can set up an IOV and read:

```
SETIOV (iov [0], &cbuf [6], 4096);
SETIOV (iov [1], &cbuf [2], 4096);
SETIOV (iov [2], &cbuf [5], 4096);
MsgReadv (rcvid, iov, 3, sizeof(header));
```

Offset, 12 bytes

From client to server:

Data copied from client's address space to server's address space by kernel

Continuing from client to server:

Data copied from client's address space to server's address space by kernel

Messaging - Using IOVs

Or, alternatively:

Data copied from client's address space to server's address space by kernel

And then getting more later:

Data copied from client's address space to server's address space by kernel

MsgWrite

For copying from server to client:

```
MsgWrite (rcvid, smsg, sbytes, offset)
MsgWritev (rcvid, siov, sparts, offset)
```

They write bytes from the server to the client, but don't unblock the client.

The data from smsg or siov is copied to the client's reply buffer.

To complete the message exchange (i.e., unblock the client), call *MsgReply*()*.

MsgWrite Example

Example:

```
nbytes = MsgSend(  coid, ...,
                &replybuf, ... );
```

① 1st call to *MsgWrite()*

② 2nd call to *MsgWrite()*

③ 3rd call to *MsgWrite()*

⋮

When no more data, Msg*Reply()*

```
int  offset = 0;
char  data[1000];


while (1) {
  rcvid = MsgReceive( chid,... ); /* for data */
  isData = getSomeData( data );
  while ( isData ) {
    MsgWrite( rcvid, data, 1000, offset );
    offset += 1000; /* set for next MsgWrite()*/
    isData = getSomeData( data );
  }
  /* reply with the number of bytes written
     as the status */
  MsgReply( rcvid, offset, NULL, 0 );
}
```

Stressing where the message data goes:

```
MsgSend (coid, txmsg, txbytes, rxmsg, rxbytes);

MsgReceive (chid, rxmsg, rxbytes, &info);

MsgRead (rcvid, rxmsg, rxbytes, offset);

MsgWrite (rcvid, txmsg, txbytes, offset);

MsgReply (rcvid, status, txmsg, txbytes);
```

In summary, the following client calls exist:

```
coid = ConnectAttach (int nd, pid_t pid, int chid,
                      unsigned index, unsigned flags);
ConnectDetach( int coid );
MsgSend      (int coid, void *smsg, int sbytes,
              void *rmsg, int rbytes);
MsgSendnc    (...);          // same params as MsgSend
MsgSendv     (int coid, iov_t *siov, int sparts,
              iov_t *riov, int rparts);
MsgSendvnc   (...);          // same params as MsgSendv
MsgSendsv    (int coid, void *smsg, int sbytes,
              iov_t *riov, int rparts);
MsgSendsvnc  (...);          // same params as MsgSendsv
MsgSendvs    (int coid, iov_t *siov, int sparts,
              void *rmsg, int rbytes);
MsgSendvsnc  (...);          // same params as MsgSendvs
```

As well as server calls:

```
chid = ChannelCreate (unsigned flags);
ChannelDestroy (int chid);

rcvid = MsgReceive (int chid, void *rmsg, int rsize,
                    struct _msg_info *info);
rcvid = MsgReceivev (int chid, iov_t *riov, int rparts,
                     struct _msg_info *info);

MsgReply (int rcvid, int status, void *msg, int size);
MsgReplyv (int rcvid, int status, iov_t *msg,
           int parts);
MsgError (int rcvid, int error);
```

continued...

Server calls (continued):

```
MsgRead (int rcvid, void *msg, int size, int offset);
MsgReadv (int rcvid, iov_t *iov, int rparts,
          int offset);

MsgWrite (int rcvid, void *msg, int size, int offset);
MsgWritev (int rcvid, iov_t *iov, int sparts,
           int offset);
```

Designing For Message Passing - Large messages

When dealing with large data carrying messages:

- they should be built as a header followed by the data
  - client will generally send them using an iov, e.g.

```
SETIOV(&iov[0], &hdr, sizeof(hdr) );
SETIOV(&iov[1], data_ptr, bytes_of_data );
MsgSendv(coid, iov, 2, …);
```

- server will generally want a receive buffer large enough to handle all non-data carrying messages
  - can easily do this by declaring the receive buffer to be a union of all message structures
  - use *MsgRead*()* to process large data messages

Message Information - Getting the information

To get more info about the client:

```
struct _msg_info info;

rcvid = MsgReceive (chid, rmsg, rbytes, &info);
MsgInfo (rcvid, &info);
```

You can get the information here, or later, with the *MsgInfo()* call

Message Information - struct _msg_info

The **`_msg_info`** structure contains at least:

`int`	`nd;`	node descriptor
`pid_t`	`pid;`	sender's process ID
`int`	`tid;`	sender's thread ID
`int`	`chid;`	channel ID
`int`	`scoid;`	server connection ID
`int`	`coid;`	sender's connection ID
`int`	`priority;`	sender's priority
`int`	`msglen;`	message size information (see next slide…)
`int`	`srcmsglen;`	
`int`	`dstmsglen;`	

Message Information - Message sizes

Message size info:

```
                info.srcmsglen                    info.dstmsglen
                      ||                                ||
MsgSend(coid,smsg,100,rmsg,300)
```

```
rcvid = MsgReceive(chid,rmsg,200,&info)
```

```
                    ||
          info.msglen (number of bytes copied)

MsgReply(rcvid,status,rmsg,150)
```

- ☞ **`srcmsglen`**: valid only if **`_NTO_CHF_SENDER_LEN`** was passed to *ChannelCreate()*
- ☞ **`dstmsglen`**: valid only if **`_NTO_CHF_REPLY_LEN`** was passed to *ChannelCreate()*

Message buffers

Priority Queueing

What happens when the server calls *MsgReceive()* and there are several clients SEND blocked?

- the message from the highest priority SEND block client is received
- if multiple clients have the same priority, the one that has been waiting longest is handled
- this follows the same rules as scheduling

Priority Queueing

If threads call MsgSend() in order:

Thread id	Priority
1	10
2	15
3	10
4	20

They will be received in order: 4,2,1,3

Messaging - Delayed reply

The server doesn't need to reply immediately:

Singlethread Model

Typical server:

Singlethread Model

If the request takes a long time:

Panic!
Clients 3 and 5
missed a deadline!

Multithread Model

The server can start up some threads:

Threads do the work!

Multithread Model

The multithread model:

- threads all use the same chid to receive messages from clients,
- can distribute over SMP,
- threads inherit the priority of their respective clients...

ChannelCreate flags

ChannelCreate() takes a flags parameter, some flags are:

`_NTO_CHF_DISCONNECT`

request notification when a client goes away

`_NTO_CHF_COID_DISCONNECT`

request notification when a server goes away

`_NTO_CHF_UNBLOCK`

request notification if a REPLY blocked client wants to unblock

`_NTO_CHF_SENDER_LEN`

request the send length passed to *MsgSend*()* be calculated for the _msg_info structure's srcmsglen element

`_NTO_CHF_REPLY_LEN`

request the reply length passed to *MsgSend*()* be calculated for the _msg_info structure's dstmsglen element

`_NTO_CHF_THREAD_DEATH`

request notification of thread death in the server

The disconnect flag:

`_NTO_CHF_DISCONNECT`:

Tells the kernel to deliver a pulse when all connections from a particular process are detached, including when all connections are detached from within the calling process. This allows the server to perform tidying activities. The **`code`** member of the pulse message will contain **`_PULSE_CODE_DISCONNECT`**. You must do **`ConnectDetach(pulse.scoid)`** for proper cleanup to be done.

Unblock Problems

Unblock problems:

ChannelCreate flags - Unblock

The unblock flag illustrated:

See the notes for the details.

ChannelCreate flags - Unblock

The unblock flag:

`_NTO_CHF_UNBLOCK`:

Tell the kernel to deliver a pulse when a thread which is REPLY blocked on the server would be unblocked. A thread could be unblocked due to receiving a signal, a cancellation, or a kernel timeout. The **`code`** member of the pulse message will contain **`_PULSE_CODE_UNBLOCK`** and the **`value.sival_int`** member will contain the rcvid that the original *MsgReceive*()* had returned.

This allows the server to clean up any resources that may have been allocated to the client, since the client is no longer interested in waiting for the result.

The server **MUST** *MsgReply*()* or *MsgError()* to the client to acknowledge the unblock, otherwise the client will remain blocked.

How does the client find the server?

How does the client find the server?

- How does the sender find the receiver? At minimum it needs to know the process id and channel id so that it can connect to it

The answer depends on how the receiver is written:

- as a resource manager
- as a simple *MsgReceive*()* loop

How does the client find the server? - Resource managers

If the receiver is a resource manager then:

- the resource manager registers a name:

```
// ... some setup code
resmgr_attach( ..., "/dev/sound", ...  );
// ... some more setup code
```

- the client does:

```
fd = open( "/dev/sound", ... );
```

OR network case:

```
fd = open("/net/nodename/dev/sound", ...);
...
write( fd, ... );    // sends some data
read( fd, ... );     // gets some data
```

OR

```
devctl( fd, ... );  // send data, get data back
```

OR

```
MsgSend( fd, ... ); // send data, get data back
```

Finding the server - name_attach()/name_open()

If the server is simple *MsgReceive()* loop:

– use *name_attach()* and *name_open()*:

The server does:

```
name_attach_t *attach;
attach = name_attach( NULL, "myname", 0 );
...
rcvid = MsgReceive( attach->chid, &msg, sizeof(msg),
                    NULL );
...
name_detach( attach, 0 );
```

The client does:

```
coid = name_open( "myname", 0 );
...
MsgSend( coid, &msg, sizeof(msg), NULL, 0 );
...
name_close( coid );
```

Finding the server - name_attach()/name_open()

But using *name_attach()* means you will receive some additional pulse messages:

`_PULSE_CODE_DISCONNECT`

- sent to you when a client detaches all its connections
- you must call **`ConnectDetach(pulse.scoid)`**

`_PULSE_CODE_UNBLOCK`

- sent to you when a client wants to unblock but can't since you haven't replied yet

`_PULSE_CODE_COID_DEATH`

- sent to you when a channel that you have a connection to is destroyed

`_PULSE_CODE_THREADDEATH`

- sent to you when a thread in your process dies

name_attach()/name_open() - Example

Example:

```
attach = name_attach(NULL, "my_name", 0);
while (1) {
  rcvid = MsgReceive(attach->chid, &msg, sizeof(msg), NULL);
  if (rcvid == -1) {
    exit(EXIT_FAILURE); /* Error condition, exit */
  } else if (rcvid == 0) { /* Pulse received */
    switch (msg.pulse.code) {
    case _PULSE_CODE_DISCONNECT:
      /* A client disconnected all its connections (called
       * name_close() for each name_open() of our name) or
       * terminated
       */
      ConnectDetach(msg.pulse.scoid); /* you must do this */
      break;
```

continued

name_attach()/name_open() - Example (continued)

```
    default:
      /* A pulse sent by one of your processes or a
       * _PULSE_CODE_UNBLOCK, _PULSE_CODE_COIDDEATH or
       * _PULSE_CODE_THREADDEATH from the kernel?
       */
    }
    continue;
  }
  /* A QNX IO message received, reject */
  if (msg.hdr.type >= _IO_BASE && msg.hdr.type <= _IO_MAX) {
    MsgError(rcvid, ENOSYS);
    continue;
  }
  /* A message (presumably ours) received, handle */
  MsgReply(rcvid, ...);
}
```

Topics:

- Native QNX Neutrino Messaging
- → Pulses
- Signals
- Event Delivery
- Shared Memory
- Pipes, POSIX Msg Queues
- POSIX fd/fp Based Functions
- Conclusion

Pulses

Pulses have the following characteristics:

- small, non-blocking messaging
  - 32 bit value
  - 8 bit code
- fast
- inexpensive

Sending Pulses

Pulses are sent as follows:

```
MsgSendPulse (coid, priority, code, value);
```

8-bits

32-bits

- **`code`** is usually used to mean "pulse type"
  - valid range is **`_PULSE_CODE_MINAVAIL`** to **`_PULSE_CODE_MAXAVAIL`**
- **`priority`** works like the sending thread's priority for a message
  - receiving thread runs at that priority
  - delivery order is based on priority
- to send a pulse across process boundaries, the sender must be the same effective userid as the receiver or be root user

Sending Pulses

MsgSendPulse() takes a coid:

- this only works from client to server
- what if a server needs to send a pulse to a client?
  - an alternate way of sending pulses is with *MsgDeliverEvent()*
  - we'll look at this in the Event Delivery section
- receiving a pulse works the same no matter how it was sent

Receiving Pulses

Pulses are received just like messages, with a MsgReceive*() call.

- The return value (**`rcvid`**) from *MsgReceive()* will be **`0`** to indicate a pulse was received
- the pulse data will be written to the receive buffer
- the kernel does not fill in the **`_msg_info`** structure when you receive a pulse
- you do not (and can not) reply to pulses

Receiving Pulses - Example

Example:

```
typedef union {
    struct _pulse   pulse;
    // other message types you will receive
} myMessage_t;

…
myMessage_t   msg;

while (1) {
    rcvid = MsgReceive (chid, &msg, sizeof(msg), NULL);
    if (rcvid == 0) {
        // it's a pulse, look in msg.pulse… for data
    } else {
        // it's a regular message
    }
}
…
```

Pulse Structure

When received, the pulse structure has at least the following members:

```
struct _pulse {
  signed char        code;     ←——  8-bit code
  union sigval       value;    ←——  32-bit value
  int                scoid;
};
```

Receiving Pulses - Example

Zoom in:

```
...
   rcvid = MsgReceive (chid, &msg, sizeof(msg), NULL);
   if (rcvid == 0) {
        // it's a pulse, look in msg.pulse… for data
        switch (msg.pulse.code) {
        case    _PULSE_CODE_UNBLOCK:
                // a kernel unblock pulse
                ...
                break;
        case    MY_PULSE_CODE:
                // do what's needed
                ...
                break;
...
```

MsgReceivePulse() :

– is useful if you have a channel on which you may be receiving messages from *MsgSend*()* calls and pulses but at some point in time want to receive only pulses.

```
rcvid = MsgReceivePulse (chid, &pmsg,
                         sizeof(pmsg), NULL);
```

Pulses - Receive order

If a process is receiving messages and pulses:

- receive order is still based on priority, using the pulse's priority
- the kernel will run the receiving thread at the priority of the pulse it received
- pulses and messages may get intermixed

Receive Order - example

If threads send pulses and messages as follows:

TIME ↓

Thread id	Thread priority	Action	
1	10	Send pulse p1 with pulse priority 15	2
2	16	Send message	1
1	10	Send pulse p2 with pulse priority 9	5
3	11	Send message	3
1	10	Send message	4
4	17	Send pulse p3 with pulse priority 6	6

☞ the message from thread 1 gets to the server before the pulse that was sent before it

Topics:

- Native QNX Neutrino Messaging
- Pulses
- → Signals
- Event Delivery
- Shared Memory
- Pipes, POSIX Msg Queues
- POSIX fd/fp Based Functions
- Conclusion

Signals

Signals are a non-blocking form of IPC that have these characteristics:

- are POSIX
- asynchronous, can interrupt the recipient at any time
  - their asynchronous nature makes them very much like an interrupt although they are done completely in the kernel
- can be treated as synchronous, the recipient can wait for signals

Signals

Before a signal is sent, the recipient can specify what will happen:

- ignore the signal, it will be discarded by the kernel as if nothing happened
- mask the signal, when the signal arrives the signal remains pending, later when it is unmasked then it takes effect
  - if it is a queued signal then it will hit you the same number of times as it did while it was masked otherwise it will hit you only once
- **termination (the default for most signals)**
- a thread can block, waiting for the signal
- a signal handler can be set up, it will be called when the signal arrives ...

The simple case - sending the signal to a single-threaded process:

The above *kill()* function sends signal **`SIGUSR1`** to process **`67`**. Don't be misled by its name. **`SIGUSR1`** is defined in **`<signal.h>`**.

The *kill()* function will return immediately but the handler will not be called until the recipient thread gets to run - based on its priority and the scheduling algorithm of other threads at its priority.

What happens when a signal hits a multi-threaded process?

Process (process id = 67)
with three threads

The signal handler will not be called until one of the three threads gets to run. This big question is, which one does the kernel pick? This is another way of saying, which thread will get interrupted?

continued...

Signals and multi-threading (continued)

- A signal handler gets called when the recipient next runs ... but in a multi-threaded process, which thread is the recipient?
- When targeting a process (as with the *kill()* function), the kernel selects a thread that has the signal unmasked:
  - If only one thread has the signal unmasked, that thread gets it.
  - If multiple threads have that signal unmasked, then a thread that has it unmasked gets it (do not rely on which one).
  - If no threads have that signal unmasked, then the signal is made pending on the process. The first thread to unmask the signal will get it.

☞ It is important to note that a signal handler is attached to the process, not a thread. The question is which thread will be interrupted in order to call the handler? It could be any one.

Signal flowchart:

Standard signal calls:

sigqueue(), kill(), raise(), pthread_kill()

- set a signal on a process group, process, or thread

sigaction(), signal()

- define action to take on receipt of a signal

pthread_sigmask(), sigprocmask()

- change signal blocked mask of a thread

sigsuspend()

- block until a signal invokes a signal handler

sigwaitinfo()

- wait for a signal and return info on it

Signals

To deliver a signal to:

- any process:
  - `kill (pid_t pid, int signo)`
  - `sigqueue (pid_t pid, int signo, union sigval value)`
  - these require appropriate permission
    - sender must be root or be the same user as target process
- your own process:
  - `raise( int signo )`
- a thread in your own process:
  - `pthread_kill( pthread_t tid, int signo);`

To handle a signal:

```
sigaction (int signo, struct sigaction *action,
           struct sigaction *oldaction);
signal (int signo, void (*func)(int));
```

- **`signo`** is the signal affected
- both modify the same thing
- *signal()* is the easier function to use
  - to ignore a signal, pass **`SIG_IGN`** as the **`func`** parameter
  - to restore the default behavour, pass **`SIG_DFL`**
- for more complete control, use *sigaction()*...

☞ This is a process level setting

Signals

sigaction(..., action, oldaction):

- are structures of type **sigaction**:

```
struct sigaction {
  union {
      (*sa_handler) ();
      (*sa_sigaction) (int signo, siginfo_t *info, void *other);
  };
  int      sa_flags;
  sigset_t sa_mask;
};
```

- to specify the behavior when the signal is received:
  - set the **sa_handler**/**sa_sigaction** union to the address of a signal handler function, **SIG_DFL**, or **SIG_IGN**
  - if providing a signal handler, set **sa_mask** to additional signals to be masked while in the handler
- to have the signal be queued:
  - set the **sa_flags** to **SA_SIGINFO**

Handling a signal:

```
void
myHandler (int signo, siginfo_t *info, void *extra)
{
  // signo tells us which signal it was
  // info has extra information (si_code and si_value)
}

main()
{
  struct sigaction        action;

  action.sa_sigaction = myHandler; // set up handler
  sigemptyset (&action.sa_mask);   // no other signals to be
                                   // blocked during execution
                                   // of the handler
  action.sa_flags = SA_SIGINFO;    // request signal queuing
  sigaction (SIGRTMIN+7, &action, NULL);    // handle RT signal #7
  ...
}
```

To mask or unmask signals:

```
pthread_sigmask (int how, sigset_t *set,
                 sigset_t *oldset)
```

how

- one of:
  - **SIG_BLOCK** add the signals in **set** to the current mask (i.e. mask them)
  - **SIG_UNBLOCK** remove the signals in **set** from the current mask (i.e. unmask them)
  - **SIG_SETMASK** replace the current mask with **set**

set

- the signals to mask and/or unmask

☞ Every thread has its own signal mask

Signals - Manipulating sigset_t

There are functions for manipulating the sigset_t:

```
sigemptyset( sigset_t *set );
```

- remove all signals from set

```
sigfillset( sigset_t *set );
```

- add all signals to set

```
sigaddset( sigset_t *set, int signo );
```

- add one signal to set

```
sigdelset(sigset_t *set, int signo );
```

- remove one signal from set

Note: these change a local value, that must then be passed to one of the other functions

Masking SIGHUP (hangup signal):

```
// declare a local variable to hold
// the signals to be masked
sigset_t myset;

sigemptyset( &myset ); // clear
// add SIGHUP to myset
sigaddset( &myset, SIGHUP );
// mask the signal
pthread_sigmask( SIG_BLOCK, &myset, NULL );
```

To wait for signals:

```
sigwaitinfo (sigset_t *set, siginfo_t *info)
```

`set`

defines the set of signals that are to be waited for. If a signal in the set is already pending, does not wait.

`info`

gathers information about the signal into this info structure.

Optional, can be **`NULL`**.

Waiting for a signal:

```
/* mask the SIGRTMIN+0 signal so that it will not interrupt us */
sigemptyset(&mask);
sigaddset(&mask, SIGRTMIN+0);
sigprocmask(SIG_BLOCK, &mask, NULL);

/* make it a queued signal, normally SIG_DFL would cause termination
   but we've masked the signal */
action.sa_handler = SIG_DFL;
action.sa_flags = SA_SIGINFO;      /* this makes it a queued signal */
sigaction(SIGRTMIN+0, &action, NULL);

while (1) {
    sigwaitinfo(&mask, &info);     /* block, waiting for the signal */
    ... /* we received a signal, info.si_signo has the signal no. */
 }
```

Some important points:

- only certain functions are safe to call from a signal handler
  - in the Library Reference manual, the documentation for each function says if it's safe to call. Also in the Library Reference manual is a section called "Summary of Safety Information" that lists them
- if a signal interrupts a thread while the thread is blocked and a signal handler has been set up, when the signal handler returns the thread <u>will no longer be blocked</u>
  - usually the blocking function returns **`-1`** and **`errno`** is set to **`EINTR`**
  - remember: many library functions call *MsgSend*()* and would be affected (*open()*, *read()*, *write()*, *mmap()*, *spawn()*, ...)
  - to solve this, either:
    - always check return values or
    - mask all signals around blocking calls or
    - have the signal go to another thread

Topics:

- Native QNX Neutrino Messaging
- Pulses
- Signals
- → Event Delivery
- Shared Memory
- Pipes, POSIX Msg Queues
- POSIX fd/fp Based Functions
- Conclusion

Event Delivery

Events are a form of notification:

- can come from a variety of places
- can arrive in the form of pulses, signals, can unblock an *InterruptWait()*, ...

Event Delivery Example

Main idea: Client picks how it is woken up:

1. client prepares a structure describing what event it wants to receive and sends it along with some request to the server
2. server receives, stores the event description somewhere, and responds with "I'll do the work later"
3. when the server has completed the work, it delivers the event, which the client then receives. At that time the client can send another message asking for the results of the work

☞ Notice that at no time does the server have to know what type of event the client will get

Event Delivery - struct sigevent

Question: So, what's in the sigevent structure?

```
#define sigev_coid sigev_signo // reuse signo
struct   sigevent {
  int           sigev_notify;
  int           sigev_signo;
  union sigval  sigev_value;      // int, or void*
  short         sigev_code;
  short         sigev_priority;
};
```

Answer: Anything the client wants!

Event Delivery - Notification types

The client specifies what kind of event it wants to receive:

`sigevent.sigev_notify` can be:

- **`SIGEV_SIGNAL`**
- **`SIGEV_SIGNAL_CODE`**
- **`SIGEV_SIGNAL_THREAD`**

} send a *signal*

- **`SIGEV_PULSE`** ← send a *pulse*
- **`SIGEV_INTR`** ← used with interrupts
- **`SIGEV_UNBLOCK`** ← used with kernel timeouts only

Event Delivery - pulse example

Here's the data for a typical pulse:

```
chid = ChannelCreate (...);

sigevent.sigev_notify = SIGEV_PULSE;          // it's a pulse
sigevent.sigev_coid =
       ConnectAttach (0, 0, chid, ...);       // to self
sigevent.sigev_priority = MyPriority;         // our priority
sigevent.sigev_code = OUR_CODE;               // our PULSE ID
sigevent.sigev_value.sival_int = value;       // 32bits of data
```

or

```
chid = ChannelCreate (...);
/* connection to our channel */
coid = ConnectAttach (0, 0, chid, _NTO_SIDE_CHANNEL, flags);

SIGEV_PULSE_INIT (&sigevent, coid, MyPriority, OUR_CODE, value);
```

The client then gives this sigevent structure to whomever is going to be delivering it.

Topics:

- Native QNX Neutrino Messaging
- Pulses
- Signals
- Event Delivery
- → Shared Memory
- Pipes, POSIX Msg Queues
- POSIX fd/fp Based Functions
- Conclusion

Shared Memory:

- memory is automatically shared between threads in a process
- memory can be shared between different processes as well
  - but it requires extra work to set this up

Shared Memory

The same RAM is seen by two different processes:

Shared Memory - Setup

To set up shared memory:

```
fd = shm_open( "/myname", O_RDWR|O_CREAT, 0666 );
```

- name should start with leading `/` and contain only one `/`
- using `O_EXCL` can help do synchronization for the case where you have multiple possible creators

```
ftruncate( fd, SHARED_SIZE );
```

- this allocates `SHARED_SIZE` bytes of RAM associated with the shared memory object
  - this will be rounded up to a multiple of the page size, 4K

```
ptr = mmap( NULL, SHARED_SIZE, PROT_READ|PROT_WRITE,
            MAP_SHARED, fd, 0 );
```

- this returns a virtual pointer to the shared memory object
- the next step would be to initialize the internal data structures of the object

```
close(fd);
```

- you no longer need the fd, so you can close it

Shared Memory - Access

To access a shared memory object:

```
fd = shm_open( "/myname",O_RDWR );
```

- same name that was used for the creation

```
ptr = mmap( NULL, SHARED_SIZE, PROT_READ|PROT_WRITE,
            MAP_SHARED, fd, 0 );
```

- for read-only access (view), don't use **PROT_WRITE**
- you can gain access to sub-sections of the shared memory by specifying an offset instead of **0**, and a different size
  - both must be on pagesize boundaries

```
close(fd);
```

- you no longer need the fd, so you can close it

Shared Memory - Cleanup

The allocated memory will be freed when there are no further references to it:

- each fd, mapping, and the name is a reference
- can explicitly close, and unmap:

```
close(fd);
munmap( ptr, SHARED_SIZE );
```

- on process death, all fds are automatically closed and all mapping unmapped
- the name must be explicitly removed:

```
shm_unlink( "/myname" );
```

- during development and testing this can be done from the command line:

```
rm /dev/shmem/myname
```

Shared Memory

Problems with shared memory:

- readers don't know when data is stable
- writers don't know when it is safe to write

These are synchronization problems.

Let's look at a few solutions...

Shared Memory - Synchronization

There are a variety of synchronization solutions:

- thread synchronization objects in the shared memory area
  - if using *sem_init()* the **`pshared`** parameter must be non-zero
  - mutexes and conditional variables need the **`PTHREAD_PROCESS_SHARED`** flag
- *atomic_*()* functions for control variables
- IPC
  - *MsgSend()/MsgReceive()/MsgReply()* has built-in synchronization
  - use the shared memory to avoid large data copies

IPC for synchronization:

1. client prepares shared memory contents - update animation image
2. client tells server that a new image is ready - and waits for reply
3. server processes the image that is in the shared memory
4. server replies so that client can prepare another image

Since the *MsgSend()* does not return until the server calls *MsgReply()* this synchronises access to the shared memory.

Topics:

- Native QNX Neutrino Messaging
- Pulses
- Signals
- Event Delivery
- Shared Memory
- → Pipes, POSIX Msg Queues
- POSIX fd/fp Based Functions
- Conclusion

Pipes, Message Queues are:

- built on top of native messaging
- go through a server that buffers the data and deals with any other complexities

Pipes

Pipes are:

- slower than messages
- good for porting from UNIX source
- created by *pipe()*, *popen()* or *mkfifo()*

☞ In order to use pipes you must have the **`pipe`** process running.

POSIX Message Queues have:

- structure
- prioritization
- control

☞ In order to use these message queues you must have the **`mqueue`** process running.

Topics:

- Native QNX Neutrino Messaging
- Pulses
- Signals
- Event Delivery
- Shared Memory
- Pipes, POSIX Msg Queues
- → POSIX fd/fp Based Functions
- Conclusion

POSIX fd/fp Based Functions

If a server is a resource manager then:

- clients can communicate with it using POSIX file descriptor or file pointer based functions

```
resmgr_attach(…, "/dev/myname", …);
while (1) {
  ctp = dispatch_block(…);
  dispatch_handler(…);
}
```

```
fd = open("/dev/myname", …);
read (fd, bufs, nbytes);
...
```

- under the covers *open()*, *read()*, ... are calling *MsgSend*()* and *dispatch_block()* is calling *MsgReceive()*
- what's going on in the resource manager? ...

What's going on in the resource manager?

- the resource manager library function, *dispatch_handler()*, sees the message from *read()* and calls your resource manager's read handler function (e.g. *io_read()*)

- since the client is doing a *read()*, your io_read handler would usually reply with data

Why be a resource manager?

- client's API is:
  - standard functions that everyone knows
  - common for all your resource managers, though you can still use other IPC mechanisms too
- resource manager side is not standard but this architecture where you provide open/read/… handlers is common with drivers in many other OSes
- can use command line tools to communicate with your resource manager:

```
echo valve03=off >/dev/valve/control
cat /dev/valve/status
```

this makes testing/diagnosing problems easier

When not to be a resource manager?

- when you're not receiving messages
  - but you could still have a thread that implements a resource manager so that you can give/get information from the command line
- when speed is an issue
  - resource managers are a layer on top of native messaging and so are slower
  - however, a resource manager can still use pulses, shared memory, ...

Topics:

- Native QNX Neutrino Messaging
- Pulses
- Signals
- Event Delivery
- Shared Memory
- Pipes, POSIX Msg Queues
- POSIX fd/fp Based Functions
- → Conclusion

Conclusion - Choosing IPC

You've learned how to use a variety of forms of IPC under QNX Neutrino

- QNX Native messaging:
  - *MsgSend*()/MsgReceive()/MsgReply()*
- pulses
  - *MsgSendPulse()* or *MsgDeliverEvent()*
- POSIX signals
- Shared Memory

And seen a few others. How do you choose which to use?

Conclusion - Choosing IPC

Choosing IPC:

- QNX Native Messaging
  - client-server or RPC model
  - includes inherent synchronization
  - copies any size data
  - carries priority information
- Pulses
  - non-blocking notification compatible with QNX native messaging
  - only 39 bits of data
  - carry priority information

Conclusion - Choosing IPC

Choosing IPC:

- Signals
  - POSIX
  - non-blocking notification
  - interrupts target, making receiving difficult
  - does not carry priority information
- Shared Memory
  - POSIX
  - can eliminate need for a data copy
  - requires some additional way of synchronizing
    - if mutexes are used, they carry priority information
  - not network distributable

Conclusion - Choosing IPC

Choosing IPC:

- Pipes
  - POSIX
  - built on QNX native messaging
  - slow
    - 2 copies of data
    - more context switches
  - do not carry priority information
  - mostly for porting existing code
- POSIX message queues
  - basically Pipes with extra features

Conclusion - Choosing IPC

Choosing IPC:

- TCP/IP
  - built on QNX messaging
  - slow for local communication
    - 2 copies of data
  - POSIX
  - best way to communicate to a non-QNX machine
- fd/fp to a resource manager
  - built on QNX messaging
  - provides POSIX interface for clients
    - server must be QNX messaging aware
  - works well as a driver interface

Conclusion - Choosing IPC

Look at what you need for your IPC, and the features each offers. Some things to think about:

- Is POSIX a requirement?
- How much data is being moved?
- Can I afford to block?
- Do I need to communicate across a network?
- Can I use a combination of these in different places?

References

System Architecture (QSSL)

Programmer's Guide (QSSL)
Writing a Resource Manager

POSIX 1003 series specifications (IEEE)

W. Richard Stevens, *Advanced Programming in the Unix Environment*, Addison Wesley, 1993, ISBN 0-201-56317-7

W. Richard Stevens, *Unix Network Programming*, Prentice Hall, 1998, ISBN 0-13-490012-X

Rob Krten, *Getting Started with QNX Neutrino 2*, Parse Software Devices, 1999, ISBN 0-9682501-1-4

Time

Introduction

You will learn how:

- QNX Neutrino handles time
- to read and update the system clock
- to use system timers and kernel timeouts

Time

Topics:

→ **Timing Architecture**

Getting and Setting the System Clock

Timers

Kernel Timeouts

Conclusion

QNX® Neutrino®'s Concept of Time:

Ticksize

Ticksize:

- if your processor is >= 40MHz then the default ticksize is 1ms
- if your processor is < 40MHz then the default ticksize is 10ms
- this means *all* timing will be based on a resolution <u>no better</u> than 1ms or 10ms respectively

Note that the clock cannot usually be programmed for exactly 1ms in which case we use the next lowest value that the clock can do (e.g. on IBM PC hardware, you will actually get 0.999847ms).

Time Slice

The time slice:

- is 4 times the ticksize so it defaults to either 4ms or 40ms
- the multiplier, 4, cannot be changed. If you change the ticksize then the time slice will also change

Time

Topics:

- Timing Architecture
- → Getting and Setting the System Clock
- Timers
- Kernel Timeouts
- Conclusion

Let's look at the time of day functions:

Time Representation

QNX Neutrino time representation:

- internally stores time as 64-bit nanoseconds since 1970
- POSIX uses a struct timespec
  - 32-bit seconds and 32-bit nanoseconds since last second
  - QNX Neutrino uses an unsigned interpretation for the seconds since 1970

Keeping Track of Time

At bootup time:

- the kernel is given the current date and time from somewhere (battery backed up clock as on a PC, GPS, NTP, some atomic clock, ...)
- from then on, every tick, the kernel adds the ticksize to the current time
  - e.g. for a 1ms tick, every 1ms the kernel adds 1ms to the current time

Read and/or Set the System Clock:

```
struct timespec tval;

clock_gettime( CLOCK_REALTIME, &tval );
tval.tv_sec += (60*60)*24L; /* add one day */
tval.tv_nsec = 0;
clock_settime( CLOCK_REALTIME, &tval );
```

We can adjust the time:

- Bring it forward by one second:

```
struct _clockadjust new, old;

new.tick_nsec_inc = 10000; // 1e4 ns == 10 µs
new.tick_count = 100000;   // 100k * 10µs = 1s
ClockAdjust (CLOCK_REALTIME, &new, &old);
```

- Bring it backward by one second:

```
new.tick_nsec_inc = -10000;// 1e4 ns == 10 µs
new.tick_count = 100000;   // 100k * 10µs = 1s
ClockAdjust (CLOCK_REALTIME, &new, &old);
```

In both above examples the total adjustment will usually take 100 seconds (100k ticks (for a 1 msec tick size) = 100 seconds.)

Time

Topics:

- Timing Architecture
- Getting and Setting the System Clock
- → Timers
- Kernel Timeouts
- Conclusion

Let's look at timers:

Setting a Timer

To set a timer, the process chooses:

- what kind of timer
  - periodic
  - one shot
- timer anchor
  - absolute
  - relative
- event to deliver upon trigger
  - fill in an EVENT structure

☞ If preparing to use power management, make sure you stop your periodic timers when you don't need them.

POSIX realtime timer functions:

```
timer_create (clockID, &event, &timerID);
timer_delete (timerID);
```

Create / Destroy Timers

```
timer_gettime (timerID, &itime);
timer_settime (timerID, flags, &newtime, &oldtime);
```

Query/Set/Clear Timers

```
timer_getoverrun (timerID);
```

For use with signals

Timer example:

We want to have the server receive maintenance timer messages every 1.2 seconds, so that it can go out and perform housekeeping duties / integrity checks, etcetera.

continued

Timer Example - Setting the timer

Timer example (continued):

```
#define TIMER_PULSE_CODE        _PULSE_CODE_MINAVAIL+2
struct  sigevent                sigevent;
struct  itimerspec              itime;
timer_t                         timerID;
int                             coid;

coid = ConnectAttach (..., chid, ...);
SIGEV_PULSE_INIT (&sigevent, coid, maintenance_priority,
    TIMER_PULSE_CODE, 0);
timer_create (CLOCK_REALTIME, &sigevent, &timerID);

itime.it_value.tv_sec = 1;
itime.it_value.tv_nsec = 500000000; // 500 million nsecs=.5 secs
itime.it_interval.tv_sec = 1;
itime.it_interval.tv_nsec = 200000000; // .2 secs
timer_settime (timerID, 0, &itime, NULL);
```

Fill in the sigevent to request a PULSE when the timer expires

Specify an expiry of 1.5 seconds

Repeating every 1.2 seconds thereafter

Relative, not absolute

continued

Timer Example - Receiving the timer pulses

Timer example (continued):

```
typedef union {
   struct _pulse   pulse;
   // other message types you will receive
} myMessage_t;

myMessage_t   msg;
    ... // the setup code from the previous page goes here
    while (1) {
      rcvid = MsgReceive (chid, &msg, sizeof(msg), NULL);
      if (rcvid == 0) {
        // it's a pulse, check what type...
        switch (msg.pulse.code) {
          case  _TIMER_PULSE_CODE:
                periodic_maintenance();
                break;
        }
      }
```

Query Functions

How much time is left before expiry?

```
struct itimerspec  timeLeft;

timer_gettime (timerID, &timeLeft);
```

Returns:

`timeLeft.it_value.tv_sec` `timeLeft.it_value.tv_nsec`	Time left before expiry, or zero if timer is disarmed
`timeLeft.it_interval.tv_sec` `timeLeft.it_interval.tv_nsec`	Timer reload value, zero if timer is a one-shot, nonzero if timer is a repetitive timer.

Cancelling a Timer

Often you'll want to cancel a timer without destroying it (i.e. without removing it from the timer list)

- useful if you will frequently be canceling it then restarting it
- to cancel a timer:

```
struct itimerspec itime;
itime.it_value.tv_sec = 0;
itime.it_value.tv_nsec = 0;
timer_settime (timeID, 0, &itime, NULL);
```

- to restart it simply fill in the timing and call *timer_settime()* again

Time

Topics:

- Timing Architecture
- Getting and Setting the System Clock
- Timers
- → Kernel Timeouts
- Conclusion

Setting Timeouts

The kernel provides a timeout mechanism:

```
#define BILLION        1000000000
#define MILLION        1000000
struct sigevent        event;
uint64_t               timeout;

event.sigev_notify = SIGEV_UNBLOCK;

timeout = (2 * BILLION) + 500 * MILLION;

flags = _NTO_TIMEOUT_SEND | _NTO_TIMEOUT_REPLY;

TimerTimeout (CLOCK_REALTIME, flags, &event, &timeout, NULL);
MsgSend (…);   // will time out in 2.5 seconds
```

Specify the event

Length of time (2.5 seconds)

Which blocking states

Setting Timeouts

Some notes on timeouts:

- timeout is relative to when *TimerTimeout()* is called
- the timeout is automatically cancelled when the next kernel call returns
  - therefore you should not do anything else between the call to *TimerTimeout()* and the function that you are trying to timeout
  - but what if a signal handler is called? Might there be kernel calls made there? The same does not apply if the kernel call is made from within a signal handler that had preempted

Setting Timeouts

It can be used for checking/cleanup:

No time means “do not block”

```
event.sigev_notify = SIGEV_UNBLOCK;
flags = _NTO_TIMEOUT_RECEIVE;
/* loop, receiving (cleaning up) all pulses in receive queue */
do {
    /* MsgReceivePulse() wont block, if there’s a pulse it
     * will return 0, otherwise it will timeout immediately
    */
    TimerTimeout (CLOCK_REALTIME, flags, &event, NULL, NULL);
    rcvid = MsgReceivePulse (chid, &pulse, ...);
} while (rcvid != -1);
/* if errno is ETIMEDOUT, then we got all the pulses */
```

This practice is not recommended for implementing polling since polling in general is wasteful of CPU. Usage like above is fine.

Topics:

- Timing Architecture
- Getting and Setting the System Clock
- Timers
- Kernel Timeouts
- → Conclusion

Conclusion

You learned:

- ticksize is the fundamental quantum of time
- how to set or gradually adjust the system time
- how to get periodic notification
- how to timeout kernel calls

Interrupts

Introduction

You will learn:

- how QNX Neutrino handles hardware interrupts
- how to handle interrupts in your code
- different interrupt handling strategies

Interrupts

Topics:

→ **Concepts**

IPC from ISR

Handler Architecture

Conclusion

Interrupt Handling:

All hardware interrupts are vectored to the kernel.

A process can either:

- register a function to be called by the kernel when the interrupt happens
- request notification that the interrupt has happened

Interrupt Scheduling (preemptive):

Interrupt Scheduling (non-preemptive):

Interrupts - Concepts

Interrupts:

- have a higher priority than any thread priority
- handlers are pre-emptable by higher priority interrupts
  - requires platform support
- can be handled by user-space processes
  - drivers can be dynamically loaded
  - drivers are not in kernel space
- handlers are called by the kernel
  - have kernel privilege level

Interrupt Latency:

What affects interrupt latency?

- time spent with interrupts disabled
- time spent in an equal or higher priority interrupt handler
- time spent in other handlers for this interrupt
- time spent with the particular interrupt level masked
  - common with shared interrupts and *InterruptAttachEvent()*

Interrupt calls:

```
id = InterruptAttach (int intr,
                      struct sigevent *(*handler)(void *, int),
                      void *area, int size, unsigned flags);
id = InterruptAttachEvent (int intr, struct sigevent *event,
                           unsigned flags);
InterruptDetach (int id);
InterruptWait (int flags, uint64_t *reserved);
InterruptMask (int intr, int id);
InterruptUnmask (int intr, int id);
InterruptLock (struct intrspin *spinlock);
InterruptUnlock (struct intrspin *spinlock);
```

☞ You must have I/O privity for the above functions to work. To get I/O privity you call *ThreadCtl(_NTO_TCTL_IO, 0)* and you must have root (user id 0) permissions.

Handling an Interrupt

The simplest possible interrupt handler (none at all!):

```
struct sigevent event;

main ()
{
  ThreadCtl (_NTO_TCTL_IO, 0);
  SIGEV_INTR_INIT (&event);
  id = InterruptAttachEvent (intnum, &event, ...);
  for (;;) {
    InterruptWait (0, NULL);
    // do the interrupt work here, at thread priority
    InterruptUnmask (intnum, id);
  }
}
```

InterruptAttachEvent

To associate a event with an interrupt vector:

```
id = InterruptAttachEvent (intr, event, flags);
```

☞ Remember, you must have I/O privity for this to work.

The kernel automatically unmasks the interrupt once an event has been associated with it.

Another simple interrupt handler:

```
struct sigevent event;

const struct sigevent *
handler (void *not_used, int id)
{
  if (check_status_register())
    return (&event);
  else
    return (NULL);
}
main ()
{
  ThreadCtl (_NTO_TCTL_IO, 0);
  SIGEV_INTR_INIT (&event);
  id = InterruptAttach (intnum, handler, NULL, 0, ...);
  for (;;) {
    InterruptWait (0, NULL);
    // do some or all of the work here
  }
}
```

InterruptAttach

To associate a handler with an interrupt vector:

```
id = InterruptAttach (intr, handler, area, size, flags);
```

☞ Remember, you must have I/O privity for this to work.

The kernel automatically unmasks the interrupt once a handler has been associated with it.

Interrupt Numbers

Logical interrupt vector numbers are:

- defined by startup
- documented in the build file for your board, e.g.

```
# Interrupt Assignments
# ---------------------
#
# vector:   0 (PPC800_INTR_IRQ0)
# trigger:  falling edge
# device:   unassigned
...
# vector:   15 (PPC800_INTR_LVL7)
# trigger:  N/A
# device:   Programmable Interval Timer interrupt (system timer)
...
# vector:   0x8001001e (PPC800_INTR_CPMSCC1)
# trigger:  N/A
# device:   CPM SCC1
```

Build files live in $QNX_TARGET/*cpu*/boot/build.

InterruptAttach*() Flags

InterruptAttachEvent and InterruptAttach's flag parameter can contain:

`_NTO_INTR_FLAGS_END`

The kernel maintains a queue of handlers and events for each interrupt. This flag says the new handler or event should be added to the end of the queue rather than to the start.

By combining a known startup order for drivers with the choice of using or not using this flag, the order of handling for shared interrupts can be explicitly configured.

continued...

InterruptAttachEvent and InterruptAttach's flag parameter (continued):

`_NTO_INTR_FLAGS_PROCESS`

Associate the handler or event with the process. Normally it is associated with the thread that did the attached. If that thread dies then the handler is detached. With this flag, when the thread dies, the handler remains attached to the process.

You should use an event that is directed at a process, such as a pulse or signal.

continued...

InterruptAttachEvent and InterruptAttach's flag parameter (continued):

`_NTO_INTR_FLAGS_TRK_MSK`

This flag specifies that when the application detaches from the interrupt, the kernel should perform the proper number of unmasks to ensure that the interrupt functions normally.

☞ ALWAYS set this flag.

Controlling when interrupts happen:

- *InterruptMask()* and *InterruptUnmask()* mask and unmask a specific interrupt
  - on SMP these are masked for all CPUs
- to unmask an interrupt, you must do the same number of unmasks as there were masks
  - this is done using a system wide count for each interrupt
- *InterruptLock()* and *InterruptUnlock()* disable and enable all interrupts as well as use a spinlock for SMP synchronization
  - there is no count associated with these
  - making a kernel call enables interrupts
  - on SMP the enable and disable are per CPU

Interrupt association:

Interrupt source is masked off

Initial state

Interrupt handler IH1is associated with the interrupt source, and the source is enabled

```
id1 = InterruptAttach (SRC3, IH1, NULL, 0, ...);
```

event2 gets put in FRONT of the existing handler (the default)

```
id2 = InterruptAttachEvent (SRC3, &event2, ...);
```

Attach & Detach

Interrupt association:

event2 is detached, has
no effect on existing interrupt handlers
in the chain

```
InterruptDetach (id2);
```

Interrupt handler IH3
attached at end of chain

```
id3 = InterruptAttach (SRC3, IH3, NULL, 0, _NTO_INTR_FLAGS_END |
                                           _NTO_INTR_FLAGS_TRK_MSK);
```

Interrupt handler IH1 is detached, has
no effect on existing interrupt handlers
in the chain

```
InterruptDetach (id1);
```

When last interrupt handler is detached
The kernel automatically masks the interrupt

```
InterruptDetach (Id3);
```

Should you attach a handler or an event?

- The kernel is the single point of failure for a QNX system, attaching a handler increases the size of the SPOF, an event does not
- debugging is far simpler with an event
  - ISR code can not be stepped/traced with the debugger
- full OS functionality when doing h/w handling in a thread
- events impose far less system overhead at interrupt time than handlers
  - no need for the MMU work to gain access to process address space if using an event
- scheduling a thread for every interrupt could be more overhead, if you could do some work at interrupt time and only need to schedule a thread some of the time
- handlers have lower latency than getting a thread scheduled
  - does your hardware have some sort of buffer or FIFO? If not then you might not be able to wait until a thread is scheduled

Interrupt Handler Environment

An interrupt handler operates in the following environment:

- it is sharing the data area of the process that attached it
- the environment is very restricted:
  - cannot call kernel functions except *InterruptMask()*, *InterruptUnmask*() and *TraceEvent()* (see notes)
  - cannot call any function that might call a kernel function
    - the documentation for each function specifies whether or not that function is safe to call from an interrupt handler
    - there is also a section in the Library Reference manual called “Summary of Safety Information” that lists all safe functions
  - the interrupt handler is using the kernel’s stack, keep stack usage small (if you have a lot of data use variables defined outside of the handler rather than variables defined local to the function, don’t go too many function levels deep)
  - can't do floating point

Interrupt Handler Environment

Why the previous restrictions?

- latency
  - the kernel does the minimum amount of work to get to the interrupt handler
  - does not save/restore floating point context or registers as that is expensive
  - does not make the kernel re-entrant from interrupt handlers

What happens if you break those rules?

- if you're lucky, the OS crashes immediately
- if you're not lucky, you've corrupted internal kernel data structures and something bad will happen later

Topics:

- Concepts
- → IPC from ISR
- Handler Architecture
- Conclusion

IPC Methods

We have the following IPC methods:

- SIGEV_INTR/*InterruptWait()*
  - simplest to use and fastest
  - must dedicate a thread
  - queue is only 1 entry deep
- Pulse
  - can have multiple threads waiting to receive on the channel
  - are queued
  - most flexible
- Signal
  - most expensive solution if using a signal handler but slightly faster than a pulse if waiting with *sigwaitinfo()*
  - can be queued

InterruptWait

Simplest way to wait for an interrupt to happen is to call InterruptWait():

```
InterruptWait (reserved, reserved);
```

The waiting thread must be the ***same thread*** that attached the handler in the first place!

Note: the interrupt queue is one entry deep -- if multiple interrupts occur before *InterruptWait()* is called, *InterruptWait()* will return immediately. But, the next time *InterruptWait()* is called, it will wait for an interrupt to occur before returning.

Using a pulse for notification:

```
#define INTR_PULSE _PULSE_CODE_MINAVAIL
struct sigevent event;
main () {
  ...
  chid = ChannelCreate( 0 );
  coid = ConnectAttach( ND_LOCAL_NODE, 0, chid, _NTO_SIDE_CHANNEL, 0 );
  SIGEV_PULSE_INIT( &event, coid, MyPriority, INTR_PULSE, 0 );
  InterruptAttach( intnum, handler, NULL, 0, _NTO_INTR_FLAGS_TRK_MSK );
  for (;;) {
    rcvid = MsgReceive( chid, ... );
    if (rcvid == 0) {
      // we got a pulse
  }
}
const struct sigevent *
handler (void *area, int id) {
   // do whatever work is required
   return (&event); // wake up main thread
}
```

Multiple Priorities

Or a combination:

```
const struct sigevent *
intHandler (void *not_used, int id)
{
    ...
    if (nothing_to_report) {
        return (NULL);
    } else {
        if (low_priority_event) {
            return (&lowpri_event);
        } else {
            return (&highpri_event);
        }
    }
    ...
}
```

Low priority pulse

SIGEV_INTR going to
a high priority thread

Topics:

- Concepts
- IPC from ISR
- → Handler Architecture
- Conclusion

Interrupt Architecture

There is a range of approaches to interrupt handlers:

- From Minimum Handler
  - does as little as possible
  - does rest of work in a thread
  - allows threads to have CPU as per priorities
- To Maximum Handler
  - does most of the work (if not entire task)
  - does little or no work in “main” thread
  - uses up CPU at highest priority while active

Most real-world interrupt handlers will fall somewhere in between.

Minimum

The ultimate MINIMUM handler:

```
InterruptAttachEvent (intNum, &event, _NTO_INTR_FLAGS_TRK_MSK);
```

We don't need a handler if all we are going to do is use it to wake up a thread. Instead, just give the kernel an event. When the interrupt goes off, the kernel will then mask the interrupt and process the event (schedule the thread.) The thread then does the work and unmasks the interrupt.

Minimum

Advantages:

- minimum amount of CPU spent just in the kernel
- handler's function is obvious since there is none, no need to debug!
- built-in throttling of high frequency interrupts since the kernel masks the interrupt before waking up the thread and the thread unmasks it after doing the work

Disadvantages:

- masking may introduce latency for other drivers on a shared interrupt
- scheduling latency may be too high

A Maximum handler:

```
const struct sigevent *
maximumHandler (void *area, int id)
{
  struct sigevent *pevent = (struct sigevent *) area;
  // perform all processing in handler.  For example, digital
  // signal processing for an audio application might need to
  // get a sample, perform some calculation, and then write a
  // sample back out quickly, and frequently…

  return (someNotableEvent ? pevent : NULL);
}

mainThread ()
{
  // initializations…
  while (1) {
    MsgReceive (...);    // messages from clients & handler
  }
}
```

Maximum

Advantages:

- all time-critical work gets done at interrupt priority
- non-time-critical work gets done as time permits

Disadvantages:

- interrupt handler can hog all available CPU on slower machines
- increases interrupt and maximum scheduling latency
- totally unsuitable as a general purpose solution
- does not work & play well with others

Topics:

- Concepts
- IPC from ISR
- Handler Architecture
- → Conclusion

Conclusion

You learned:

- that the kernel is the first handler for all interrupts
- that processes can register handlers or can register for notification of interrupts
- that interrupt handlers run in a very restricted environment
- a range of interrupt handling strategies

References

Programmer's Guide (QSSL)

Writing Interrupt Handlers

Freedom from Hardware and Platform Dependencies

System Architecture (QSSL)

Rob Krten, *Getting Started with QNX Neutrino 2*, Parse Software Devices, 1999, ISBN 0-9682501-1-4

I/O

I/O

Topics:

→ **Overview**

Memory Mapping

Port I/O

DMA

PCI

Overview

Hardware access:

Hardware devices can occupy memory, I/O, DMA, and / or interrupt "slots"...

Overview

Some example devices:

- USB on Bigsur-Amanda (SH4):
  - I/O at `0x1bFF5000`
  - memory at `0x18800000`
  - interrupt `0x800a001b`
- Serial port on rpx-lite (PPC):
  - I/O at `0x30000000`
  - interrupt `0x80010004`
- VGA card (x86):
  - I/O at `0x3B0`
  - memory at `0xA0000`

I/O

Topics:

- Overview
- → Memory Mapping
- Port I/O
- DMA
- PCI

Mapping Physical Memory

```
ptr = mmap_device_memory (0, 0x4000,
                          PROT_READ | PROT_WRITE | PROT_NOCACHE,
                          0, 0xa0000)
```

I/O

Topics:

- Overview
- Memory Mapping
- → Port I/O
- DMA
- PCI

Interfacing to I/O ports:

```
// enable I/O privilege for this thread
ThreadCtl (_NTO_TCTL_IO, NULL);

// get access to a devices registers
iobase = mmap_device_io (len, base_port);

val8 = in8 (iobase+N);      // read an 8 bit value
val16 = in16 (iobase+N);    // read a 16 bit value
val32 = in32 (iobase+N);    // read a 32 bit value

out8 (iobase+N, val8);      // write an 8 bit value
out16 (iobase+N, val16);    // write a 16 bit value
out32 (iobase+N, val32);    // write a 32 bit value
```

What does mmap_device_io() do?

```
uintptr_t mmap_device_io(size_t len, uint64_t io) {
#if defined(__X86__)
    return io;
#else
    return (uintptr_t) mmap64(0, len,
                    PROT_NOCACHE|PROT_READ|PROT_WRITE,
                    MAP_SHARED|MAP_PHYS, NOFD, io);
#endif
}
```

Port I/O - Under the covers

What do the in*() and out*() functions do?

- They are inline functions that are processor specific
- defined in:

  `${QNX_TARGET}/usr/include/${CPU}/inout.h`

- handle read and write operations for control registers in a properly synchronised way

Port I/O - Synchronisation

How various processors synchronize these:

- x86 uses special instructions for port I/O and a special address line
- PPC and MIPS have an "eieio" instruction: Enforce In-order Execution of I/O
- on SH4, the core will enforce the wait for completion of memory operations for certain address ranges before continuing
- on ARM, the core will enforce the wait for completion of memory operations for any uncached and unbuffered region

I/O

Topics:

- Overview
- Memory Mapping
- Port I/O
- → DMA
- PCI

DMA - Overview

Using DMA is:

- usually more complex than memory mapped/ io-mapped
- usually used for performance
- in some cases, the only way to talk to the hardware

DMA operations generally use physical addresses, but...

Allocated Memory Characteristics

For generally allocated memory:

– virtually contiguous isn't physically contiguous:

DMA Interface

To use DMA:

- allocate physically contiguous memory (via **`mmap`**)
  - use the following `prot` parameters:

    `PROT_NOCACHE | PROT_READ | PROT_WRITE`
  - and the following `flags` parameters:

    `MAP_ANON | MAP_PHYS | MAP_PRIVATE`
- so that the DMA controller knows where to transfer data, convert the virtual address to a physical address

Allocating Contiguous Physical Memory

```
ptr = mmap (0, len, PROT_READ | PROT_WRITE | PROT_NOCACHE,
            MAP_PHYS | MAP_ANON | MAP_PRIVATE, NOFD, 0)
```

Adding **MAP_PHYS** here with **MAP_ANON** means allocate physically contiguous zero filled memory.

DMA Interface

Convert the virtual address to a physical address via:

```
mem_offset (void *virtual_addr, int fd,
            size_t length, off_t *offset,
            size_t *contig_len);
mem_offset64 (void *virtual_addr, int fd,
              size_t length, off64_t *offset,
              size_t *contig_len);
```

- `virtual_addr` returned by *mmap()*
- `offset` will contain the physical address

Example:

```
mem_offset(virtual_addr, NOFD, len, &paddr, NULL);
// paddr now contains the physical address
```

– give the physical address to the DMA controller

DMA Interface

Operation:

2 (2) Software causes hardware device to initiate a DMA transfer

3 (3) Hardware device transfers data to data bus, and tells DMA controller to go to the next address

1 (1) DMA controller is initialized with the physical address and length of the transfer

3 (3) DMA controller generates addresses, and read/write requests to the bus

DMA Interface

The DMA controller will then generate an interrupt on a terminal count:

- this means the specified number of bytes has been transferred
- in your interrupt service routine:
  - reload the DMA controller with the next transfer
  - start DMA
  - perhaps notify the controlling thread that buffer is full/empty (as appropriate & as required)

I/O

Topics:

- Overview
- Memory Mapping
- Port I/O
- DMA
- → PCI

PCI API

To find and configure a PCI device:

- you must run one of the pci servers:
  `pci-bios`, `pci-p5064`, ...
- On x86:
  - the PCI BIOS configures the PCI space and sets up devices
  - configuration is done at boot time
  - `pci-bios` calls into the PCI BIOS to do the work
- On non-x86:
  - the pci server configures the PCI address space
  - configuration is often dynamic when drivers request it

PCI - The calls

The PCI calls include:

`pci_attach()`	connect to PCI server, will fail if no PCI bus or server not running
`pci_detach()`	disconnect from PCI server
`pci_find_device()`	find hardware by Device ID and Vendor ID
`pci_find_class()`	find hardware by class
`pci_attach_device()`	find hardware and get basic configuraton information
`pci_detach_device()`	release device configuration
`pci_read_config()`	read configuration information
`pci_read_config*()`	read blocks of 8/16/32-bit values
`pci_write_config()`	write configuration information
`pci_write_config*()`	write blocks of 8/16/32-bit values

PCI - Using the PCI calls

Generally when writing a driver you:

- know what vendor id(s) and device id(s) you can support
- connect to the pci server with *pci_attach()*
  - this verifies that there is a pci server (and bus) then connects to it
- use *pci_find_device()* to query the known id(s)
- call *pci_attach_device()* to configure and get configuration information for your device

Conclusion

You learned how to:

- access memory-mapped and io-mapped hardware
- allocate and use DMA safe memory
- find and access PCI devices

Resource Managers

Introduction

Topics:

→ Overview

A Simple Resource Manager

- Initialization
- Handling *read()* and *write()*

Device-Specific and Per-Open Data

Out-of-Band/Control Messages

The different methods

- io_msg method
- devctl method

Conclusion

Overview

What is a resource manager?

- a program that looks like it is extending the operating system by:
  - creating and managing a name in the pathname space
  - providing a POSIX interface for clients (e.g. *open()*, *read()*, *write()*, ...)
- can be associated with hardware (such as a serial port, or disk drive)
- or can be a purely software entity (such as **`/dev/mqueue`**)

Let's take a look at the pathname space

Name mapping:

Overview

The prefix tree:

- is the root of the pathname space
  - every name in the pathname space is a descendant of some entry in the prefix tree
- is maintained by the Process Manager
  - stored as a table
- Resource Managers add and delete entries
- associates a **`nd, pid, chid, handle`** with a name
- is searched for the longest slash delimited matching prefix

Overview

For example, to resolve the pathname:

/dev/ser1

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

The longest match is /dev/ser1, which points to **`devc-ser8250`**

Overview

A Client requests a service:

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

or

```
fp = fopen ("/home/bill/abc", "w");
```

which results in the client's library code (ultimately "**open**") sending a message to the process manager...

Overview

Interactions:

1. *open()* sends a "query" message
2. Process Manager replies with who is responsible (**`nd`**, **`pid`**, **`chid`**, **`handle`**)
3. *open()* establishes a connection to the specified resource manager (**`nd`**, **`pid`**, **`chid`**), and sends an open message (containing the **`handle`**)
4. Resource manager responds with status (pass/fail)

All further communication goes directly to the resource manager.

Resource Managers

Resource managers are built on message passing:

1. client calls the POSIX *write()*
2. this builds and sends a message
3. the resource manager receives and processes the `write` message appropriately
4. the resource manager replies with a result and this is returned to the client from *write()*

Overview

A resource manager performs the following steps:

- creates a channel
- takes over a portion of the pathname space
- waits for messages & events
- processes messages, returns results

Resource Manager Messages

There are three major types of messages:

- Connect messages:
  - pathname based (eg: **`open ("/dev/ser1", …)`**)
  - may create an association between the client process and the resource manager, which is used later for I/O messages
- I/O messages:
  - file descriptor (**`fd`**) based (eg: **`read (fd, …)`**)
  - rely on association created previously by connect messages
- Other:
  - pulses, private messages, etc

Connect Messages:

message	client call
`IO_OPEN`	*open()*
`IO_UNLINK`	*unlink()*
`IO_RENAME`	*rename()*

Defined in **`<sys/iomsg.h>`**

I/O Messages (frequently used):

message	client call
`IO_READ`	*read()*
`IO_WRITE`	*write()*
`IO_CLOSE`	*close()*
`IO_DEVCTL`	*devctl(), ioctl()*

Defined in **`<sys/iomsg.h>`**

continued

I/O Messages (continued):

```
IO_NOTIFY,        IO_STAT,
IO_UNBLOCK,       IO_PATHCONF,
IO_LSEEK,         IO_CHMOD,
IO_CHOWN,         IO_UTIME,
IO_LINK,          IO_FDINFO,
IO_LOCK,          IO_TRUNCATE,
IO_SHUTDOWN,      IO_DUP
```

Resource Manager Library

Writing resource managers is simplified greatly with a resource manager shared library that:

- simplifies main receive loop (table driven approach)
- has default actions for any message types that do not have handlers specified in tables,

Client calls go to handlers:

Resource Manager Structure

The resource manager layer:

Topics:

Overview

→ A Simple Resource Manager

- Initialization
- Handling *read()* and *write()*

Device-Specific and Per-Open Data

Out-of-Band/Control Messages

The different methods

- io_msg method
- devctl method

Conclusion

The Examples that we'll use

We're going to write the code for several Resource Managers:

Resource Manager	Registered Name	Description
`example`	**`/dev/example`**	does nothing, but in a good way
`time1` & **`time2`**	**`/dev/time/now`**	"**`cat /dev/time/now`**" prints the current time
" & "	**`/dev/time/min`**	"**`cat /dev/time/min`**" prints the minutes
" & "	**`/dev/time/hour`**	"**`cat /dev/time/hour`**" prints the hours

We'll illustrate the use of the resource manager shared library, and handling I/O messages with these examples…

The example Resource Manager

To talk about setting up a resource manager we'll use the **`example`** resource manager:

Client side:

- Here's how it behaves from a client's point of view:
  - **`read`** always returns 0 bytes
  - **`write`** of any size always works
  - other things behave as expected

The example Resource Manager - Setting things up

The example resource manager:

create & initialize structures:

- a dispatch structure
- list of *connect* message handlers
- list of *I/O* message handlers
- device attributes
- resource manager attributes
- dispatch context

Attach a pathname, passing much of the above

from the main loop:

- block, waiting for messages
- call a handler function, the handler function handles requests and performs callouts to your specified routines.

Resource Manager - Setting things up

1) Initialize the dispatch interface
2) Initialize the resource manager attributes
3) Initialize functions used to handle messages
4) Initialize the attribute structure used by the device
5) Put a name into the namespace
6) Allocate the context structure
7) Start the resource manager message loop

1 Setting things up - dispatch_create()

First, create a dispatch structure:

```
dispatch_t *dpp;

dpp = dispatch_create ();
```

dpp

- This is the glue the resource manager framework uses to hold everything together
- The contents are hidden. Among other things, it contains the chid. The **`dpp`** is pointed to by the dispatch context structure (**`dispatch_context_t`**).
- The channel is not actually created until the first attach call (*resmgr_attach()*, *pulse_attach()*, *message_attach()*, *select_attach()*).

2 Initialize the resource manager attributes

The resource manager attribute structure is used to configure:

- how many IOV structures are available for server replies (*nparts_max*)
- the minimum receive buffer size (*msg_max_size*)
- memset(&resmgr_attr, 0, sizeof resmgr_attr);
- resmgr_attr.nparts_max = 1;
- resmgr_attr.msg_max_size = 2048;

resmgr_attach() - rattr

The rattr contains at least the following members:

```
typedef struct _resmgr_attr {
    unsigned  nparts_max;
    unsigned  msg_max_size;
} resmgr_attr_t;
```

You can have the library do your replies for you, but the library will use *MsgReplyv()* to do it. This means that you will need to provide the library with an **`iov`** array that points to your data. The library provides such an array for you in the context structure (**`ctp`**). However, the array defaults to size one meaning it can handle only one buffer of data. If you will have more than one buffer then set **`nparts_max`** to the size you want the array to be. We'll see the use of this **`iov`** later.

`msg_max_size` is the size needed for the receive buffer, the actual

Setting things up - Connect and I/O functions

Next, we set up two tables of functions:

- connect functions
  - these are called as a result of POSIX calls that take a filename

    e.g.: *open (filename, ...)*, *unlink (filename)*, ...
- I/O functions
  - these are called as a result of POSIX calls that take a file descriptor

    e.g.: *read (fd, ...)*, *write (fd, ...)*, ...

Setting things up - Connect functions

The connection functions structure contains at least the following members:

```
typedef struct
  _resmgr_connect_funcs {
  unsigned nfuncs;
  int (*open) (/* actual prototype
  */);
  int (*unlink) (...);
  int (*rename) (...);
} resmgr_connect_funcs_t;
```

The I/O functions structure:

`resmgr_io_funcs_t`, has at least the following elements :

```
nfuncs,        read,          write,         close_ocb,
stat,          notify,
   devctl,        unblock,
pathconf,      lseek,         chmod,         chown,
utime,         openfd,        fdinfo,        lock,
space,         shutdown,      mmap,          msg,
umount,        dup,           close_dup,     lock_ocb,
unlock_ocb,    sync
```

io_funcs

io_read()
io_write()

Example of declaring and initializing the connect and the I/O functions structures:

```
resmgr_connect_funcs_t connect_funcs;
resmgr_io_funcs_t      io_funcs;

iofunc_func_init (_RESMGR_CONNECT_NFUNCS, &connect_funcs,
          _RESMGR_IO_NFUNCS, &io_funcs);

connect_funcs.open = io_open;
io_funcs.read   = io_read;
io_funcs.write  = io_write;
```

iofunc_func_init() places default values into the passed connect and I/O functions structures, based on the number of values that you have specified via the first and third integer arguments. It is recommended that you use the **_RESMGR_CONNECT_NFUNCS** and **_RESMGR_IO_NFUNCS** constants for those two arguments.

Next, put /dev/example into pathname space:

```
resmgr_attach( …,"/dev/example",… );
```

Setting things up - resmgr_attach() details

The parameters are:

```
id = resmgr_attach (dpp, &rattr, path,
file_type,flags, &connect_funcs,
&io_funcs, handle);
```

dpp = pointer returned by *dispatch_create()*

rattr = **NULL** or structure of further paramaters...

path = "**/dev/example**"

file_type = **_FTYPE_ANY**; (the usual case)

flags = **0** or control flags...

connect_funcs and **io_funcs** point to the tables of functions we just created

handle = pointer to device attribues

id = id of this pathname, used for *resmgr_detach()* call

resmgr_attach() - flags

The resmgr_attach(..., flags, ...):

`_RESMGR_FLAG_BEFORE` (default) and

`_RESMGR_FLAG_AFTER`

if the pathname has already been attached, let this resource manager handle it BEFORE or AFTER others which have attached the same pathname

`_RESMGR_FLAG_DIR`

allow pathnames that extend past the registered pathname to be handled by this resource manager

used by filesystem resource managers (eg: **`/cdrom/`**...)

resmgr_attach()

When you call resmgr_attach() you are:

- creating your device
- associating data and handlers with it

```
id = resmgr_attach (dpp, &rattr, "/dev/example", _FTYPE_ANY,
            NULL, &connect_funcs, &io_funcs, &ioattr);
```

☞ The library does not make copies of these structures

resmgr_attach()

resmgr_attach() puts the name in the pathname space:

- your resource manager becomes visible to clients
  - clients will expect you to be ready to handle messages
- before doing so you should have completed most of your initilization:
  - hardware detection and initialization
  - buffer allocation and configuration
- if something fails, don't attach name

Setting things up - dispatch_context_alloc()

Lastly, allocate a dispatch context structure:

```
dispatch_context_t *ctp;
ctp = dispatch_context_alloc (dpp)
```

- this is the operating parameters of the message receive loop
- it is passed to the blocking function and the handler function
- it contains things like the **`rcvid`**, pointer to the receive buffer, and message info structure
- it will be passed as the **`ctp`** parameter to your connect and I/O functions

Putting together what we have so far:

```
dpp = dispatch_create ();

iofunc_func_init (_RESMGR_CONNECT_NFUNCS,
  &connect_funcs,
                  _RESMGR_IO_NFUNCS, &io_funcs);
connect_funcs.open = io_open;
io_funcs.read   = io_read;
io_funcs.write  = io_write;

iofunc_attr_init (&ioattr, S_IFCHR | 0666, NULL,
                   NULL);

id = resmgr_attach (dpp, NULL, "/dev/example",
  _FTYPE_ANY,
            0, &connect_funcs, &io_funcs, &ioattr);

ctp = dispatch_context_alloc (dpp);
```

Setting things up - The dispatch loop

So now that everything's set, the loop:

```
while (1) {
    ctp = dispatch_block (ctp);
    dispatch_handler (ctp);
}
```

- *dispatch_block()* blocks waiting for messages,
- *dispatch_handler()* handles them, including calling any callout functions which you've provided (connect function, I/O functions)

The example Resource Manager

Let's see what happens when a client uses the new **`/dev/example`** device (by, for example, doing "**`cat /dev/example`**"):

Internally, "**`cat`**" basically does:

```
fd = open ("/dev/example", O_RDONLY);
while (read (fd, buf, BUFSIZ) != 0)
    /* write buf to stdout */
close (fd);
```

The example Resource Manager

Which results in:

- Communications with the Process Manager:
  - an inquiry message to the process manager:
    - "who is responsible for **`/dev/example`**?"
    - returns a reply, "(nd, pid, chid)" (our resource manager, **example**), "is responsible"
- Communications with **example**:
  - an open message
    - "open this device for read"
    - returns a reply, "yes, open succeeded, proceed"
    - the *open()* library call returns a file descriptor, **`fd`**
  - a read message
    - "get me some data"
    - returns a reply, "here are 0 bytes" (I.e. EOF)
  - a close message

The example Resource Manager

Let's look at the connect functions first.

example only defines the **io_open** function:

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

io_open arguments -- ctp

ctp

- pointer to a resource manager context structure
- information about the received message
- contains at least:

```
typedef struct _resmgr_context {
    int                     rcvid;
    struct _msg_info        info;
    resmgr_iomsgs_t         *msg;
    unsigned                msg_max_size;
    int                     status;
    int                     offset;
    int                     size;
    IOV                     iov [1];
} resmgr_context_t;
```

io_open arguments -- msg

```
io_open_t *msg:
    typedef union
    { // contains at least the following
      struct  _io_connect          connect;
    } io_open_t;

    struct    _io_connect
    { // contains at least the following
      uint16_t                     type;
      uint16_t                     sflag;
      uint32_t                     ioflag;
      uint32_t                     mode;
      uint16_t                     access;
      char                  path [1];    // variable length
    };
```

this is mostly used by filesystem resource managers, ones that set **_RESMGR_FLAG_DIR** and intend to manage a file system below their mount point

io_open arguments -- handle

handle

device structure
iofunc_attr_t

- the device attributes that we associated with the device when it was created (during the initial call to *resmgr_attach()*).
- In the case of **`/dev/example`**, we passed a **`iofunc_attr_t`** as the **`handle`** parameter in *resmgr_attach()*.

The `io_open` code:

```
int
io_open (resmgr_context_t *ctp, io_open_t *msg,
         RESMGR_HANDLE_T *handle, void *extra)
{
  printf("got an open message\n");
  return (iofunc_open_default (ctp, msg, handle,
  extra));
}
```

We didn't really need an open handler since the call to *iofunc_func_init()* inserted *iofunc_open_default()* as the function to call for *io_open()*. But you would use it if there were other

The example Resource Manager's io

Let's look at example's I/O functions:

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

They both share the **ocb**...

I/O functions arguments -- ocb

`ocb`

- Open Control Block
- one **`ocb`** per *open()*
- maintains context between the *open()* call and subsequent I/O calls, i.e. *iofunc_open_default()* allocates and initializes it, I/O functions use it.
- will be either an **`iofunc_ocb_t`**, or
- an **`iofunc_ocb_t`** encapsulated within your own structure with additional state information, etc.
- points to the attribute structure for the device opened

Topics:

Overview

A Simple Resource Manager

→ Initialization

– Handling *read()* and *write()*

Device-Specific and Per-Open Data

Out-of-Band/Control Messages

The different methods

– io_msg method

– devctl method

Conclusion

READ

The message you'll receive is:

```
typedef union
{
  struct _io_read  i;
} io_read_t;

struct _io_read
{ // contains at least the following
  unsigned short   type;   // message type = _IO_READ
  long             nbytes; // number of bytes to be read
  uint32_t         xtype;  // extended type
};
```

READ

For the reply:

- if successful:

  the reply message would be your data (i.e. there is no header to worry about)

  the return value from the read()'s MsgSend() would be the number of bytes successfully read. To set this, do:

```
_IO_SET_READ_NBYTES (ctp, nbytes_read);
SETIOV (ctp->iov, data, nbytes_read);
return (_RESMGR_NPARTS(1));
```

  When you return to the resource manager library, it will pass nbytes_read as the status parameter to MsgReplyv(). The read() will return this value.

  if failed, do:

```
return (errno_value);
```

READ

example's **read** function:

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

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

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

  _IO_SET_READ_NBYTES (ctp, 0); /* 0 bytes successfully
   read */

  if (msg->i.nbytes > 0) /* mark access time for update */
```

The message you'll receive is:

```
typedef union
{
  struct _io_write i;
} io_write_t;
/* the data to be written usually follows the
  io_write_t */

struct _io_write
{ // contains at least the following
  unsigned short   type;   // message type =
  _IO_WRITE
  long             nbytes; // number of bytes to
```

For the reply:

- if successful:
  - there is no data to reply with
  - the return value from the *write()*'s *MsgSendv()* would be the number of bytes successfully written. To set this do:

```
_IO_SET_WRITE_NBYTES (ctp, nbytes_written);
return (_RESMGR_NPARTS(0));
```

    When you return to the resource manager library, it will pass **`nbytes_written`** as the status parameter to *MsgReplyv()*. The *write()* will return this value.

- if failed, do:

```
return (errno_value);
```

WRITE

example's **write** function:

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

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

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

  // msg -> i.nbytes is the number of byte to be written,
  // we are telling it that we wrote everything (msg ->
   i.nbytes)
  _IO_SET_WRITE_NBYTES (ctp, msg -> i.nbytes);

  if (msg->i.nbytes > 0) /* mark times for update */
```

WRITE - Getting the data

example doesn't do anything with the data to be written, but what if you want to?

- The data usually follows the **`io_write_t`** in the message buffer.
- But it may not all have been received, what happens in the following case?

```
MsgSend(coid, smsg, sbytes, ...);
```

```
MsgReceive(chid, rmsg, rbytes, ...);
```

As you know, the kernel copies the lesser of the two sizes, so in this case only 1000 bytes will have been received. How do you handle this?

WRITE - Getting the Data

In your io_write callback:

```
int io_write (resmgr_context_t *ctp, io_write_t
   *msg, …)
```

You have several pieces of information:

- **`msg->i.nbytes`**
  - is the number of bytes passed to the client's *write()* call: **`write(fd, buf, nbytes)`**
- **`ctp->msg`**
  - is a pointer to the actual receive buffer
- **`ctp->msg_max_size`**
  - is the size of the receive buffer, **`ctp->msg`**
- **`ctp->info.msglen`**
  - is the number of bytes that have actually been copied into the receive buffer
  - includes the **`io_write_t`** header and any headers before it
- **`ctp->offset`**
  - is the size of any headers before the **`io_write_t`**
  - **`msg = (char *)(ctp->msg) + ctp->offset`**

WRITE - getting the data

We got all the write data:

`msg->i.nbytes <=`

WRITE - getting the data

We didn't get all the write data:

msg->i.nbytes >

WRITE - Getting the Data

How do you use this?

- if **`msg->i.nbytes`** is less than or equal to **`ctp->info.msglen - (ctp->offset + sizeof(io_write_t))`**
  - all the write data has already been received, and we can use it directly from the receive buffer
- otherwise, we don't have the whole message
  - need to find somewhere to put the data
  - need to go get the rest of the data using *resmgr_msgread()*

WRITE - Getting the data

A simple method is to reread all of it from the sender's buffer:

```
int
io_write (resmgr_context_t *ctp, io_write_t *msg, void
*ocb)
{
  char *buf;
  buf = malloc( msg->i.nbytes );
  ...
  resmgr_msgread (ctp, buf, msg -> i.nbytes,
                  sizeof (msg -> i));
  // do something with buf
  free( buf );
  ...
}
```

offset to skip,
in this case the header

WRITE - Getting the Data

But there are other choices, including:

- find available cache buffers and use *resmgr_msgreadv()* to fill them
- use a small buffer and multiple *resmgr_msgread()* calls to work through the client's message a piece at a time
- ensure in advance that the receive buffer will be large enough for your largest write
- copy the already received data from the receive buffer, and then use *resmgr_msgread()* for the rest

Topics:

Overview

A Simple Resource Manager

- Initialization
- → Handling *read()* and *write()*

Device-Specific and Per-Open Data

Out-of-Band/Control Messages

The different methods

- io_msg method
- devctl method

Conclusion

Time1 resource manager

To illustrate we'll look at the **`time1`** resource manager:

– it behaves as follows:

Resource Manager	Registered Name	Description
`time1`	**`/dev/time/now`**	"**`cat /dev/time/now`**" prints the current time
`time1`	**`/dev/time/min`**	"**`cat /dev/time/min`**" prints the minutes

Time1 resource manager

So our resource manager must:

- handle multiple devices
  - **`/dev/time/now`**
  - **`/dev/time/hour`**
  - **`/dev/time/min`**
- return some data on a read
  - return the data once
  - on subsequent reads, return zero bytes to indicate end of file
- maintain context between **open** and **read**

To register the new names, we call *resmgr_attach()* 3 times:

```
resmgr_attach (dpp, &rattr, "/dev/time/now",
  _FTYPE_ANY, 0, &connect_funcs, &io_funcs,
  &nowattr);

resmgr_attach (dpp, &rattr, "/dev/time/hour",
  _FTYPE_ANY, 0, &connect_funcs, &io_funcs,
  &hourattr);

resmgr_attach (dpp, &rattr, "/dev/time/min",
  _FTYPE_ANY, 0, &connect_funcs, &io_funcs,
  &minattr);
```

Time Resource Manager

Resource Manager Structures - ocb, att, mount

These structures work as follows:

1. Device Specific Information - attr

The device specific information or handle:

```
resmgr_attach (..., RESMGR_HANDLE_T *handle);
```

device structure
iofunc_attr_t

– is specific to the device (path) being attached

– the helper functions need an **`iofunc_attr_t`**. So where do we store our device specific data?

– we override the **`iofunc_attr_t`**, encapsulating it as part of our data.

1. Device Specific Information - attr

From the time resource manager example:

```
struct Timeattr_s;
#define IOFUNC_ATTR_T   struct Timeattr_s

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

typedef struct Timeattr_s {
  iofunc_attr_t attr;   // encapsulate iofunc layer's
                        // attr
  int           device; // our data
} Timeattr_t;
```

device structure
iofunc_attr_t

Why define **IOFUNC_ATTR_T**?

1. Device Specific Information - attr

Why define IOFUNC_ATTR_T?

To reprototype functions, e.g.:

device structure
iofunc_attr_t

```
int io_open (..., RESMGR_HANDLE_T *handle, ...)
```

From **`sys/iofunc.h`** we see that **`RESMGR_HANDLE_T`** is defined as follows:

```
#ifndef RESMGR_HANDLE_T
  #ifndef IOFUNC_ATTR_T
    #define RESMGR_HANDLE_T  iofunc_attr_t
  #else
    #define RESMGR_HANDLE_T  IOFUNC_ATTR_T
  #endif
#endif
```

So **`IOFUNC_ATTR_T`** and **`RESMGR_HANDLE_T`** are now the structure that we defined (**`Timeattr_t`** in this case.)

1. Device Specific Information - attr

Why do things in the library still work?

- because we put an **`iofunc_attr_t`** as the first member of our structure
  - a pointer to our structure is also a pointer to the standard structure, so
  - the library code will continue to work

```
typedef struct Timeattr_s {
  iofunc_attr_t attr;    // encapsulate iofunc layer's
                         // attr
  int           device;  // our data
} Timeattr_t;
```

device structure
iofunc_attr_t

The Time Resource Manager's attr - setting it up

From the time resource manager:

device structure
iofunc_attr_t

```
#define HNow          0
#define HHour         1
#define HMin          2
#define NumDevices    3
Timeattr_t    timeattrs [NumDevices];
```

... and in *main()* ...

```
for (i = 0; i < NumDevices; i++) {
  iofunc_attr_init (&timeattrs [i].attr, ...);
  timeattrs [i].attr.mount = &time_mount; // we'll see
   this later
  timeattrs [i].device = i;
  resmgr_attach (dpp, &rattr, devnames [i], _FTYPE_ANY, 0,
                 &connect_funcs, &io_funcs,
                 &timeattrs [i]);
```

The Time Resource Manager's attr - using the attr

The final effect on the prototype for our io_open function:

```
int io_open (..., RESMGR_HANDLE_T
 *handle, ...)
```

– since **RESMGR_HANDLE_T** in the time resource manager is now defined to be **Timeattr_t**, the io_open function could become:

```
int io_open (..., Timeattr_t
 *tattr, ...)
```

– But, as you'll see, we don't have an io_open

2. “per open” data - ocb

Next, the “per open” data:

- We already saw that the ocb is allocated on a “per open” basis.
- But it is defined as a **`RESMGR_OCB_T`** which, as we’ll see, is just a **`iofunc_ocb_t`**.
- So just like the **`iofunc_attr_t`**, we must override it but still encapsulate it.

2. “per open” data - ocb

The first step in overriding the iofunc layer’s ocb is to define IOFUNC_OCB_T:

```
struct Timeocb_s;
#define IOFUNC_OCB_T  struct Timeocb_s

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

typedef struct Timeocb_s {
  iofunc_ocb_t ocb;  // encapsulate iofunc layer’s
  ocb
  char     *buffer;   // for buffer of data we return
  int      bufsize;   // how many bytes are in buffer
```

2. “per open” data - ocb

ocb

Why define IOFUNC_OCB_T?

To reprototype functions, e.g.:

```
int io_read (..., RESMGR_OCB_T *ocb)
```

From **`sys/iofunc.h`** we see that **`RESMGR_OCB_T`** is defined as follows:

```
#ifndef RESMGR_OCB_T
  #ifndef IOFUNC_OCB_T
    #define RESMGR_OCB_T  iofunc_ocb_t
  #else
    #define RESMGR_OCB_T  IOFUNC_OCB_T
  #endif
#endif
```

So both **`IOFUNC_OCB_T`** and **`RESMGR_OCB_T`** are really the structure that we defined (**`Timeocb_t`** in our case.)

2. “per open” data - using the ocb

So now we've changed the prototype for our I/O functions (e.g. the io_read function):

```
int io_read (..., RESMGR_OCB_T *ocb)
```

– since **`RESMGR_OCB_T`** in our /dev/time example is now defined to be **`Timeocb_t`**, the io_read function would now become:

```
int io_read (..., Timeocb_t *tocb)
```

ocb

2. “per open” data - ocb - allocating and freeing

But the ocb is a dynamically allocated structure:

– so we must supply functions to allocate and free this structure:

```
IOFUNC_OCB_T *time_ocb_calloc (resmgr_context_t *ctp,
                               IOFUNC_ATTR_T *attr)
void time_ocb_free (IOFUNC_OCB_T *ocb)
```

time_ocb_calloc() is called by the default open function: *iofunc_open_default()*

time_ocb_free() is called by the default close_ocb function: *iofunc_close_ocb_default()*

The Time Resource Manager's ocb - allocating and freeing

First, we allocate some structures:

```
/*
 * we put the addresses of our ocb allocating and freeing
 * functions in the iofunc_funcs_t structure and then ...
*/
iofunc_funcs_t time_ocb_funcs = {
  _IOFUNC_NFUNCS,
  time_ocb_calloc,
  time_ocb_free
};

/*
 * ... put the address of the iofunc_funcs_t structure in
   the
 * mount structure
*/
```

mount structure

ocb_calloc()
ocb_free()

The Time Resource Manager's ocb - allocating and freeing

Then we put the address of the mount structure into the iofunc_attr_t structure:

```
for (i = 0; i < NumDevices; i++){
  iofunc_attr_init (&timeattrs [i].attr, ...);
  timeattrs [i].attr.mount = &time_mount;
  timeattrs [i].device = i;
  resmgr_attach (dpp, &rattr, devnames [i],
  _FTYPE_ANY, 0,
                &connect_funcs, &io_funcs,
                &timeattrs [i]);
}
```

The Time Resource Manager's ocb - allocating and freeing

And here are the functions:

```
Timeocb_t *
time_ocb_calloc (resmgr_context_t *ctp, Timeattr_t *tattr)
{
   Timeocb_t   *tocb;

   tocb = calloc (1, sizeof (Timeocb_t));
   // do anything else to the ocb
   tocb -> ocb . offset = 0;
   tocb -> buffer = NULL;
   return (tocb);
}

void
time_ocb_free (Timeocb_t *tocb)
{
   if (tocb -> buffer) {
        free (tocb -> buffer);
   }
   free (tocb);
}
```

Things our functions need to know - one last detail

One last detail - Here's some of iofunc_ocb_t:

```
typedef struct _iofunc_ocb {
  IOFUNC_ATTR_T  *attr;
  ...
} iofunc_ocb_t;
```

It includes a pointer to the **`IOFUNC_ATTR_T`**, which we overrode. The **`IOFUNC_ATTR_T`** encapsulates **`iofunc_attr_t`**.

So given the ocb ("per open" data) we can get at the attr (device specific data)!

So, for **`time1`** it looks like:

Time Resource Manager's io_read

The io_read routine:

```
int
io_read (resmgr_context_t *ctp, io_read_t *msg, Timeocb_t *tocb)
{
  int nleft, onbytes, device;   // nleft - number left to send,
                                // onbytes - number replying with
  ... // do usual xtype checking and iofunc_read_verify()

  // figure out which device we're handling
  device = tocb -> ocb . attr -> device;

  if (tocb -> buffer == NULL) { // this is the first time here!
    tocb -> buffer = formatTime (device, 0);
    tocb -> bufsize = strlen (tocb -> buffer) + 1; // +1 for NULL
  }
```

Time Resource Manager's io_read

io_read (conclusion):

```
// calculate bytes left to send, we need to do this because
// the buffer client supplies might be smaller

nleft = tocb -> bufsize - tocb -> ocb . offset;
onbytes = min (msg -> i.nbytes, nleft);

// ocb -> ocb . offset points to bytes that we HAVE NOT YET
// SENT! so, we have to do our own reply, and THEN update!

if (onbytes) {                          // returning anything?
  MsgReply (ctp -> rcvid, onbytes,
            tocb -> buffer + tocb -> ocb . offset, onbytes);
} else {                                // no, no data to return
  MsgReply (ctp -> rcvid, 0, NULL, 0);
}

tocb -> ocb . offset += onbytes;        // now go ahead and adjust

return (_RESMGR_NOREPLY);               // we already did it!
```

How does it work?

How does it all work?

- how does the library know which ocb allocation functions to use
- how does the library find the right ocb when an I/O message comes in?
- where does the ctp (the dispatch_context) fit in with the ocb & attribute structures?

Let's put together the structures we've seen so far...

Resource Manger Structures

Resource Manger Structures

Topics:

Overview

A Simple Resource Manager

- Initialization
- Handling *read()* and *write()*

Device-Specific and Per-Open Data

Out-of-Band/Control Messages

The different methods

→ io_msg method

- devctl method

Conclusion

Out-of-Band Messages – io_msg

io_msg:

- used for sending “out-of-band” (i.e. control) messages from a client to a resource manager, using the file descriptor (**fd**)
- format is specific to the resource manager
- client calls *MsgSend*()*
- resource manager handles it using an **io_msg** handler

Out-of-Band Messages – io_msg (Client)

On the client side:

The client formats a message using the following structure as a message header:

```
struct _io_msg {
uint16_t type;              // _IO_MSG
uint16_t combine_len;       // sizeof(struct _io_msg)
uint16_t mgrid;             // see below
uint16_t subtype;           // anything you want, probably a
                            // different value for each
                            // possible message
}
```

– **mgrid** is meant to identify your specific resource manager. Use a value >= **_IOMGR_PRIVATE_BASE** (from **sys/iomgr.h**).

– Any data should follow the above header.

Out-of-Band Messages – io_msg (Client)

So the client would do:

```
typedef struct {
  struct _io_msg hdr;
  char mydata;
} my_msg_t;

my_msg_t msg;
int fd;
fd = open("/dev/my_resmgr", O_RDWR);
msg.hdr.type = _IO_MSG;
msg.hdr.combine_len = sizeof(msg.hdr);
msg.hdr.mgrid = _IOMGR_IO_MSG;              /* you make this up */
msg.hdr.subtype = IO_MSG_SUBTYPE_XXX; /* your message
                                                 type */
msg.mydata = 'X'; /* your data */...
MsgSend(fd, &msg, sizeof(msg), &reply, sizeof(reply));
                                    // 'reply' is whatever you want
```

Out-of-Band Messages – io_msg (resmgr)

On the resource manager side:

– register a function to be called to handle the messages which the client will send (**_IO_MSG** type messages):

```
resmgr_connect_funcs_t connect_funcs;
resmgr_iofuncs_t io_funcs;

iofunc_func_init (_RESMGR_CONNECT_NFUNCS,
&connect_funcs,_RESMGR_IO_NFUNCS, &io_funcs);

io_funcs.read = io_read;
io_funcs.write = io_write;
io_funcs.msg = io_msg; // will be called when the
                       // client's _IO_MSG arrives
```

Out-of-Band Messages – io_msg (resmgr)

And then handle any messages:

```
int
io_msg (resmgr_context_t *ctp, io_msg_t *msg,
RESMGR_OCB_T *ocb)
{
        my_msg_t *mymsg;
        // do message length testing as in io_write if needed or
        // if message is small enough
        mymsg = (my_msg_t *) msg;
        // process the message
        ...
        // reply with whatever you want, there is nothing resource
        // manager specific to put in your reply
        MsgReply(ctp->rcvid, 0, &reply, sizeof(reply));
        return (_RESMGR_NOREPLY); }
```

Topics:

Overview

A Simple Resource Manager

- Initialization
- Handling *read()* and *write()*

Device-Specific and Per-Open Data

Out-of-Band/Control Messages

The different methods

- io_msg method
- devctl method

Conclusion

Out-of-Band Messages - devctl

Device control (**`devctl`**):

- used for sending “out-of-band” (i.e. control) messages from a client to a resource manager, using the file descriptor (**`fd`**)
- very much like the UNIX *ioctl()* call
- format is specific to the resource manager
- client calls *devctl (…)*
- resource manager handles it using an **`io_devctl`** handler

Out-of-Band Messages - devctl

These are used for:

- Sending data to the server: Client sends a command and data block and gets back an integer status

- Receiving data from the server: Client sends a command (integer) and gets back a buffer of data and a status

- Sending and Receiving data to/from: Client sends a command and data block and gets back a buffer of data and a status

Out-of-Band Messages - devctl

Prototype:

```
devctl (int          fd,                  // fd of device to
   control
        int    dcmd,                 // device command & direction
        void   *dev_data_ptr,        // device data (structure)
        size_t nbytes,               // number of bytes of data to
                                     // send/receive (maximum of the
   two)
        int    *dev_info_ptr);       // device return status
   (optional)
```

Sends and/or receives device specific information between the client and resource manager. **`dcmd`** contains the command code, direction and size, and **`dev_data_ptr`** points to the information.

Create device controls:

- define commands:

```
#define      DCMD_DEVTIME_SETFMT
#define      DCMD_DEVTIME_GETFMT
```

- define associated data

  we'll see the actual definitions on the next slide…

  - must be capable of being an operand of **`sizeof()`**
  - if required, define structures for the commands:
  - in our example, we pass a fixed length string, so a structure is not needed

The command definitions:

```
#define  MaxFormatSize     64

#define DEVTIME_SETFMT_CMD_CODE 1
#define DEVTIME_GETFMT_CMD_CODE 2

#define DCMD_DEVTIME_SETFMT __DIOT(_DCMD_MISC, DEVTIME_SETFMT_CMD_CODE,
                                   char [MaxFormatSize])
#define DCMD_DEVTIME_GETFMT __DIOF(_DCMD_MISC, DEVTIME_GETFMT_CMD_CODE,
                                   char [MaxFormatSize])
```

Class, 0x000 - 0xfff is reserved for QNX, **_DCMD_MISC** is for general use (see devctl.h)

Associated data, the macro will do a sizeof() on this.

Out-of-Band Messages - devctl

The previous slide showed macros for:

- sending data (To):

```
__DIOT(class, code, data_type)
```

- getting data (From):

```
__DIOF(class, code, data_type)
```

Other possibilities:

- To both send and get back data in one devctl:

```
__DIOTF(class, code, data_type)
```

  just keep in mind that the same buffer is used both for the sending data and for the data gotten back

- To just give a command:

```
__DION(class, code)
```

Out-of-Band Messages - devctl (client)

To use **devctl**:

```
int  fd;                        // file descriptor for the
   device
char        format [MaxFormatSize]; // for the format

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

to set the format:

```
strcpy (format, "%y%m%d");
devctl (fd, DCMD_DEVTIME_SETFMT, &format, strlen (format)+1,
   NULL);
```

clean up after use:

```
close (fd);
```

Out-of-Band Messages - devctl (resmgr)

On the server (resource manager) side:

- provide an **`io_devctl`** handler
  - add an entry into the I/O functions table
- call the default devctl handler
- decode commands
  - **`switch/case`** on the **`dcmd`** member
- perform actions
  - these vary depending upon the resource manager

Out-of-Band Messages - devctl (resmgr)

devctl messages:

```
struct _io_devctl { // contains at least the following:
   uint16_t          type;
   uint16_t          combine_len;
   int32_t           dcmd;      <---- Device control command
   int32_t           nbytes;
   int32_t           zero;      <---- Data (if any) follows zero
};                                     for nbytes bytes.
struct _io_devctl_reply {
   uint32_t          zero;
   int32_t           ret_val;
   int32_t           nbytes;
   int32_t           zero2;     <---- Return data (if any) follows
};                                     zero2 for nbytes bytes.
typedef union {
   struct    _io_devctl              i;
   struct    _io_devctl_reply        o;
} io_devctl_t;
```

Out-of-Band Messages - devctl (resmgr)

Let's add **`io_devctl`** to our time resource manager:

```
resmgr_connect_funcs_t connect_funcs;
resmgr_iofuncs_t io_funcs;

iofunc_func_init (_RESMGR_CONNECT_NFUNCS, &connect_funcs,
                  _RESMGR_IO_NFUNCS, &io_fincs);
io_funcs.read      = io_read;
io_funcs.write     = io_write;
io_funcs.devctl    = io_devctl;
```

time2's io_devctl

The io_devctl routine:

```
char   formats [3][];          // global, used by helper routines

int
io_devctl (resmgr_context_t *ctp, io_devctl_t *msg, Timeocb_t
   *tocb)
{
  void *dptr;                  // pointer to actual data area
  int  status, nbytes;

  status = iofunc_devctl_default(ctp, msg, &tocb->ocb);
  /* _RESMGR_DEFAULT flags an unknown devctl... that is,
      one we need to handle */
  if (status != _RESMGR_DEFAULT) return (status);

  nbytes = 0;                  // number of bytes we are returning
  dptr = _DEVCTL_DATA (msg -> i); // used for read/write
```

continued

time2's io_devctl (continued)

io_devctl (conclusion):

```
  switch (msg -> i.dcmd) {
  case DCMD_DEVTIME_SETFMT:
       tocb->ocb.attr->format = strdup((char *)dptr);
       break;
  case DCMD_DEVTIME_GETFMT:
       strcpy ((char *) dptr, tocb->ocb.attr->format);
       nbytes = strlen ((char *)dptr) + 1; // include NULL
       break;
  }

  if (nbytes == 0) {
    return (EOK);
  } else {
    msg -> o.ret_val = 0;
    msg -> o.nbytes = nbytes;
    return (_RESMGR_PTR (ctp, &msg -> o, sizeof (msg -> o) +
                         nbytes));
  }
}
```

Conclusion

You learned:

– that a resource manager is a device driver framework

– how to initialize and register a resource manager

– how to track per-open and per-device data, and extend the standard structures with your own data

– how to handle *read()*, *write()*, and *devctl()* client requests

Overview of QNX® Embedded Systems

AdvanTRAK Technologies Pvt Ltd

Introduction

You will learn:

- what happens at boot time
- what types of filesystems are available

- these are both things you will need to know before you can learn how to embed QNX

Topics:

→ **Booting**

Images & Buildfiles

Filesystems

Conclusion

Boot sequence:

- IPL code:
  - does chip selects and sets up RAM, then jumps to startup code
- startup code:
  - sets up some hardware and prepares environment for **`procnto`**
- **`procnto`**:
  - sets up kernel and runs boot script
- the boot script contains:
  - drivers and other processes, including yours

IPL and startup code:

– the IPL and startup code:
  - are board specific
  - are part of the Board Support Package (BSP)

OS image file

Much of this is in an OS image file:

IPL Overview

The IPL is responsible for the following:

- getting control either at reset time or after the BIOS/ROM monitor (as a BIOS extension)
- setting up the environment for the image:
  - configuring programmable chip selects,
  - initializing the DRAM refresh controller,
  - ...
- getting the image from whatever medium it is located on
- copying the startup code to RAM, if required
- transferring control to the startup code

IPL Overview

In general:

- if you do not have a BIOS or ROM monitor program that can boot from a device, then:
  - the IPL will have to do the steps on the previous slide
  - we call this a “Cold-start IPL”
- otherwise,
  - the IPL is a BIOS/ROM extension
  - an example of this type of IPL would be a BOOTP ROM
  - most (if not all) of the steps on the previous slide will already be done
  - we call this a “Warm-start IPL”

Cold-start IPL - Location of IPL: at reset vector

When the processor resets (no BIOS):

Warm-start IPL - Location of IPL: as BIOS extension

When the processor resets (with BIOS):

The Role of the IPL – Loading the OS Image

Role : Configure the hardware to create an environment that will allow the startup program, and then Neutrino microkernel, to run.

- **Start execution from the reset vector.**
- **Configure the memory controller, which may include configuring chip selects and/or PCI controller.**
- **Configure clocks.**
- **Set up a stack to allow the IPL lib to perform OS verification and setup (image download, scan, setup, and jump).**
- **IPL or the BIOS/ROM monitor code is responsible for transferring the image to linearly addressable memory**

Types of IPL : Warm-start and cold-start IPL

Warm-start IPL :

Invoked by a ROM-monitor or BIOS; some aspects of the hardware and processor configuration will have already been set up. If there's a BIOS or ROM monitor already installed at the reset vector, then your IPL code is simply an extension to the BIOS or ROM monitor.

Cold-start IPL:

Nothing has been configured or initialized -- the CPU and hardware have just been reset. When power is first applied to the processor (the processor is reset), some of its registers are set to a known state, and it begins executing from a known memory location (i.e. the *reset vector*).

IPL must be located at the reset vector and must be able to:

Set up the processor.

Locate the OS image.

Copy the startup program into RAM.

Transfer control to the startup program.

IPL Examples

ipl-diskpc1 :

Primary Boot Loader for x86 BIOS based machines. Eg. Boot Partition : 1,2,3,4

ipl-diskpc1-flop :

Primary Boot Loader for older x86 BIOS based machines. Eg. Boot Partition : 1,2,3,4

ipl-diskpc2 :

Secondary Boot Loader for x86 BIOS based machines. Eg. Hit Esc for .altboot…

ipl-diskpc2-flop :

Secondary Boot Loader for older x86 BIOS based machines. Eg. Hit Esc for .altboot…

Startup

The IPL's job was to:

- optionally handle the reset vector
- do the bare minimum required to:
  - create a friendly environment for startup to run in
  - get startup into memory (if necessary)
- transfer control to the startup code

Startup

The startup code's job is to:

- initialize hardware
  - MMU, timer hardware, interrupt controllers
- setup the system page
  - time of day
  - how much memory is installed
  - type of processor & coprocessor
  - hardware bus type
- provide kernel callouts
- setup a debugging device
- load and transfer control to the next program in the image (i.e. start QNX Neutrino)

The role of the startup program

- Configure the processor and hardware, detect system resources, and start the OS
- Program the base timers, interrupt controllers, cache controllers, and so on.
- Provide kernel callouts, which are code fragments that the kernel can call to perform hardware-specific functions
- Initializes the system and places the information about the system in the system page area (a dedicated piece of memory that the kernel will look at later),
- Responsible for transferring control to the Neutrino kernel and process manager (procnto), which perform the final loading step.
- Startup's responsibilities

  Copy and decompress the image, if necessary.

  Configure hardware.

  Determine system configuration.

  Start the kernel.

Some startups are:

`startup-bios`

- knows how to get the information from a standard, PC-compatible BIOS.
- works on most (if not all) desktop PC's and a large number of embedded systems.

`startup-vr41xx`

- startup for NEC Vr4102/Vr4111/VrC4171 unified evaluation board (MIPS).

`startup-800fads`

- preconfigured for Motorola 800 Application Development System board (PowerPC).

Custom Configuration

On custom hardware architectures, you may have to ‘roll your own’:

- Start with the source that most closely matches your hardware, beginning with the processor.

Definition: OS image file

- a file containing many of the software components needed for booting up
- looks something like this:

Ways of booting

You can boot from:

1. ROM/flash
2. hard disk or pseudo disk
3. network/serial

Let’s talk about these...

1. Booting from ROM/flash:

2. Booting from hard disk or pseudo disk:

- the image is loaded into RAM by various components - starting with the BIOS
- the last component jumps to the beginning of the image file

2. hard disk or pseudo disk

Looking closely at the image file:

Disk

The image can be loaded from a disk containing a QNX 4 filesystem:

- build image (using **`mkifs`**)

```
cp image /.boot
```

- can also be alternate image

```
cp image /.altboot
```

Pseudo-disk

Image stored on pseudo-disk (Flash):

– custom (or manufacturer supplied) tools required

3. Booting from network - ROM monitor/ TFTP:

Host running Windows, QNX Neutrino, ...

ROM monitor

ROM monitor on target (makes tftp request for `/xfer/broadcom.ifs`)

ROM monitor copies image to RAM and jumps to startup code

1. inetd monitors the ports and when request arrives on the TFTP port, it runs TFTP server
2. TFTP server delivers delivers image file over network to target

Network - bootp ROM

Image can be on any OS that has bootp:

1. bootpd server monitors port (or inetd monitors port and runs bootpd)
2. looks in /etc/bootptab for image to deliver
3. delivers image to boot ROM

physical address

```
/etc/bootptab:
 mcpn750:tc=defaults:ht=ethernet:ha=08.00.3b.26.01.7f:\
      :ip=10.30.30.29:sm=255.0.0.0:hd=/xfer:bf=mcpn750.img:bs
```

image

3. Booting from network - BOOTP/DHCP:

1. DHCP server monitors port
2. looks in **/etc/dhcpd.conf** for target's image filename
3. delivers image filename to bootp ROM
4. bootp ROM uses TFTP to get image file

```
/etc/dhcpd.conf:
 subnet 10.0.0.0 netmask 255.0.0.0 {
    ...
    host target {
       hardware ethernet 08:00:3b:26:01:7f;
       filename "target.ifs";
       ...
```

physical address

bootable image filename

3. Booting from serial:

Host running Windows,
QNX Neutrino, ...

ROM/flash with IPL

Serial port hardware on target

IPL code copies image to RAM and jumps to startup code

1. sendnto (or other file transfer process) monitors serial port
2. delivers image over serial port.

Example sendnto command line:

```
sendnto -d/dev/ser1 rpx-lite.ifs
```

bootable image filename

Building a Boot Image

Topics:

- Booting
- → Images & Buildfiles
- Filesystems
- Conclusion

What is an image?

What is an image?

- a file
- contains executables, and/or data files
- can be bootable

After boot, contents presented as a filesystem:

- **`/proc/boot`**
- simple
- read-only
- memory-based

What is an image?

Image:

The image tool chain

Images are created by the following:

What is in an image?

Bootable image components must include:

- **`startup-*`**
- **`procnto`** (kernel and Process Manager)

Image components may also include:

- drivers and managers, e.g.: **`io-net`**, **`devn-epic.so`**, **`devc-ser*`**, **`devb-eide`**
- **`esh`** (embedded shell), **`ksh`**
- and your applications & data files

Items with an asterisk following mean that there are multiple versions, depending on the hardware or platform.

What is a buildfile?

- specifies files / commands that will be included in the image,
- the startup order for executables,
- command line arguments and environment variables for executables,
- and loading options for files & executables.

let’s take a closer look...

“hello” example

A sample build file:

```
#   This is “hello.bld”
[virtual=x86,bios] .bootstrap = {
    startup-bios
    PATH=/proc/boot procnto
}
[+script] .script = {
    devc-ser8250 -e -b115200 &
    reopen /dev/ser1
    hello
}
[type=link] /usr/lib/ldqnx.so.2=/proc/boot/libc.so
libc.so
[data=c]
devc-ser8250
hello
```

To make an image from this, do:

```
mkifs hello.build hello.ifs
```

Buildfile format

General format of a buildfile:

```
attribute filename contents
attribute filename contents
…
```

Can include blank lines and comments as well (comments begin with the pound sign, "#")

All components are optional, but not all combinations are supported (see notes).

Attributes

Let's examine attributes:

There are two types of attributes:

- Boolean
  - **`[+attribute]`**
    - turns on the specified attribute (e.g. **`[+script]`**)
  - **`[-attribute]`**
    - turns off the specified attribute (e.g. **`[-optional]`**)
- Value
  - **`[attribute=`*value*`]`**
    - assigns a value to an attribute type (e.g. **`[uid=0]`**)

Attributes

When combining attributes, use this:

```
[attr1 attr2 …]
```

and not this:

```
[attr1] [attr2] …   # WRONG!
```

For example:

```
[uid=0 gid=0] file_owned_by_root
```

Attributes

Attributes can apply to single files:

– as in the following:

```
[uid=7] file1_owned_by_user7
[uid=6] file2_owned_by_user6
```

Or to all subsequent files:

– as in the following:

```
[uid=7]
file1_owned_by_user7
file2_owned_by_user7
```

In Line Files

The file can be given in line:

```
readme = {
   This is a handy way to get a file into the image
without actually having a file.  The file, readme, will be
accessible as /proc/boot/readme.
}
```

- Leading spaces count. The word "**`This`**" will be indented by 3 spaces as given above.
- To put a **`{`**, **`}`**, or a **`\`** character in the file, preceed them by a **`\`** (e.g. **`\{`**, **`\}`**, **`\\`**).

Scripts

Using the [+script] attribute, a file is treated as a script:

- It will be executed after the process manager has completed its startup.
- Multiple scripts will be concatenated into one and be interpreted in the order given.
- There are modifiers that can be placed before commands to run:

  Example: **`[pri=27f] esh`**
- There are also some builtin commands:

  Example: **`reopen /dev/con1`**

Example script:

```
[+script] .script = {
    display_msg Starting serial driver
    devc-ser8250 -e -b115200 &
    waitfor /dev/ser1 # don't continue until /dev/ser1 exists

    display_msg Starting pseudo-tty driver
    devc-pty &

    display_msg Setting up consoles
    devc-con &
    reopen /dev/con2  # set stdin, stdout and stderr to /dev/con2
    [+session pri=27r] PATH=/proc/boot esh &
    reopen /dev/con1  # set stdin, stdout and stderr to /dev/con1
    [+session pri=10r] PATH=/proc/boot esh &
}
```

Internal Commands

Internal commands are:

– ones that **`mkifs`** recognizes and are not loaded from the host's filesystem

- **`display_msg`** outputs the given text
- **`procmgr_symlink`** is the equivalent of **`ln -P`**, except that you don't have to have **`ln`** present
- **`reopen`** causes stdin, stdout, and stderr to be redirected to the given filename
- **`waitfor`** waits until a *stat()* on the given pathname succeeds

Examples of **`display_msg`**, **`reopen`** and **`waitfor`** can be found on the previous slide

Buildfile Contents

A buildfile for a bootable image ***must*** contain:

- bootstrap loader and operating system
- startup script
- executables and shared libraries
  - executables aren't strictly required, but then the system wouldn't actually ***do*** anything without them!

Everything else is optional.

"hello" example

We'll use the **`hello.bld`** example we saw earlier:

```
#   This is "hello.bld"
[virtual=x86,bios] .bootstrap = {
    startup-bios
    PATH=/proc/boot procnto
}
[+script] .script = {
    devc-ser8250 -e -b115200 &
    reopen /dev/ser1
    hello
}
[type=link] /usr/lib/ldqnx.so.2=/proc/boot/libc.so
libc.so
[data=c]
devc-ser8250
Hello
```

bootstrap file

startup script

shared library

executables

The Bootstrap File

The bootstrap file:

```
[virtual=x86,bios] .bootstrap = {
    startup-bios
    PATH=/proc/boot procnto
}
```

Contains:

- attribute

  ```
  [virtual=x86,bios]
  ```

- filename

  ```
  .bootstrap
  ```

- and contents

  ```
  startup-bios
  PATH=/proc/boot procnto
  ```

The Bootstrap File - Details

The "virtual" attribute refers to the virtual addressing model

Indicates that **bios.boot** contains information needed by **mkifs** to prepare the image for the IPL

Target processor

This name can actually be anything

```
[virtual=x86,bios] .bootstrap = {
    startup-bios
    PATH=/proc/boot procnto
}
```

As a result of this:

- **startup-bios** and **procnto** will be included in the image
- **startup-bios** will run first, it will then run **procnto**

startup-bios is an example of a startup program. This one is used on targets that have a BIOS. It initializes:

- hardware, the system page and kernel callouts
- runs the next program in the image, **procnto**

The Bootstrap File - Details

Other details:

- The target processor (e.g. **`[virtual=x86,bios]`**) is optional
  - this will be put in the **`$PROCESSOR`** environment variable
  - if not given then **`$PROCESSOR`** will default to the same as the host processor
- You can compress all of the image except the startup code using the **`+compress`** attribute

  Example: **`[virtual=x86,bios +compress]`**
  - the startup program will do the decompression at boot time

The Rest

The rest of the buildfile:

```
[type=link] /usr/lib/ldqnx.so.2=/proc/boot/libc.so
libc.so
[data=c]
devc-ser8250
hello
```

Contains:

- attributes

  `[type=link]` and `[data=c]`

- filenames

  `ldqnx.so.2`, `libc.so`, `devc-ser8250`, and `hello`

- and contents

  `/proc/boot/libc.so` (on the line with the attribute `[type=link]`)

Where files are found

To find files, mkifs, looks in:

${QNX_TARGET}/${PROCESSOR}/bin, **../usr/bin**, **../sbin**, **../usr/sbin**	binaries (**esh**, **ls**, etc)
${QNX_TARGET}/${PROCESSOR}/boot/sys	OSes (**procnto**, etc)
${QNX_TARGET}/${PROCESSOR}/lib	libraries and shared objects
${QNX_TARGET}/${PROCESSOR}/lib/dll	shared objects

- The above can be overridden:
  - using the **MKIFS_PATH** environment variable:

```
MKIFS_PATH=/usr/nto/x86/bin:
  /usr/nto/x86/sys:/usr/nto/x86/dll:
  /usr/nto/x86/lib:/project/bin
```

  - using the **search** attribute for a particular file:

```
[search=/projecta/bin:/projectb/bin] myexec
```

Where files end up

Once QNX Neutrino is up and running:

– the files will be in **`/proc/boot`**

So giving the following in a buildfile ...

```
devc-ser8250
/etc/passwd
```

... would result in:

```
/proc/boot/devc-ser8250
/proc/boot/passwd
```

- or they can be aliased to elsewhere:

So giving the following in a buildfile ...

```
devc-ser8250
/etc/passwd = /etc/passwd
```

... would result in:

```
/proc/boot/devc-ser8250
/etc/passwd
```

Including a whole whack of files

To include the contents of a directory, including subdirectories, you can do:

`/release1.0` = a directory

- everything under **`/release1.0`** will appear under **`/proc/boot`**

To have the contents appear in a different location:

`/product = /release1.0`

- everything under **`/release1.0`** will appear under **`/product`**

Sample Buildfiles

Sample buildfiles are in:

`${QNX_TARGET}/${PROCESSOR}/boot/build/`*board***`.build`**

Also see the documentation for the **`mkifs`** utility

Topics:

- Booting
- Images & Buildfiles
- → Filesystems
- Conclusion

Embedded filesystem requirements:

Embedded filesystems:

Image filesystem

- read-only
- free -- comes with the image itself
- no special access requirements

continued

Embedded filesystems (continued):

RAM disks

- read/write, non-persistent
- limited by size of free RAM
- choice of **`devb-ram`** or **`procnto`** memory objects
- **`devb-ram`**
  - full POSIX filesystem
- **`procnto`** memory objects
  - free -- implied by design of process manager
  - establishes files under **`/dev/shmem/`**
  - to use a different name, put this in your buildfile:

    `[type=link] /tmp=/dev/shmem`
  - not a POSIX filesystem - does not support mkdir

continued

Embedded filesystems (continued):

Flash filesystem

- read-only, read/write, read/write/reclaim; persistent
- our own media format, extent based like FFS 2.0
- PC Card (PCMCIA) support
- driver required (d**`devf-`*`type`***)

Rotating media filesystem

- read-only, read/write; persistent
- driver required (e.g. **`devb-eide`**)

continued

Embedded filesystems (continued):

Network filesystem

- filesystem resides somewhere else on a network
- NFS (**`fs-nfs*`**)
- CIFS, also known as SMB, (**`fs-cifs`**)
- Qnet™

Conclusion

You learned:

- the boot sequence and what components are needed at boot time
- some boot scenarios
- which filesystems are available

References

Building Embedded Systems (QSSL)

Utilities Reference (QSSL)

QNX Momentics Development & Debugging using IDE

Introduction

The IDE is a graphical environment:

- for developing and debugging applications
- for transparently accessing your target system
- for analysing the RTOS and your applications running on your target system
- is a highly integrated collection of tools
- is very configurable

Introduction

In this course you'll learn to use the IDE to:

- develop code
  - creating projects, compiling, accessing your target, running code
- debug
  - using conventional debugger and other IDE tools
- analyse performance issues
  - both at a process level and a system level
- do some embedding
- and more...
- even though we are taking a "how to" approach, along the way you'll also have learned a large amount of the IDE's individual features

Agenda

Topics:

- **IDE Basics**
  - Central IDE Concepts
  - Projects
  - Your Workspace
  - Accessing your Target
  - Preferences

Agenda

Managing C/C++ Projects

- Overview
- Basic Project Management
- Standard Make C/C++ Projects
- QNX C/C++ Application Projects
- QNX C/C++ Library Projects
- Getting Code from Elsewhere

Agenda

Topics:

Compiling and Running

- Compiling
- Fixing Errors
- Running

Basic Debugging

- Overview
- Setup
- Starting a Debugging Session
- Overview of the Debug Perspective
- Debugging Techniques

IDE Basics

You will learn:

- the basic layout of the IDE and how to:
  - rearrange it to meet your needs
  - get around
- the structure of your information in the IDE
- basics behind working in a host/target environment
- how to set your preferences

Topics:

- → Central IDE Concepts
- Projects
- Your Workspace
- Accessing your Target
- Preferences
- Exercise
- Conclusion

Workbench and IDE:

- are both terms referring to the the same thing -- the desktop development environment
- both terms are used and we'll use them interchangeably

Central IDE Concepts - Workbench window

The Workbench window:

- contains most of the components of the development environment

- the help window is one component that appears outside the Workbench window

Central IDE Concepts - Workbench window

Orientation:

Central IDE Concepts - Editors

An Editor is:

- a component of the IDE where you edit or browse a resource (such as a C source file)

The Editor area is:

- the section of the Workbench window reserved for editors

Central IDE Concepts - Views

A View is:

- an area that provides:
  - navigation
  - information
  - control

Perspectives

- the contents and layout of the Workbench window can vary
- you configure this using perspectives

- a Perspective is:
  - a collection of views, editors, menu items, and tool bar buttons that are helpful for doing a specific task

Central IDE Concepts - Perspectives

e.g. C/C++ Development perspective has:

– things for writing, building and running C/C++ code

To open a perspective:

click on the Open a Perspective button and choose from the list. Choose Other... if the one you want isn't there

OR

go to the Window menu and choose Open Perspective

To switch between perspectives and to close them:

open perspectives are listed in the top-right corner

if there isn't space for all, this button will complete the list

to switch between them click on or select the perspective name

to close a perspective, right-click on the name or icon for the one you want to close and choose Close.

once you are familiar with the icons, you can save space by unsetting Show Text

Some more perspectives include:

- CVS Repository Exploring
  - version/source control using CVS
- Debug
  - source level debugging
- QNX Application Profiler
  - process level profiling
- QNX Memory Analysis
  - stack, memory object, malloc debugging, heap analysis
- QNX System Builder
  - OS and embedded image creation, embedding tools
- QNX System Profiler
  - system level profiling

You can customize existing perspectives

– why might you do this?

- when working with C/C++ code it's handy to have the File System Navigator view shown. This is useful for dragging and dropping newly built executables to your target system

To show a particular view:

1. go to the Window menu…

2. … and choose Show View

3. pick a view. If the view you want is not avaiable, choose Other… for a complete list of all the views available...

The Show View dialog:

use this to filter the views that are listed to find the view you want

expand the view category and choose the view you want

Five ways to arrange a view:

1. tabbed

   tabbed with other views or by itself

to bring a view to the front, click on its tab

2. as a fast view

   one that you can quickly open and then quickly close again

3. detached

   in its own window

4. maximized

   takes the entire Workbench window

5. minimized

   reduced to just its title bar

To tab a view with others...

1 click on the titlebar of the view you want to tab with others

or right-click then select move -> view if you don't have drag & drop

2 drag the view (represented by a rectangle) to on top of the views you want to tab it with. When the mouse looks like drop it.

3 the view is now tabbed with the views you dropped it on top of

Docking views:

- you can also position views by docking them to the edge of something

Example: we want to move the Outline view so that it is where the left side of these tabbed views (Variables, …) are now

continued

Docking views (continued):

1 click on the titlebar of the Outline view

2 drag it until the mouse is pointing to the left edge of the tabbed views and the mouse looks like this, a pointer to the left edge, and drop it

3 the end result is that the Outline view is now where the left edge of the Variables (and other tabbed views) was. Notice that they are each taking up half the space

– we can do same thing for the other three edges

Resizing views:

1. move your mouse to between the views until it looks like this and press the mouse button down

2. drag the mouse until the the vertical line is positioned where you want the split between views to be

3. release the mouse button

Maximized and Minimized views:

- to maximize a view:
  - double-click the title bar or
  - click the maximize button
- to restore:
  - double-click the title bar or
  - click the restore button
- to minimize a view:
  - click the minimize button
- to restore:
  - click the restore button

Some useful perspective options:

Customize Perspective allows you to configure what appears in the menus and toolbar when the current perspective is visible

Reset Perspective restores the current perspective to it's default arrangement.

You can create your own perspectives

- why might you do this?
  - you might want to create a new perspective that contains all the various views that let you do things with your target system: Target Navigator view for a list of running processes, Terminal view for a dumb terminal, File System Navigator view for accessing the files on your target, … Maybe call it the Target perspective?
- to do this, rearrange an existing perspective, then…

go the the Window menu and choose Save Perspective As… Use this option to save the resulting perspective with a name of your choosing

Topics:

- Central IDE Concepts
- → Projects
- Your Workspace
- Accessing your Target
- Preferences
- Exercise
- Conclusion

Projects

Projects are:

- the basic containers in the IDE for:
  - C/C++ code
  - target information
  - OS and Embedded images
  - PhAB applications
  - and other things
- directories/folders

Projects

Some types of projects are:

- Standard Make C Project
- Standard Make C++ Project
- QNX C Project
- QNX C++ Project
- QNX Target System Project

Examples of use:

- if doing C and you have your own non-trivial source tree, create a Standard Make C Project
- if doing C++ code and you have multiple targets and no existing build structure, create a QNX C++ Project
- to access a target from the IDE, create a QNX Target System Project

Projects - Creating a project

For example, to create a QNX C++ project:

① go to the New menu

② And choose QNX C++ Project..

The end result:

the resulting project

Projects – Importing Project(s)

Another convenient way of creating projects is to import existing ones:

- File -> Import…

Existing Projects…

Archive file or Directory…

Then select the projects to import…

Projects – C/C++ Projects view

The C/C++ Projects view can filter what it shows you:

- show projects of all types
- show only open projects
- show only C/C++ projects
- show only a specific list of projects by defining a Working Set

Topics:

- Central IDE Concepts
- Projects
- → Your Workspace
- Accessing your Target
- Preferences
- Exercise
- Conclusion

Your Workspace

The workspace is:

- the directory/folder that contains your projects or pointers to your projects

Multiple workspaces:

- you can work with only one workspace at a time
- but you can select which workspace to use when you launch the IDE:

- or from the File menu:

- if the workspace doesn't exist, it will be created

Topics:

- Central IDE Concepts
- Projects
- Your Workspace
- → Accessing your Target
- Preferences
- Exercise
- Conclusion

Self-hosted vs Remote development:

Self-hosted:

PC running QNX Neutrino

Remote:

Host running Windows/Neutrino/Linux

Target running QNX Neutrino

Accessing your Target - qconn

qconn is:

- a process that must be running on your target
- what the IDE talks to for getting information and for controlling things on your target
  - even in the self-hosted case, **`qconn`** is needed

Accessing your Target - Target Navigator view

The Target Navigator view:

- from it you can:
  - connect to your target (create a Target System project)
  - see a list of all the processes running on your target
  - terminate processes
  - launch a telnet session
  - …
- but first you must create a Target System project...

A Target System project:

- contains configuration information for connecting to a target
- you must create one before the IDE can talk to your target
- there are a few places where you can create Target System projects:
  - Target Navigator view
  - Target File System Navigator view
  - Launch configurations window

Creating a Target System project:

from the Target Navigator view right-click in here and choose New QNX Target…

fill in a name representing your target. This will be the Target System project's name

fill in your target's IP address. `qconn` uses port 8000 by default so you wouldn't normally change this

You should now be connected to your target

click here to control sort order, refresh, ...

your new Target System

click here to expand this and get a list of the processes running on your target

process name process id (pid)

- if it fails to connect, check the IP address, check if qconn is running, check cables

Accessing your Target - Access filesystems on the target

To access files on the target:

- use the File System Navigator view

- you can drag and drop file(s) to/from the C/C++ Projects view
  - this will copy the files
- you can also drag and drop from external tools
- double-click on an executable to run it

Accessing your Target - Getting a command line

Some ways to get a command line on your target:

1. launch a telnet session
2. use a serial terminal
3. run a shell with console from the target File System Navigator view

1. Launch a telnet session:

① from the Target Navigator view right-click here for this menu

② choose Launch Telnet session

③ telnet should now be running and you should have some type of prompt

☞ the target must be configured correctly for telnet

2. Use a serial terminal:

– use one of the views named Terminal 1 to Terminal 4:

click here for a menu of configuration options

click here to send files

Accessing your Target - Getting a command line

3. Run a shell:

Target File System Navigator

my_vmware_target
- bin
- boot
- dag
- dev
- etc

File name
- gunzip
- gzip
- hostn
- kill
- ksh
- link

New
Copy to
Delete
Copy
Cut
Properties...
Run...
Edit

② right-click and choose Run… or double-click

① browse to a shell in the target File System Navigator view

③ Enable Create Console in the Advanced settings

Launch application on target

Enter application parameters

Extra environment variables

Name	Value

Add
Delete

☑ Create console
☑ Use pseudo terminal on target

Regular
OK
Cancel

The System Information perspective:

- has a lot of views with information about your target:
  - System Summary (host name, system memory, …)
  - Process Information (command line arguments, environment variables, threads and thread information, …)
  - Memory Information (stack sizes, code/data, heap, …)
  - Malloc Information (heap usage, history, number of mallocs/frees of certain sizes, …)
- some other views not shown by default:
  - Connection Information (file descriptor and side channel connection information)
  - Signal Information
  - System Resources (resource utilization)

Topics:

- Central IDE Concepts
- Projects
- Your Workspace
- Accessing your Target
- → Preferences
- Exercise
- Conclusion

Preferences

To set global preferences:

go to the Window menu
and choose Preferences

- the Preferences window appears with many, many, many options

Topics:

- Central IDE Concepts
- Projects
- Your Workspace
- Accessing your Target
- Preferences
- → Exercise
- Conclusion

IDE Basics exercise(optional):

- create a new perspective containing many useful components for accessing your target
- start with the Resource perspective since it's fairly empty
- add at least the following views:
  - Navigator view
  - Target Navigator view
  - File System Navigator view
  - a Terminal view

  and arrange them to suit your needs
- save your perspective as the Target perspective

Topics:

- Central IDE Concepts
- Projects
- Your Workspace
- Accessing your Target
- Preferences
- Exercise
- → Conclusion

Conclusion

You learned:

- how to get around in the IDE and arrange things to make working more efficient
- the IDE concept of a project
- the basic structure of your workspace so that you can manage it well. You even learned how to have more than one!
- the many ways you can interface with your target and make accessing it seamless
- where to set preferences

Managing C/C++ Projects

Introduction

You will learn:

- how to create projects for your C/C++ code
- general management of your projects, such as importing existing code

Topics:

→ Overview

Standard Make C/C++ Projects

QNX C/C++ Application Projects

Getting Code from Elsewhere

Exercise

Conclusion

Overview

A C/C++ Project is:

- a container for C/C++ code
  - there are projects types, but we won't talk about here
- before you can write C/C++ code in the QNX® IDE
  - you must create a project for code to go into

Building using “make” and “Makefiles”:

- the **`make`** command-line tool is used by the IDE to manage building executables and libraries
- **`make`** takes a Makefile that:
  - may contain commands for building that source tree (e.g. invoke complier and linker)
  - describes dependencies in a source tree
    - e.g. xxx.o would need to be rebuilt if xxx.c and xxx.h change
    - this is done using the modification date and time
    - e.g. if date and time of xxx.c or xxx.h is newer than xxx.o then build a new xxx.o
- for more information about Makefiles see **`make`** in the Utilities Reference

Overview - Types of projects

The projects for C or C++ code are:

- Standard Make C
- Standard Make C++
- QNX C Application
- QNX C++ Application
- QNX C Library
- QNX C++ Library

Which one do you use? ...

Overview - Selecting the type of project to use

Standard Make projects vs. QNX projects:

- Standard Make C/C++ Projects:
  - more flexible porting
  - easier to import existing, non-trivial, source trees
  - ability to use custom build tools or a tool other than **`make`**
  - the usual choice
- QNX C/C++ projects:
  - easily supports more than one target processor, e.g. developing for PPC *and* x86
  - you don't need to write a Makefile
  - QNX-designed Makefile system recognizes source differently depending upon it's 'level' within the directory structure

Basic Project Management - The two project views

You develop C/C++ code from:

- the C/C++ Development perspective
- this perspective has two views for accessing projects:
  - C/C++ Projects view
    - filters and formats the project information in a form suitable for working with code
    - you should develop from this view
  - Navigator view
    - shows all types of projects, including ones you don't care about when developing C/C++ code
    - has the generic project management features e.g. open/close projects, import/export)
- each has a unique right-click context menu
- some features are common to both views

Topics:

- Overview
- → Standard Make C/C++ Projects
- QNX C/C++ Application Projects
- Getting Code from Elsewhere
- Exercise
- Conclusion

A Standard Make C/C++ Project:

- is an empty project into which you can put whatever you want
- is a generic project, not QNX specific
- by default when you build your project it will run the following command:

```
make -k all
```

  but as you'll see this can be changed to be any build command line you want
- this type of project is useful if
  - you have an existing build structure and/or tools and want to keep using them
  - you want to be able to build your code without using the IDE

Projects - Creating a project

To create a project:

- if for C/C++ code, open the C/C++ Development perspective

① go to the New menu

② choose the project type you want

- the Project Creation wizard should appear

continued

Projects - Creating a project

Or, to create the project:

1 click on the New button

2 then select the type of project to create

Name it and you're done:

default location is inside workspace directory, un-checking this allows you to put it anywhere on your file-system

Standard Make C/C++ Projects - Creating the project

There are more settings:

but usually they can be left at default

Standard Make C/C++ Projects:

- are generic, not QNX specific
- for a variety of things to work well, or better, you need to label them as "QNX" source:

- this tells the IDE:
  - where to find header files
  - how to parse code for code completion, etc
  - environment to pass to make

Next, create a Makefile:

- the newly created project
- if you expand it, you'll see that it's empty
- now you have to fill it in
  - if you have an existing source tree, then copy it into the project, otherwise you'll need to create files, starting with a Makefile...

select your project,
right-click,
choose New->File

  - follow the steps in the Wizard to create the file

Minimal Makefile for C Project:

– a good minimal Makefile can contain:

this assumes you'll be creating a source file called **`hello.c`**

Minimal Makefile for C++ Project:

– a good minimal Makefile can contain:

this assumes you'll be creating a source file called **`hello.cc`**

– for more information about Makefiles see **`make`** in the Utilities Reference

Building your executable:

– to build, right-click on your project

only build items for which source code has changed (e.g. **`make -k all`**)

clean removes: executables, object code, error files, etc (e.g. **`make -k`**)

Working with your own **make** targets:

– in this example we look at a more complex example

we have a target called **dlk_consumer** in our Makefile which will build only that executable

Creating a **make** target (continued):

- the new target will be displayed to you with using the Target Name you choose
- the actual behaviour will be driven by the Make Target you supply
- this example when built will do the equivalent of:
  `make -k dlk_consumer`

To build the **`make`** target you created:

– select the **`make`** target, then click Build

Working with your own **`make`** targets:

- you can also add, delete, and build with targets in the 'Make Targets' view
  - the Make Targets view is on the right side, tabbed with the 'Outline' view
- quickest way is to just double-click a **`make`** target
- or you can use the menu:

Changing the build command line:

– right-click on an existing project, choose Properties

① select C/C++ Make Project ② turn off ③ Fill in your build command

Effect of changing your build command:

- you can still use the IDE to do your builds, it will automatically use your specified build command
- to build using a custom command, right-click on your project...

as we've seen, you can add your own make targets, in which case it does:

your_command your_target

Build - this will do:

your_command `all`

Clean - this will do:

Building your executable - outside the IDE:

- your project is basically a directory containing your files
- you can build your project from a command line

Command Prompt

```
C:\QNXsdk\workspace\hellostandard>make
qcc      hello.c   -o hello

C:\QNXsdk\workspace\hellostandard>
```

The IDE has knowledge of a project's contents:

- e.g. if you do...

  `make clean`

  … from the command line to remove executables, the IDE won't know you did that and will still display their filenames

- to fix this, tell the IDE to refresh it's information

right-click and choose 'Refresh'

Your built project looks like:

- the **`Binaries`** folder doesn't actually exist
  - It's there to show you all of the executables from this project in one place.
  - If you have many executables, or a complex directory structure, then this may be helpful

Your built project looks like:

- the **`Binaries`** folder doesn't actually exist
  - It's there to show you all of the executables from this project in one place.
  - If you have many executables, or a complex directory structure, then this may be helpful

Topics:

- Overview
- Standard Make C/C++ Projects
- → QNX C/C++ Application Projects
- Getting Code from Elsewhere
- Exercise
- Conclusion

A QNX C/C++ Application Project:

- is a project:
  - with a pre-designed, QNX supplied Makefile structure
  - that supports multiple variants/CPUs/targets/ platforms

The logical structure:

QNX C/C++ Application Projects - Example

A project in the IDE:

MIPS-Microprocessor without Interlocked Pipeline Stages

pseudo-folder with all executables for your convenience

MIPS build variants

x86 build variant

source

MIPS big-endian release version of the executable

MIPS big-endian debug version of the executable

MIPS little-endian variants

we'll talk more about `common.mk` and

QNX C/C++ Application Projects - Structure details

For details about the project structure:

- more details about this structure can be found in the Conventions for Makefiles and Directories appendix in the Programmer's Guide

QNX C/C++ Application Projects - Creating the project

To create the project:

- start as we saw for a standard Make project
- or the New C/C++ Project menu:

To create the project (continued):

① give the project a name

② choose 'Application', as as opposed to various types of libraries

③ click 'Next'

To create the project (continued):

1. from the Build Variants tab, select whatever build variants you want, including debug and/or release versions

2. click 'Finish' and your project is created

Working with Makefiles:

- don't edit the Makefiles
- the Makefiles include **common.mk**
- you can edit **common.mk**, but first check if there is an option in the project properties

Build related properties:

changes CPU architecture(s) and release and/or debug versions

Compiler options, e.g.
- location of source files outside the project directory
- location of include files outside the default locations
- optimization level, warning level

Linker options, e.g. extra libraries

these choices affect what gets built

Build related properties - Compile options

Setting compile options:

Options | Build Variants | General | Compiler | Linker | Make Builder | Error Parsers

Category: General options

Compiler type: Default (GNU Compiler Collection (2.95.3))

Output options

Warning level: Default

Preprocessor output

Code generation

Optimization level: Default

Stack size: 256K

Definitions

Other options: -gdwarf-2

Compilation options: -N256K -gdwarf-2

Build related properties - Adding a link library

To link in the math library:

– from the linker pane

set the Category to Extra libs

click on Add

And enter the library name: just put in the middle part (e.g. for `libm.a`, put `m`)

If you have header file dependencies:

- if we have A.h that includes B.h, and B.h is changed…
- **`make`** may not know that every .c file that includes A.h needs to be recompiled
- you can turn on dependency checking (two ways of doing this)

• right-click on project:

OR

• in project properties:

Building your executable - from the IDE:

– to build from the IDE, right-click on your project...

to build a user-created make target
`make -k` *your_target*

this will build only what needs to be built
`make -k all`

this will remove all files that are not source (e.g. executables, object modules, error files)
`make -k clean`

Building your executable - outside the IDE:

- your project is basically a directory containing your files
- you can build your project from a command line

Command Prompt

```
C:\QNXsdk\workspace\helloapp>make
make -j 1 -Cmips -fMakefile  && make -j 1 -Cx86 -fMakefile
C:\QNXsdk\host\win32\x86\usr\bin\make.exe[1]: Entering directory
space/helloapp/mips'
C:/QNXsdk/host/win32/x86/usr/bin/make.exe -j 1 -Co-le -fMakefile
ost/win32/x86/usr/bin/make.exe -j 1 -Co-le-g -fMakefile
C:\QNXsdk\host\win32\x86\usr\bin\make.exe[2]: Entering directory
space/helloapp/mips/o-le'
C:/QNXsdk/host/win32/x86/usr/bin/qcc -Vgcc_ntomips -c -Wc,-Wall -
eses -Wc,-fno-builtin -Wc,-fno-inline-functions -O -DNDEBUG
/QNXsdk/workspace/helloapp/mips/o-le -IC:/QNXsdk/workspace/helloa
Xsdk/workspace/helloapp -IC:/QNXsdk/target/qnx6/usr/include    -E
e  C:/QNXsdk/workspace/helloapp/helloapp.c
```

- but be careful, some settings can break external builds

Topics:

- Overview
- Standard Make C/C++ Projects
- QNX C/C++ Application Projects
- → Getting Code from Elsewhere
- Exercise
- Conclusion

Getting Code from Elsewhere

There are several ways to bring code into the IDE:

- source control systems
- IDE Import Wizard
- using code that is outside of the workspace
- drag and drop/external file manager type tools

Let's talk about these...

Getting Code from Elsewhere - Source control

Source control: *Concurrent Versions System* (*CVS*)

- IDE uses CVS by default, but plug-ins for other systems are available
- for CVS:
  - use the CVS Repository Exploring perspective
  - in the C/C++ Projects view (from the C/C++ Development perspective), right-click on a project and look at the Team submenu
- see the Eclipse documentation's Workbench User Guide for more on this (look for "team" related topics)

Getting Code from Elsewhere – Import Wizard

The Import Wizard:

- code can come from files/directories and go into an existing project

  or,

- can come from projects, for use by/in an existing workspace
  - often would come from someone using the export feature

- code can be:
  - copied into your workspace,

    or,

  - used in place
    - the IDE project will contain a pointer to the location of the code

Getting Code from Elsewhere – Import Wizard

To use the Import Wizard:

- right-click in either C/C++ Projects or Navigator view

- you'll be given a number of options, we'll discuss a few common ones:

Getting Code from Elsewhere – Import Wizard

To import files and put them into a project (files→project):

- if they are in an archive (e.g. zip) then use:

- otherwise, use:

- in both cases you must have already created the project where you want the files to go

Getting Code from Elsewhere – Import Wizard

To import projects into your workspace (projects→workspace), use:

–you can select the project(s) from your file-system, or from an archive

–you can choose to have the project code be copied into your workspace, or use the project(s) in place, wherever it(they) currently exist(s)

Getting Code From Elsewhere – QNX source

To use source supplied by QNX Software Systems:

- download source installer from **`www.myqnx.com`**
- install source package
- from the File menu, select Import…
- and choose QNX Source Package

Getting Code from Elsewhere – Projects outside the workspace

A project doesn't have to be in the workspace directory:

- can be
  - on a file server
  - in a directory created by an external source control tool
  - anywhere you want
- only one active IDE should ever point to the project
- will create `.project` and `.cdtproject` files in the selected location

External Projects - Creating

To point a project outside the workspace:

– unselect “Use Default”

– then browse to external location

External Projects - Caveat

Some things to be careful of:

- unlike most of the other techniques in this section, this one does not copy the source into the IDE/workspace
- on project deletion:

delete contents will delete all the external files

Projects are just directories/folders:

1. from outside the IDE, use any file management tool to copy files into your project directory/folder
2. in the C/C++ Projects view, right-click on your project and choose **Refresh** to tell the IDE that there are new files in your project

Getting Code from Elsewhere - Drag and Drop

On some hosts you can:

- drag from an external file manager and drop into the C/C++ Projects or Navigator views
  - e.g. from the Windows Explorer

Topics:

- Overview
- Standard Make C/C++ Projects
- QNX C/C++ Application Projects
- Getting Code from Elsewhere
- → Exercise
- Conclusion

Exercise

Creating a Standard Make C/C++ Project exercise:

- create either a Standard Make C Project or a Standard Make C++ Project. Call it **`standard`**
- in it you'll need to add at least two new files:
  - a source file with some code in it. This should contain *main()*
  - a Makefile - see the slides that we covered for some idea of what to put in here
- build your project to make sure everything is done right

Topics:

- Overview
- Standard Make C/C++ Projects
- QNX C/C++ Application Projects
- Getting Code from Elsewhere
- Exercise
- → Conclusion

Conclusion

You learned how to:

- create C/C++ projects
  - Standard Make C/C++ Projects
  - QNX C/C++ Application Projects
- do general management of your projects
- bring source code into the IDE

Editing and Compiling

Introduction

You will learn:

- some nifty features of the C/C++ editor
- the various ways of building your code
- how to use the IDE to find your errors

Topics:

- **Using the C/C++ Editor**
  - Writing Code
  - Navigating Code
  - Oops, What Have I Done?
- **Compiling**
- **Fixing Errors**
- **Exercise**
- **Conclusion**

Standard Make C/C++ Projects – Tell the IDE it's a QNX Project

For much of the following stuff to work for Standard Make C/C++ Projects:

- you need to label them as "QNX" source by doing a "Set QNX Build Environment…"

Topics:

- **Using the C/C++ Editor**
  - **Writing Code**
  - **Navigating Code**
  - **Oops, What Have I Done?**
- **Compiling**
- **Fixing Errors**
- **Exercise**
- **Conclusion**

Editing Code - Preferences

The appearance and behavior of the editor is highly configurable:

- start with Windows→Preferences
- the main place to look is:

  C/C++→Editor
- fonts are changed through:

  Workbench→Colors and Fonts

Writing Code – Code completion

The IDE does code completion:

- three types:
  - function completion
  - code blocks
  - structure member completion
- triggered by Ctrl-Space
- structure member completion can also be triggered by waiting
  - default is ½ second wait

Code Completion – Function completion

Function completion:

- will list functions that match
  - updates/shrinks list as you type
  - Enter selects

- then prompts you for function parameters:

Code block completion:

- will list blocks that match
  - shows what the result will be:

- replaces placeholders with the variable you give:

```
for (int count = 0; count < max; ++count) {

}
```

- use Tab to move through the elements:

```
for (int count = 0; count < 50; ++count) {

}
```

Structure completion:

- will complete structures or unions:
  - gives element names and types

- can be configured to trigger automatically on delay:
  - set by Window→Preferences→C/C++→Editor→Content Assist

Writing Code – Hover help

The IDE does hover help for:

– library functions:

```
printf ("%s\n", filenames[i]);
```

Name: printf
Protoype: int printf(const char * format, ...)
Description:
Write formatted output to stdout

```
#include <stdio.h>
int printf( const char * format, ... );
```

– your functions:

```
display_filenames (filenames, nfilenames);
```

```
void display_filenames (char **filenames, int nfilenames);
```

– global variables:

```
dirname, &filenames);
```

```
char *dirname = "/proc/boot";
```

Writing Code – Header files

The IDE can include system headers for you:

- click or select the function you want:

```
1 int
2 main(int argc, char **argv) {
3     int fd;
4
5     fd = open("/tmp/outfile", O_RDWR|O_CREAT, 0 666 );
6 }
```

- then right-click and Add Include:

Content Assist	Ctrl+Space
Add Include	Ctrl+Shift+N
Format	Ctrl+Shift+F

- and any needed headers are added:

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <fcntl.h>
4 int
5 main(int argc, char **argv) {
6     int fd;
7
8     fd = open("/tmp/outfile", O_RDWR|O_CREAT, 0666);
```

Writing Code – Block editing

You can do some basic block editing:

- select a block:

- you can:
  - comment (C++ style)
  - uncomment
  - tab right or
  - tab left

the whole block

Topics:

- **Using the C/C++ Editor**
  - Writing Code
  - → Navigating Code
  - Oops, What Have I Done?
- **Compiling**
- **Fixing Errors**
- **Exercise**
- **Conclusion**

Navigating Code – Working Sets

A working set:

- is a sub-set of the resources the IDE knows about
  - is usually a collection of related projects
    - e.g. an application and two libraries it uses
- let's you:
  - reduce or filter what is shown in your C/C++ Projects view
  - reduce the domain of your searches, giving:
    - more relevant results
    - faster searches

Working Sets – Creating a working set

To create one, in the C/C++ Projects view:

1

2

3

4

Working Sets – C/C++ Projects view

After selecting the working set, your C/C++ Projects view will only show the contents:

Navigating Code – Finding and searching

The IDE provides both find and search:

- find will match (or match and replace) strings in the current editor:
  - hot key is Ctrl-F
  - only the first result is found
    - can go to next
- search will examine multiple resources (files):
  - hot key is Ctrl-H
  - will return multiple matches (if they exist)
  - search domain can be restricted to a particular working set
  - displays results in the Search view

Finding and Searching – File Search

File search:

- gives a simple string search of the resources you choose

C/C++ Search:

- gives a C/C++ context sensitive search
  - can restrict the matches based on how they are used

Navigating Code – Outline view

The outline view:

- gives a filterable outline of your source file
- clicking on an element will take the editor there
- opens header files
- Refactor allows wide renaming
- quick access to search for references or declarations

Navigating Code – Open Definition/Declaration

Quick navigation:

- Open Definition and Open Declaration work on a selected object
  - e.g. function, variable, structure

- keyboard shortcuts are F3 for Open Declaration and Ctrl-F3 for Open Definition

Navigating Code – Which file is it?

Sometimes you don't know which file you have open:

- hover help on editor title bar will give path:

- show in C/C++ Projects will take you to the current element in that view:

- toggling link with editor will scroll the C/C++ Projects view to the current active editor:

Navigating Code – Code Folding

Navigating large or long source files can be difficult:

- code folding simplifies this:
  - folds functions, structures, etc into a single line
  - hover help shows contents
  - can expand only bits you need to read
  - enabled through Window→Preferences→C/C++→Editor→Folding:

Code Folding

A folded source file:

rbt_server.c

```
 75 int
 76⊕main(int argc, char **argv)
137
138 static void
139⊕say (int rcvid, char *text)
147
148 static void
149⊕raise_left_arm (int rcvid)
161
162 static void
163⊕lower_left_arm (int rcvid)
175
176 static void
177⊕raise_right_arm (int rcvid)
189
190 static void
191⊕lower_right_arm (int rcvid)
203
204 /*
205  * handle_pulse
206  *
207  * Because we did a name_attach(), we can expect the kernel to
208  * send us pulse messages under certain circumstances.  See the
209  * name_attach() docs for more.
210 */
211 static void
212⊕handle_pulse (struct _pulse *pulse)
```

Code Folding – Hover expand

Hovering shows the folded contents:

```
rbt_server.c
 75 int
 76⊕main(int argc, char **argv)
137
138 static void
139⊕say (int rcvid, char *text)
147
148 static void
149⊕raise_left_arm (int rcvid)
161
162 static void
163 lower_left_arm (int rcvid)
175 {
176     if (left_arm_state == RAISED) {
177⊕        /* pretend we make the robot lower its left arm */
189         printf ("%s:  robot lowered left arm\n", progname);
190         left_arm_state = LOWERED;
191⊕    } else
203         printf ("%s:  left arm already lowered\n", progname);
204     if (MsgReply (rcvid, EOK, NULL, 0) == -1) {
205         fprintf (stderr, "%s:  MsgReply() failed\n", progname);
206     }
207 }
208
209  * name_attach() docs for more.
210 */
211 static void
212⊕handle_pulse (struct _pulse *pulse)
```

Code Folding - Expanding

You can selectively expand the sections you need to look at:

```
rbt_server.c
138 static void
139 say (int rcvid, char *text)
147
148 static void
149 raise_left_arm (int rcvid)
161
162 static void
163 lower_left_arm (int rcvid)
164 {
165     if (left_arm_state == RAISED) {
166         /* pretend we make the robot lower its left arm */
167         printf ("%s:  robot lowered left arm\n", progname);
168         left_arm_state = LOWERED;
169     } else
170         printf ("%s:  left arm already lowered\n", progname);
171     if (MsgReply (rcvid, EOK, NULL, 0) == -1) {
172         fprintf (stderr, "%s:  MsgReply() failed\n", progname);
173     }
174 }
175
176 static void
177 raise_right_arm (int rcvid)
189
190 static void
191 lower_right_arm (int rcvid)
```

Navigating Code – Helpful hotkeys

Some additional helpful hotkeys:

- next editor editor: Ctrl-F6
- next perspective: Ctrl-F8
- go to matching bracket/brace: Ctrl-Shift-P
  - cursor must be just after bracket/brace
- “standard” cut/copy/paste: Ctrl-X/Ctrl-C/Ctrl-V

Hotkeys – Changing them

Hotkeys can be changed with:

– Window->Preferences->General->Keys:

e.g. bind F12 to “Hide Editors”:

Select category

Find the operation you want

In “Name:” field, you can try different combinations to see what they’re bound to.

Select something that is unbound, or that you don’t mind losing

View | Modify

Scheme: Default

Command

Category: Window

Name: Hide Editors

Assignments:

When	Key Sequence
In Windows	Ctrl+F12

Key Sequence

Name: Ctrl+F12

Assignments:

When	Command
In Windows	Hide Editors

When: In Windows

Add | Remove | Restore

Topics:

- **Using the C/C++ Editor**
  - Writing Code
  - Navigating Code
  - Oops, What Have I Done?
- **Compiling**
- **Fixing Errors**
- **Exercise**
- **Conclusion**

Oops – Before you save

Oops what have I done? Before saving:

- Ctrl-Z undo, Ctrl-Y redo
  - undo history is about 25 edits
- as you edit, changed lines are highlighted in pink in the left margin:

```
90}
91
92int read_filenames (char *dirname, char ***ifilenames)
93{
94    char **filenames = NULL, **tmp, *tmp2;
95    int nfilenames = 0;
```

- hover over changes will show the original:

```
91
92int read_filenames (char *dirname, char ***ifilenames)
93{
94>   char **filenames = NULL, **tmp;
95    int nfilenames = 0;
```

Oops – After saving

After saving:

– use local history:

Topics:

- Using the C/C++ Editor
  - Writing Code
  - Navigating Code
  - Oops, What Have I Done?
- → Compiling
- Fixing Errors
- Exercise
- Conclusion

The IDE calls compiling “building”:

– to build from the IDE, right-click on your project...

this will build only what needs to be built (similar to **`make -k all`**)

this will remove all files that are not source, e.g. executables, object modules, error files, … (similar to **`make -k clean`**)

Compiling - Building all projects

You can build all projects:

- you can also build all projects by pressing Ctrl-B
- the order things will be built is given in Window→Preferences→General→Workspace→Build Order

Compiling - Automatic building

Building can also happen automatically when:

- you save (modify) a resource such as a file or preferences:
  - Window→Preferences→General→Workspace:

- you run or debug a program:
  - Window→Preferences→Run/Debug→Launching:

Topics:

- **Using the C/C++ Editor**
  - Writing Code
  - Navigating Code
  - Oops, What Have I Done?
- **Compiling**
- → **Fixing Errors**
- **Exercise**
- **Conclusion**

Fixing Errors - Console view

The Console view:

- displays the output of building as it progresses:

Problems | Console | Target File System Navigator | Search | Properties | History

C-Build [editing_and_compiling]

```
cli.c:50: parse error before `:'
cli.c:58: warning: implicit declaration of function `print'
cli.c:61: warning: control reaches end of non-void function
cc: C:/QNX630/host/win32/x86/usr/lib/gcc-lib/ntox86/2.95.3/cc1 caught signal 33
make: *** [cli] Error 1
gcc -g -Vgcc_ntox86 -Wc,-Wall  -g -Vgcc_ntox86  ser.c   -o ser
ser.c: In function `main':
ser.c:37: warning: implicit declaration of function `getpid'
ser.c:44: warning: suggest parentheses around assignment used as truth value
make: Target `all' not remade because of errors.
```

- the output is saved per project
- selecting a project causes the view to show the build output for that project

Fixing Errors - The Problems view

The Problems view:

- lists problems (warnings and errors) messages that result from building

Problems | Console | Target File System Navigator | Search | Properties | History

1 error, 4 warnings, 0 infos

Description	Resource	Path	Location
Errors (1 item)			
parse error before ';'	cli.c	editing_and_compiling	line 50
Warnings (4 items)			
warning: control reaches end of non-voi	cli.c	editing_and_compiling	line 61
warning: implicit declaration of function	ser.c	editing_and_compiling	line 37
warning: implicit declaration of function	cli.c	editing_and_compiling	line 58
warning: suggest parentheses around a	ser.c	editing_and_compiling	line 44

- for messages that include location information (a line number):
  - double-clicking on the message open an editor at that line
  - if the file is the currently being edited one, clicking on the message will scroll to the offending line

Fixing Errors - Problem indicators

Many other places indicate problems:

clicking on these will go to next and previous problems
(you must be in the editor for these to be enabled)

indicates that there is a problem(s)

these markers represent problems. Their relative position vertically represents their locations in the file. You can click on these

Topics:

- **Using the C/C++ Editor**
  - **Writing Code**
  - **Navigating Code**
  - **Oops, What Have I Done?**
- **Compiling**
- **Fixing Errors**
- **Exercise**
- → **Conclusion**

Conclusion

You learned how:

- the IDE helps you write and navigate source code including how to:
  - trigger and use code completion
  - fold source files for easier viewing
  - search your source for what you need
- to build both individual projects as well as all your projects
- to use the Problems view and other parts of the IDE to find your errors

Running and Debugging

Introduction

You will learn:

- some ways of running your executable
- how to start up a debugging session
- various debugging techniques such as:
  - stepping through code as it executes
  - stopping your code at various points and when specific conditions are met
  - looking at and changing your data

Topics:

→ **Overview**

Setup

Running or Debugging

Exercise

Overview of the Debug Perspective

Debugging Techniques

- Stepping through code
- Breakpoints
- Viewing and Changing data

Exercise

Conclusion

Self-hosted vs Remote development:

Self-hosted:

PC running QNX® Neutrino®

Remote:

Host running Windows/Linux/Neutrino

Target running QNX® Neutrino®

What is a Launch Configuration?

- for the remote case, the IDE needs to know:
  - where to run or debug your program
  - how to get your program there
- in all cases, the IDE also needs to know:
  - what program to run
  - command line options
  - environment variables
  - special tools or settings
  - etc…
- a Launch Configuration is where the IDE stores this information
  - seems a lot of work up front, but
  - you only need to enter it once
  - the same one can be used for both running and debugging

Types of debugging supported:

- single threaded process

- multithreaded process

- multiprocess

- multitarget

- postmortem

Topics:

- **Overview**
- → **Setup**
- **Running or Debugging**
- **Exercise**
- **Overview of the Debug Perspective**
- **Debugging Techniques**
  - Stepping through code
  - Breakpoints
  - Viewing and Changing data
- **Exercise**
- **Conclusion**

Setup - Target side

qconn and pdebug:

- **qconn** is a program that gives all kinds of information to and does all kinds of work for the IDE
  - you should have **qconn** running on your target
  - you must be root (user ID 0) to run **qconn**
- **pdebug** is a debug agent, it acts as the interface between the IDE and the process being debugged
  - **pdebug** will generally be started as needed
  - **pdebug** requires pseudo-terminals (ptys), i.e. **devc-pty** must be running, and
  - **pdebug** requires a shell (e.g. **ksh**) to be available on the target

Processes involved in debugging:

Self-hosted:

Remote:

Setup – Launch configuration

Before running or debugging you need a Launch configuration:

- setup (wizard) is similar for both
- the configuration, once created, can be used for either debugging or running
- how you start depends on whether you initially want to run or debug

create launch config and run

create launch config and debug

Launch configuration – select type:

- the usual case:
  - C/C++ QConn (IP)
- if running self-hosted:
  - C/C++ Local Application
  - this will limit some features
- if you only have a serial connection to the target:
  - C/C++ Pdebug(Serial)
  - often you're better off using PPP in this case

Launch Configuration - Minimum

The minimum you need is:

Name for the launch configuration: pick something descriptive

What program to run: specified by Project & Application

Where to run it: if you haven't created a target project, do it here.

Creating a Target System project:

from the Launch wizard, click Add New Target
or right-click in here
and choose Add New Target

fill in a name representing your target. This will be the Target System project's name

fill in your target's IP address. `qconn` uses port 8000 by default so you wouldn't normally change this

choose Finish and your Target System project will be created

Setup - Debugging information

Also, to debug, you must have debug information in your executable:

- to get this debug information:
  - if you create a Standard Make C Project then compile and link with the **-g** option, e.g.:

```
qcc -g -o hello hello.c
```

  - if you create a QNX C or C++ Project then select a debug build variant:

  - or you can select this for an existing project by Right-Clicking on the project→Properties→QNX C/C++ Project→Build Variants

Topics:

- **Overview**
- **Setup**
- → **Running or Debugging**
- **Exercise**
- **Overview of the Debug Perspective**
- **Debugging Techniques**
  - Stepping through code
  - Breakpoints
  - Viewing and Changing data
- **Exercise**
- **Conclusion**

Running or Debugging – The final step

Once you're ready, click:

- Run to run your program:

- Debug to debug your program:

Running or Debugging – Re-using a Launch Configuration

If you've already created and used a launch configuration then:

clicking on the Debug or Run buttons will debug or run the last thing you launched (ran or debugged). This is quite useful in a code-debug cycle.

click on this triangle beside the Debug or Run buttons to get:
a history of your last few launches, allowing you to easily select one

Running - Other methods

Other ways of running an executable are:

- drag and drop it from the C/C++ Projects view to the Target File System Navigator view

- now that it's on the target, you can:
  - run it from a telnet session
  - run it from a Serial Terminal view
  - double-click on the executable in the Target File System Navigator view and it will run
    - can be run with or without a console…

Running - Other methods

Double-clicking an executable in the Target File System Navigator gives:

Click Advanced to get:

Executable can be run in the console view by checking here, otherwise you won't get

Topics:

- **Overview**
- **Setup**
- **Running or Debugging**
- → **Exercise**
- **Overview of the Debug Perspective**
- **Debugging Techniques**
  - Stepping through code
  - Breakpoints
  - Viewing and Changing data
- **Exercise**
- **Conclusion**

Exercise

Running a program exercise:

- run the **`listfiles`** program from your **`running_and_debugging`** project

Topics:

- **Overview**
- **Setup**
- **Running or Debugging**
- **Exercise**
- → **Overview of the Debug Perspective**
- **Debugging Techniques**
  - Stepping through code
  - Breakpoints
  - Viewing and Changing data
- **Exercise**
- **Conclusion**

For the remaining slides:

- we'll use the **`listfiles`** program. By default it displays the names of the files in **`/proc/boot`**

- it's in your **`run_debug`** project
- it's a Standard Make C Project so the **`Makefile`** contains the **`-g`** option to put debugging information into the executable
- it's made from two source files:

  `main.c` - contains *main()* and setup code

  `files.c` - manages filenames

Overview of the Debug Perspective

The Debug perspective:

Debug - main.c - QNX Momentics IDE

File Edit Refactor Navig Window Help

Debug view: your main control panel

Debug

listfiles [C/C++ QNX QConn (IP)]
QNX GDB Debugger (2/5/07 4:45 PM) (Suspended)
Thread [1] (Suspended)
1 main() at main.c:58 0x08048801
/tmp/listfilesDAGbbs1170711938343 on my_target pid 37685
C:\QNX630\host\win32\x86\usr\bin\ntox86-gdb.exe (2/5/07

Variables Breakpoints

Name	Value
argc	1
argv	0x08047ee4
nfilenames	0
filenames	0xffffffff

0x8047ee4

variables and other information

main.c asdf.c

```
nfilenames = read_filenames (dirnam
if (nfilenames == 0)
    exit (EXIT_FAILURE);
```

editors with your source

Outline

stdio.h
stdlib.h
unistd.h
read_filenames
sort_filenames
display_filenames

Console Task

listfiles [C/C++ QNX QCon (2/5/07 4:45 PM)

indicates next line to be executed

The Debug view:

- is the main view for terminating, relaunching program
- for stepping through code
- controls what is in the Variables view, editor, ...

- what you see above will vary based on:
  - how many threads exist
  - what state the program is in (suspended, terminated, …)

The Debug View - Examining the call stack

The call stack:

- the Debug view shows the call stack

- in the above example, we can see *options()* was called from *main()*
- as you select the items in the call stack, the Variables view will change to show the variables and their values that are on the stack for that function

The Debug view menu:

- right-clicking on an item will give a menu
- the menu contents will reflect the item

Example:

line execution stopped before

options for controlling which line to execute next

The Debug view control buttons:

– details on some of the above icons:

Icon	Hotkey	What it does
	F8	Resume - run in debugger from current point
		Suspend - regain control of running process
		Terminate - kill process
		Disconnect – detach debugger without killing process
		Remove All Terminated Launches - remove terminated processes from the Debug view
		Instruction Stepping Mode - toggle step by source line/ assembly instruction, most useful with Disassembly view

Topics:

- **Overview**
- **Setup**
- **Running or Debugging**
- **Exercise**
- **Overview of the Debug Perspective**
- **Debugging Techniques**
  - Stepping through code
  - Breakpoints
  - Viewing and Changing data
- **Exercise**
- **Conclusion**

When execution stops:

- the line to be executed next is indicated with an arrow

```
main.c
73{
74    int c;
75
76    while ((c = getopt(argc, argv, "d:")) != -1) {
77        switch (c) {
78        case 'd': dirname = optarg; break;
79        }
80    }
81}
```

- at this point you have a number of execution options:
  - resume, terminate, step intro, step over, run to return, and run to C/C++ line
- we've already seen the first two. Let's look at the others...

Stepping and running:

– details on some of the above icons:

Icon	Hotkey	What it does
	F5	Step Into - execute current line and if it's a function that you have debugging information for then make the new current line the first line in the function (i.e. step into it)
	F6	Step Over - execute current line and if it's a function, regardless of whether we have debugging information for it, don't step into it. Execute the function and make the next line the current line instead (i.e. step over the function)
	F7	Step Return – finish this function

Run to C/C++ line:

current line

desired current line

```
        nfilenames++; /* new n
    }
    closedir (dirp);

    *ifilenames = filenames;
```

References
Search Text
Run to Line Ctrl+R
Resume At Line

1 click on the line you want to run to, then right-click for the menu

2 from the menu, choose Run To Line

Topics:

- **Overview**
- **Setup**
- **Running or Debugging**
- **Exercise**
- **Overview of the Debug Perspective**
- **Debugging Techniques**
  - Stepping through code
  - Breakpoints
  - Viewing and Changing data
- **Exercise**
- **Conclusion**

Breakpoints - Definition and setting them

A breakpoint is a place you want the program to stop and return control to you:

to set a breakpoint on this line, right-click in the grey margin here and choose Toggle Breakpoint Or double-click in the grey margin at the line you want

```
files.c
        nswapped = 0;
        for (i = 0; i < nfilenames-1; i++)
            if (!strcmp (filenames[i], filenames[i+1])) {
                tmp = filenames[i];
                filenames[i] = filenames[i+1];
                filenames[i+1] = tmp;
                nswapped++;
```

Toggle Breakpoint
Disable Breakpoint
Breakpoint Properties...
Add Bookmark...

the end result is this indicator telling you that a breakpoint has been set for this line

```
files.c
79      nswapped = 0;
80      for (i = 0; i < nfilenames-1; i++)
81          if (!strcmp (filenames[i], filenames[i+1])) {
82              tmp = filenames[i];
83              filenames[i] = filenames[i+1];
```

- if you resume execution from wherever it is stopped, it'll run until it either terminates or reaches the above line. If it reaches the above line then it would stop before executing the line.

Breakpoints can also be conditional:

- execution would stop at the breakpoint only if a specific condition has been met

right-click on an existing breakpoint indicator and choose Breakpoint Properties…

… the Breakpoint Properties window appears

- the Condition and Ignore count are evaluated by **gdb** on the host each time the break point is hit:
  - if the condition is not true, **gdb** resumes the process
  - if the count has not been reached, **gdb** resumes the process

Breakpoints - Listing breakpoints

The Breakpoints view lists your breakpoints:

by right-clicking on a breakpoint you get this menu…
… with these useful options

- breakpoints live past the end of a debug session:
  - you don't have to add them again
  - the icon changes: versus

Topics:

- **Overview**
- **Setup**
- **Running or Debugging**
- **Exercise**
- **Overview of the Debug Perspective**
- **Debugging Techniques**
  - Stepping through code
  - Breakpoints
  - Viewing and Changing data
- **Exercise**
- **Conclusion**

Looking at variables:

- while in an editor, hold the mouse pointer steady over a variable and a balloon will pop-up

```
main.c   asdf.c
60          exit (EXIT_FAILURE);
61
62      sort_filenames (filenames, nfilenames);
63                                 nfilenames = 28
64      display_filenames (filenames, nfilenames);
```

- otherwise use the:
  - Variables view for local and global variables
  - Expressions view for more complex expressions (which can contain variables)

Viewing and Changing Data

Various data views are available:

- by default you get Variables and Breakpoints views:

- Other useful views to add include Expressions, Memory, and Registers:

The Variables view:

select globals to be shown

yellow means its value changed while we were stepping

allows a pointer to be interpreted as an array

use this to change the value of a variable

- shows local variables in current stack frame
- can display global variables as well:
  - will prompt with all global variables in program (including implementation ones)

Viewing and Changing Data - Expressions view

The Expressions view:

expressions will be evaluated as you step

you can create new expressions

or modify existing ones

– you can also add watch expressions from the C/C++ editor:

The Memory view:

click to enter an address or a variable that equates to an address

can track multiple memory areas, click to select each

Variables | Breakpoints | Expressions | Memory

Monitors | Renderings

*filenames
argv[0]

*filenames : 0x804D070 <Hex>

Address	0 - 3	4 - 7	8 - B	C - F
0804D070	00000000	B4FFFFFF	73746172	7475702D
0804D080	73637269	70740000	00000000	9CFFFFFF
0804D090	756E6C69	6E6B5F6[illegible]	69737400	00000000
0804D0A0	00000000	84FFFFFF	70726F63	6E746F2D
0804D0B0	696E7374	72000000	00000000	6CFFFFFF
0804D0C0	6C696263	2E736F2E	32000000	00000000
0804D0D0	00000000	54FFFFFF	6C696268	69646469
0804D0E0	2E736F2E	31000000	00000000	3CFFFFFF
0804D0F0	6C696268	69646469	2E736F00	00000000
0804D100	00000000	24FFFFFF	6C696275	73626469
0804D110	2E736F2E	32000000	00000000	0CFFFFFF
0804D120	6C696275	73626469	2E736F00	00000000

you can type here to change the memory

Topics:

- **Overview**
- **Setup**
- **Running or Debugging**
- **Exercise**
- **Overview of the Debug Perspective**
- **Debugging Techniques**
  - Stepping through code
  - Breakpoints
  - Viewing and Changing data
- **Exercise**
- → **Conclusion**

Conclusion

You learned:

- how to run or debug a program by using a launch configuration
- the set up needed for debugging
- basic debugging techniques:
  - stepping through your code
  - setting breakpoints
  - examining and changing the values of variables and memory

QNX 4.25 Architecture

Agenda

Topics:

QNX Neutrino 4.25 Architecture

- Overview
- The Microkernel
- The Process Manager
- Inter Process Communication
- Scheduling

What is QNX?

The QNX Operating System is ideal for realtime applications. It provides multitasking, priority-driven preemptive scheduling, and fast context switching - all essential ingredients of a realtime system.

Overview

- QNX is also remarkably flexible. Developers can easily customize the operating system to meet the needs of their application.

- QNX achieves its unique degree of efficiency, modularity, and simplicity through two fundamental principles:
  - Microkernel architecture
  - Message-based interprocess communication

Architecture

MicroKernel

The QNX Microkernel is responsible for the following:

- IPC - the Microkernel supervises the routing of *messages*; it also manages two other forms of
  - IPC: *proxies* and *signals*
- low-level network communication - the Microkernel delivers all messages destined for processes on other nodes

Microkernel

- process scheduling - The Microkernel's *scheduler* decides which process will execute next.
- first-level interrupt handling - all hardware interrupts and faults are first routed through the Microkernel, then passed on to the appropriate driver or system manager

Inside of Microkernel