Contents

About This Guide v
Typographical conventions vii
  Note to Windows users viii
What you’ll find in this guide ix

1 What is Multicore Processing? 1

2 A Quick Introduction to Multicore Processing 5
  Setting up the OS image 7
  Trying symmetric multiprocessing 8
  Trying bound multiprocessing 9

3 Developing Multicore Systems 11
  Building a multicore image 13
  The impact of multicore 13
    To multicore or not to multicore 14
    Thread affinity 14
    Multicore and synchronization primitives 18
    Multicore and FIFO scheduling 18
    Multicore and interrupts 18
    Multicore and atomic operations 19
    Adaptive partitioning 20
  Designing with multiprocessing in mind 20
    Use the multicore primitives 21
Assume that threads *really do* run concurrently 21
Break the problem down 21

Glossary 25

Index 29
Typographical conventions

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

<table>
<thead>
<tr>
<th>Reference</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code examples</td>
<td><code>if( stream == NULL )</code></td>
</tr>
<tr>
<td>Command options</td>
<td><code>-lR</code></td>
</tr>
<tr>
<td>Commands</td>
<td><code>make</code></td>
</tr>
<tr>
<td>Environment variables</td>
<td><code>PATH</code></td>
</tr>
<tr>
<td>File and pathnames</td>
<td><code>/dev/null</code></td>
</tr>
<tr>
<td>Function names</td>
<td><code>exit()</code></td>
</tr>
<tr>
<td>Keyboard chords</td>
<td><code>Ctrl – Alt – Delete</code></td>
</tr>
<tr>
<td>Keyboard input</td>
<td><code>something you type</code></td>
</tr>
<tr>
<td>Keyboard keys</td>
<td><code>Enter</code></td>
</tr>
<tr>
<td>Program output</td>
<td><code>login:</code></td>
</tr>
<tr>
<td>Programming constants</td>
<td><code>NULL</code></td>
</tr>
<tr>
<td>Programming data types</td>
<td><code>unsigned short</code></td>
</tr>
<tr>
<td>Programming literals</td>
<td><code>0xFF, &quot;message string&quot;</code></td>
</tr>
<tr>
<td>Variable names</td>
<td><code>stdin</code></td>
</tr>
<tr>
<td>User-interface components</td>
<td><code>Cancel</code></td>
</tr>
</tbody>
</table>

We format single-step instructions like this:

➤ To reload the current page, press Ctrl – R.

We use an arrow (→) in directions for accessing menu items, like this:
You’ll find the Other... menu item under Perspective→Show View.

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

Notes point out something important or useful.

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

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

Note to Windows users

In our documentation, we use a forward slash (/) as a delimiter in all pathnames, including those pointing to Windows files.

We also generally follow POSIX/UNIX filesystem conventions.
What you’ll find in this guide

The Multicore Technology Development Kit User’s Guide describes how you can use symmetric multiprocesssing to get the most performance possible out of a multiprocessor system. It also describes how to use bound multiprocessing to restrict which processors a thread can run on.

In order to use symmetric or bound multiprocessing, you need the Multicore Technology Development Kit (TDK).

The following table may help you find information quickly in this guide:

<table>
<thead>
<tr>
<th>For information on:</th>
<th>Go to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multicore processing in general</td>
<td>What is Multicore Processing?</td>
</tr>
<tr>
<td>Getting started with multicore</td>
<td>A Quick Introduction to</td>
</tr>
<tr>
<td>processing</td>
<td>Multicore Processing</td>
</tr>
<tr>
<td>Programming with multicore</td>
<td>Developing Multicore Systems</td>
</tr>
<tr>
<td>processing in mind</td>
<td></td>
</tr>
<tr>
<td>Terminology used in this guide</td>
<td>Glossary</td>
</tr>
</tbody>
</table>
Chapter 1

What is Multicore Processing?
Multiprocessing systems, whether discrete or multicore, can greatly improve your applications’ performance. As described in the Multicore Processing chapter of the *System Architecture* guide, there’s a multiprocessor version of Neutrino that runs on:

- Pentium-based multiprocessor systems that conform to the Intel MultiProcessor Specification (MP Spec)
- MIPS-based systems
- PowerPC-based systems

If you have one of these systems, then you’re probably itching to try it out, but are wondering what you have to do to get Neutrino running on it. Well, the answer is not much. The only part of Neutrino that’s different for a multiprocessor system is the microkernel — another example of the advantages of a microkernel architecture!

Neutrino supports these operating modes for multiprocessing:

**Asymmetric multiprocessing (AMP)**

A separate OS, or a separate instantiation of the same OS, runs on each CPU.

**Symmetric multiprocessing (SMP)**

A single instantiation of an OS manages all CPUs simultaneously, and applications can float to any of them.

**Bound multiprocessing (BMP)**

A single instantiation of an OS manages all CPUs simultaneously, but you can lock individual applications or threads to a specific CPU.

SMP lets you get the most performance out of your system, but you might need to use BMP for the few applications that may not work under SMP, or if you want to explicitly control the process-level distribution of CPU usage.
Chapter 2

A Quick Introduction to Multicore Processing

In this chapter...

- Setting up the OS image 7
- Trying symmetric multiprocessing 8
- Trying bound multiprocessing 9
This chapter gives you a quick hands-on introduction to multicore processing. The main steps are as follows:

- Setting up the OS image
- Trying symmetric multiprocessing
- Trying bound multiprocessing

## Setting up the OS image

1. Log in as `root`.
2. Go to the directory that holds the buildfile for your system’s boot image (e.g. `/boot/build`).
3. Create a copy of the buildfile:
   ```bash
cp qnxbasedma.build qnxbasedma_multicore.build
   ```
4. Edit the copy (e.g. `qnxbasedma_multicore.build`).
5. Search for `procnto`. The line might look like this:
   ```bash
   PATH=/proc/boot:/bin:/usr/bin:/opt/bin \\
   LD_LIBRARY_PATH=/proc/boot:/lib:/usr/lib:/lib/dll:/opt/lib \\
   procnto-instr
   ```
   In a real buildfile, you can’t use a backslash (`\`) to break a long line into shorter pieces, but we’ve done that here, just to make the command easier to read.
6. Change `procnto` to the appropriate multicore version; see `/proc/boot` to see which uniprocessor version you’re using, and then add `-smp` to it. For more information, see `procnto` in the Utilities Reference. For example:
   ```bash
   PATH=/proc/boot:/bin:/usr/bin:/opt/bin \\
   LD_LIBRARY_PATH=/proc/boot:/lib:/usr/lib:/lib/dll:/opt/lib \\
   procnto-smp-instr
   ```
Although the multiprocessing version of procnto has “SMP” in its name, it also supports BMP. You can even use bound and symmetric multiprocessing simultaneously on the same system.

7. Save your changes to the buildfile.

8. Generate a new boot image:

   mkifs qnxbasedma_multicore.build qnxbasedma_multicore.ifs

9. Put the new image in place. We recommend that you copy your current boot image to /.altboot as a backup in case something goes wrong:

   cp /.altboot /.old_altboot
   cp /.boot /.altboot
   cp qnxbasedma_multicore.ifs /.boot

10. Reboot your system.

---

**Trying symmetric multiprocessing**

1. Log in as a normal user.

2. Start some processes that run indefinitely. For example, use the hogs utility to display which processes are using the most CPU:

   hogs -n -%10

3. Use pidin sched to see which processor your processes are running on.

   If you’re using the IDE, you can use the System Information perspective to watch the threads migrate.

4. Create a program called greedy.c that simply loops forever:
#include <stdlib.h>

int main( void )
{
    while (1) {
    
    return EXIT_SUCCESS;
    }

Compile it, and then run it:
qcc -o greedy greedy.c
./greedy &

On a uniprocessor system, this would consume all the processing time (unless you’re using adaptive partitioning). On a multicore system, it consumes all the time on one processor.

Use `pidin sched` to see which processor your other processes are running on. They’re likely running on different processors from `greedy`.

## Trying bound multiprocessing

1. Use the `-C` or `-R` option (or both) to the `on` utility to start a shell on a specific set of processors:
   ```
   on -C 0 ksh
   ```

2. Start some new processes from this shell. Note that they run only on the first processor.

3. Use the `-C` or `-R` option (or both) to `slay` to change the runmask for one of these processes. Note that the process runs only on the processors that you just specified, while any children run on the processors you specified for the shell.

4. Use the `-C` or `-R` option (or both) and the `-i` option to `slay` to change the runmask and inherit mask for one of these
processes. Note that the process and its children run only on the newly specified processors.
Chapter 3
Developing Multicore Systems

In this chapter...

Building a multicore image 13
The impact of multicore 13
Designing with multiprocessing in mind 20
Building a multicore image

Assuming you’re already familiar with building a bootable image for a single-processor system (as described in the Making an OS Image chapter in Building Embedded Systems), let’s look at what you have to change in the buildfile for a multicore system.

As we mentioned earlier, basically all you need to use is the multicore kernel (procnto-smp) when building the image.

Here’s an example of a buildfile:

```
# A simple multicore buildfile
[virtual=x86,bios] .bootstrap = {
    startup-bios
    PATH=/proc/boot procnto-smp
}
[+script] .script = {
    devc-con -e &
    reopen /dev/con1
    [+session] PATH=/proc/boot esh
}

libc.so
[type=link] /usr/lib/ldqnx.so.2=/proc/boot/libc.so

[data=copy]
    devc-con
    esh
    ls
```

After building the image, you proceed in the same way as you would with a single-processor system.

The impact of multicore

Although the actual changes to the way you set up the processor to run SMP are fairly minor, the fact that you’re running on a multicore system can have a major impact on your software!

The main thing to keep in mind is this: in a single processor environment, it may be a nice “design abstraction” to pretend that threads execute in parallel; under a multicore system, they really do...
execute in parallel! (With BMP, you can make your threads run on a specific CPU.)

In this section, we’ll examine the impact of multicore on your system design.

**To multicore or not to multicore**

It’s possible to use the non-multicore kernel on a multicore box. In this case, only processor 0 will be used; the other processors won’t run your code. This is a waste of additional processors, of course, but it does mean that you can run images from single-processor boxes on an multicore box. (You can also run SMP-ready images on single-processor boxes.)

It’s also possible to run the multicore kernel on a uniprocessor system, but it requires a 486 or higher on x86 architectures, and a multicore-capable implementation on MIPS and PPC.

**Thread affinity**

One issue that often arises in a multicore environment can be put like this: “Can I make it so that one processor handles the GUI, another handles the database, and the other two handle the realtime functions?”

The answer is: “Yes, absolutely.”

This is done through the magic of *thread affinity*, the ability to associate certain programs (or even threads within programs) with a particular processor or processors.

Thread affinity works like this. When a thread starts up, its affinity mask (or runmask) is set to allow it to run on all processors. This implies that there’s no inheritance of the thread affinity mask, so it’s up to the thread to use `ThreadCtl()` with the _NTO_TCTL_RUNMASK_ control flag to set its runmask:

```c
if (ThreadCtl(_NTO_TCTL_RUNMASK, &my_runmask) == -1) {
  /* An error occurred. */
}
```
The runmask is simply a bitmap; each bit position indicates a particular processor. For example, the runmask 0x05 (binary 00000101) allows the thread to run on processors 0 (the 0x01 bit) and 2 (the 0x04 bit).

If you use _NTO_TCTL_RUNMASK, the runmask is limited to the size of an int (currently 32 bits). Threads created by the calling thread don’t inherit the specified runmask.

If you want to support more processors than will fit in an int, or you want to set the inherit mask, you’ll need to use the _NTO_TCTL_RUNMASK_GET_AND_SET_INHERIT command described below.

The <sys/neutralino.h> file defines some macros that you can use to work with a runmask:

\[ RMSK_SET(cpu, p) \]
\[ RMSK_CLR(cpu, p) \]
\[ RMSK_ISSET(cpu, p) \]

The CPUs are numbered from 0. These macros work with runmasks of any length.

Bound multiprocessing (BMP) is a variation on SMP that lets you specify which processors a process or thread and its children can run on. To specify this, you use an inherit mask.

To set a thread’s inherit mask, you use ThreadCtl() with the _NTO_TCTL_RUNMASK_GET_AND_SET_INHERIT control flag. Conceptually, the structure that you pass with this command is as follows:

\[ struct _thread_runmask { \]
If you set the `runmask` member to a nonzero value, `ThreadCtl()` sets the runmask of the calling thread to the specified value. If you set the `runmask` member to zero, the runmask of the calling thread isn’t altered.

If you set the `inherit_mask` member to a nonzero value, `ThreadCtl()` sets the calling thread’s inheritance mask to the specified value(s); if the calling thread creates any children by calling `pthread_create()`, `fork()`, `spawn()`, `vfork()`, and `exec()`, the children inherit this mask. If you set the `inherit_mask` member to zero, the calling thread’s inheritance mask isn’t changed.

If you look at the definition of `_thread_runmask` in `<sys/neutrino.h>`, you’ll see that it’s actually declared like this:

```c
struct _thread_runmask {
    int size;
    /* unsigned runmask[size]; */
    /* unsigned inherit_mask[size]; */
};
```

This is because the number of elements in the `runmask` and `inherit_mask` arrays depends on the number of processors in your multicore system. You can use the `RMSK_SIZE()` macro to determine how many unsigned integers you need for the masks; pass the number of CPUs (found in the system page) to this macro.

Here’s a code snippet that shows how to set up the runmask and inherit mask:

```c
unsigned num_elements = 0;
int    *rsizep, *masksize_bytes, *size;
unsigned *rmaskp, *imaskp;
void    *my_data;

/* Determine the number of array elements required to hold
   * the runmasks, based on the number of CPUs in the system. */
num_elements = RMSK_SIZE(_syspage_ptr->num_cpu);
```
/* Determine the size of the runmask, in bytes. */
masksize_bytes = num_elements * sizeof(unsigned);

/* Allocate memory for the data structure that we'll pass
  * to ThreadCtl(). We need space for an integer (the number
  * of elements in each mask array) and the two masks
  * (runmask and inherit mask). */
size = sizeof(int) + 2 * masksize_bytes;
if ((my_data = malloc(size)) == NULL) {
    /* Not enough memory. */
}
else {
    memset(my_data, 0x00, size);
    /* Set up pointers to the "members" of the structure. */
    rsizep = (int *)my_data;
    rmaskp = rsizep + 1;
    imaskp = rmaskp + num_elements;
    /* Set the size. */
    *rsizep = num_elements;
    /* Set the runmask. Call this macro once for each processor
    the thread can run on. */
    RMSK_SET(cpu1, rmaskp);
    /* Set the inherit mask. Call this macro once for each processor
    the thread's children can run on. */
    RMSK_SET(cpu1, imaskp);
    if (ThreadCtl(_NTO_TCTL_RUNMASK_GET_AND_SET_INHERIT,
                   my_data) == -1) {
        /* Something went wrong. */
    }
}

You can also use the -C and -R options to the on command to launch
processes with a runmask (assuming they don't set their runmasks
programmatically); for example, use on -C 1 io-net to start
io-net and lock all threads to CPU 1. This command sets both the
runmask and the inherit mask.
You can also use the same options to the `slay` command to modify the runmask of a running process or thread. For example, `slay -C 0 io-net` moves all of `io-net`’s threads to run on CPU 0. If you use the `-C` and `-R` options, `slay` sets the runmask; if you also use the `-i` option, `slay` also sets the process’s or thread’s inherit mask to be the same as the runmask.

**Multicore and synchronization primitives**

Standard synchronization primitives (barriers, mutexes, condvars, semaphores, and all of their derivatives, e.g. sleepon locks) are safe to use on a multicore box. You don’t have to do anything special here.

**Multicore and FIFO scheduling**

A common single-processor “trick” for coordinated access to a shared memory region is to use FIFO scheduling between two threads running at the same priority. The idea is that one thread will access the region and then call `SchedYield()` to give up its use of the processor. Then, the second thread would run and access the region. When it was done, the second thread too would call `SchedYield()`, and the first thread would run again. Since there’s only one processor, both threads would cooperatively share that processor.

This FIFO trick won’t work on an SMP system, because both threads may run simultaneously on different processors. You’ll have to use the more “proper” thread synchronization primitives (e.g. a mutex), or use BMP to tie the threads to specific CPUs.

**Multicore and interrupts**

The following method is closely related to the FIFO scheduling trick. On a single-processor system, a thread and an interrupt service routine are mutually exclusive, because the ISR runs at a higher priority than any thread. Therefore, the ISR can preempt the thread, but the thread can never preempt the ISR. So the only “protection” required is for the thread to indicate that during a particular section of code (the critical section) interrupts should be disabled.
Obviously, this scheme breaks down in a multicore system, because again the thread and the ISR could be running on different processors.

The solution in this case is to use the `InterruptLock()` and `InterruptUnlock()` calls to ensure that the ISR won’t preempt the thread at an unexpected point. But what if the thread preempts the ISR? The solution is the same: use `InterruptLock()` and `InterruptUnlock()` in the ISR as well.

We recommend that you always use `InterruptLock()` and `InterruptUnlock()`, both in the thread and in the ISR. The small amount of extra overhead on a single-processor box is negligible.

**Multicore and atomic operations**

Note that if you wish to perform simple atomic operations, such as adding a value to a memory location, it isn’t necessary to turn off interrupts to ensure that the operation won’t be preempted. Instead, use the functions provided in the C include file `<atomic.h>`, which let you perform the following operations with memory locations in an atomic manner:

<table>
<thead>
<tr>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>atomic_add()</code></td>
<td>Add a number</td>
</tr>
<tr>
<td><code>atomic_add_value()</code></td>
<td>Add a number and return the original value of *loc</td>
</tr>
<tr>
<td><code>atomic_clr()</code></td>
<td>Clear bits</td>
</tr>
<tr>
<td><code>atomic_clr_value()</code></td>
<td>Clear bits and return the original value of *loc</td>
</tr>
<tr>
<td><code>atomic_set()</code></td>
<td>Set bits</td>
</tr>
<tr>
<td><code>atomic_set_value()</code></td>
<td>Set bits and return the original value of *loc</td>
</tr>
</tbody>
</table>

*continued...*
Function | Operation
----------|------------------
atomic_sub() | Subtract a number
atomic_sub_value() | Subtract a number and return the original value of *loc
atomic_toggle() | Toggle (complement) bits
atomic_toggle_value() | Toggle (complement) bits and return the original value of *loc

The *value() functions may be slower on some systems (e.g. 386), so don’t use them unless you really want the return value.

Adaptive partitioning

You can use adaptive partitioning on a multicore system, but there are some interactions to watch out for. For more information, see “Using adaptive partitioning and multicore together” in the Adaptive Partitioning Scheduling Details chapter of the Adaptive Partitioning User’s Guide.

In order to use adaptive partitioning, you must install the Adaptive Partitioning Technology Development Kit.

Designing with multiprocessing in mind

You may not have a multicore system today, but wouldn’t it be great if your software just ran faster on one when you or your customer upgrade the hardware?

While the general topic of how to design programs so that they can scale to \(N\) processors is still the topic of research, this section contains some general tips.
Use the multicore primitives

Don’t assume that your program will run only on one processor. This means staying away from the FIFO synchronization trick mentioned above. Also, you should use the multicore-aware `InterruptLock()` and `InterruptUnlock()` functions.

By doing this, you’ll be “multicore-ready” with little negative impact on a single-processor system.

Assume that threads really do run concurrently

As mentioned above, it isn’t merely a useful “programming abstraction” to pretend that threads run simultaneously; you should design as if they really do. That way, when you move to a multicore system, you won’t have any nasty surprises (but you can use BMP if you have problems and don’t want to modify the code).

Break the problem down

Most problems can be broken down into independent, parallel tasks. Some are easy to break down, some are hard, and some are impossible. Generally, you want to look at the data flow going through a particular problem. If the data flows are independent (i.e. one flow doesn’t rely on the results of another), this can be a good candidate for parallelization within the process by starting multiple threads. Consider the following graphics program snippet:

```c
void do_graphics ()
{
    int x;

    for (x = 0; x < XRESOLUTION; x++) {
        do_one_line (x);
    }
}
```

In the above example, we’re doing ray-tracing. We’ve looked at the problem and decided that the function `do_one_line()` only generates output to the screen — it doesn’t rely on the results from any other invocation of `do_one_line()`.
To make optimal use of a multicore system, you would start multiple threads, each running on one processor.

The question then becomes how many threads to start. Obviously, starting \texttt{XRESOLUTION} threads (where \texttt{XRESOLUTION} is far greater than the number of processors, perhaps 1024 to 4) isn’t a particularly good idea — you’re creating a lot of threads, all of which will consume stack resources and kernel resources as they compete for the limited pool of CPUs.

A simple solution would be to find out the number of CPUs that you have available to you (via the system page pointer) and divide the work up that way:

```c
#include <sys/syspage.h>

int num_x_per_cpu;

do_graphics ()
{
    int num_cpus;
    int i;
    pthread_t *tids;

    // figure out how many CPUs there are...
    num_cpus = _syspage_ptr -> num_cpu;

    // allocate storage for the thread IDs
    tids = malloc (num_cpus * sizeof (pthread_t));

    // figure out how many X lines each CPU can do
    num_x_per_cpu = XRESOLUTION / num_cpus;

    // start up one thread per CPU, passing it the ID
    for (i = 0; i < num_cpus; i++) {
        pthread_create (&tids[i], NULL, do_lines, (void *) i);
    }

    // now all the "do_lines" are off running on the processors

    // we need to wait for their termination
    for (i = 0; i < num_cpus; i++) {
        pthread_join (tids[i], NULL);
    }

    // now they are all done
}
```
Designing with multiprocessing in mind

```c
void *
do_lines (void *arg)
{
    int cpunum = (int) arg; // convert void * to an integer
    int x;

    for (x = cpunum * num_x_per_cpu; x < (cpunum + 1) * num_x_per_cpu; x++) { do_line (x); }
}
```

The above approach lets the maximum number of threads run simultaneously on the multicore system. There’s no point creating more threads than there are CPUs, because they’ll simply compete with each other for CPU time.

Note that in this example, we didn’t specify which processor to run each thread on. We don’t need to in this case, because the READY thread with the highest priority always runs on the next available processor. The threads will tend to run on different processors (depending on what else is running in the system). You typically use the same priority for all the worker threads if they’re doing similar work.

An alternative approach is to use a semaphore. You could preload the semaphore with the count of available CPUs. Then, you create threads whenever the semaphore indicates that a CPU is available. This is conceptually simpler, but involves the overhead of creating and destroying threads for each iteration.
Glossary
asymmetric multiprocessing (AMP)

A separate OS, or a separate instantiation of the same OS, runs on each CPU.

bound multiprocessing (BMP)

A single instantiation of an OS manages all CPUs simultaneously, but you can lock individual applications or threads to a specific CPU.

discrete (or traditional) multiprocessor system

A system that has separate physical processors hooked up in multiprocessing mode over a board-level bus.

hard thread affinity

A user-specified binding of a thread to a set of processors, done by means of a runmask. Contrast soft thread affinity.

inherit mask

A bitmask that specifies which processors a thread’s children can run on. Contrast runmask.

multicore system

A chip that has one physical processor with multiple CPUs interconnected over a chip-level bus.

runmask

A bitmask that indicates which processors a thread can run on. Contrast inherit mask.

soft thread affinity

The scheme whereby the microkernel tries to dispatch a thread to the processor where it last ran, in an attempt to reduce thread migration from one processor to another, which can affect cache performance. Contrast hard thread affinity.
**symmetric multiprocessing (SMP)**

A single instantiation of an OS manages all CPUs simultaneously, and applications can float to any of them.
## Index

<table>
<thead>
<tr>
<th>A</th>
<th>affinity, thread 14</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AMP (Asymmetric Multiprocessing) 3</td>
</tr>
<tr>
<td></td>
<td>atomic operations 19</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
<th>BMP (Bound Multiprocessing) 3, 15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>trying it 9</td>
</tr>
<tr>
<td></td>
<td>buildfiles</td>
</tr>
<tr>
<td></td>
<td>modifying for multicore processing 7</td>
</tr>
<tr>
<td></td>
<td>sample 13</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C</th>
<th>conventions, typographical vii</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPUs, number of 22</td>
</tr>
</tbody>
</table>

| F | FIFO scheduling, using with multicore 18 |

<table>
<thead>
<tr>
<th>I</th>
<th>images, building for multicore 13</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>inherit mask 15</td>
</tr>
<tr>
<td></td>
<td>InterruptLock() 19, 21</td>
</tr>
<tr>
<td></td>
<td>interrupts, handling 18</td>
</tr>
<tr>
<td></td>
<td>InterruptUnlock() 19, 21</td>
</tr>
<tr>
<td></td>
<td>ISR, preemption considerations 18</td>
</tr>
</tbody>
</table>
Index

M
multicore processing 3
  building an image for 13
  designing for 20
  interrupts and 18
  sample buildfile for 13
mutexes 18

N
_NTO_TCTL_RUNMASK 14
_NTO_TCTL_RUNMASK_GET_AND_SET_INHERIT 15

O
on utility 9, 17
operations, atomic 19
OS images, building for
  multicore 7, 13

P
pathname delimiter in QNX
  Momentics documentation viii
pidin 8
processes, processor running on
  displaying 8
  specifying 9
procno*-smp 7, 13

R
RMSK_CLR() 15
RMSK_ISSET() 15
RMSK_SET() 15
RMSK_SIZE() 16
runmask 14

S
scheduling algorithms, using FIFO
  with multicore 18
SchedYield(), using with multicore 18
slay 9, 17
SMP (Symmetric Multiprocessing) 3
  trying it 8
synchronization primitives and
  multicore 18
system page, number of CPUs 22

T
tasks, parallel 21
thread affinity 14
ThreadCtl() 14, 15
threads, running concurrently 13,
  21
typographical conventions vii