Instant Device Activation User's Guide

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

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

Electronic edition published: Monday, July 14, 2014

Table of Contents

About This Guide

The Instant Device Activation *User's Guide* will help you set up a "minidriver" to start devices quickly when the system boots. The following table may help you find information quickly in this guide:

For information on:	Go to:
An overview of Instant Device Activation	*Using Minidrivers for Instant Device Activation* (p. 9)
How to write instant device activation code	*Writing a Minidriver* (p. 13)
An example of the code for a minidriver	*Sample Minidriver* (p. 23)
API and datatypes	*API and Datatypes* (p. 31)
An example of interacting with hardware	*Hardware Interaction Within the Minidriver* (p. 37)

Typographical conventions

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

The following table summarizes our conventions:

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

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

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

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

Notes point out something important or useful.

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

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

Note to Windows users

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

Technical support

Technical assistance is available for all supported products.

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

Chapter 1
Using Minidrivers for Instant Device Activation

Advanced CPUs are providing higher levels of hardware integration than ever before, directly controlling such interfaces as CAN, J1850, and MOST. This approach saves on hardware costs by reducing the need for extra chips and circuitry, but it also raises concerns for the software developer.

For instance, a telematics control unit must be able to receive CAN messages within 30 to 100 milliseconds from the time that it's powered on. The problem is that the complex software running on such a telematics device can easily take hundreds of milliseconds—or more—to boot up.

As another example, consider the critical milestones during the boot process for an in-car telematics or infotainment unit that typically boots from a cold condition (completely powered off) or from CPU reboot condition (returning from a state where SDRAM has been turned off). The unit must be able to:

- receive CAN messages within 30 to 100 milliseconds after power is turned on
- respond to these messages within 100 milliseconds of receiving them
- read Class 2 messages from a vehicle bus and respond to wake events
- initialize a MOST transceiver and respond to MOST requests
- animate a splash screen (graphical display) before the operating system has loaded

In order to address such timing requirements, many embedded system designs rely on a simple but expensive solution that uses an auxiliary communications processor or external power module. This auxiliary hardware can be reduced in scope and sometimes even eliminated by using *Instant Device Activation*. Also called *minidriver technology*, the approach consists of small, highly efficient device drivers that start executing before the OS kernel is initialized.

The minidriver basics

During the normal QNX Neutrino boot process, a driver process can't run until the OS image has been loaded into the RAM, and the kernel has been initialized. Depending on the particular hardware (processor, flash, architecture) and the OS image size, this time can be in the order of hundreds of milliseconds or even seconds. To reduce this time, a minidriver runs much earlier in the boot process to take care of the timing requirements for some bus protocols such as MOST or CAN.

Defined in the system's startup code, a minidriver runs user code before the operating system has been booted. This code could include responding to hardware power-up messages in a quick, timely fashion and ensuring that no message is lost when the OS boots up. Once the OS has booted, the minidriver may continue running, or it may pass control to a full-featured driver that can access any data the minidriver has buffered.

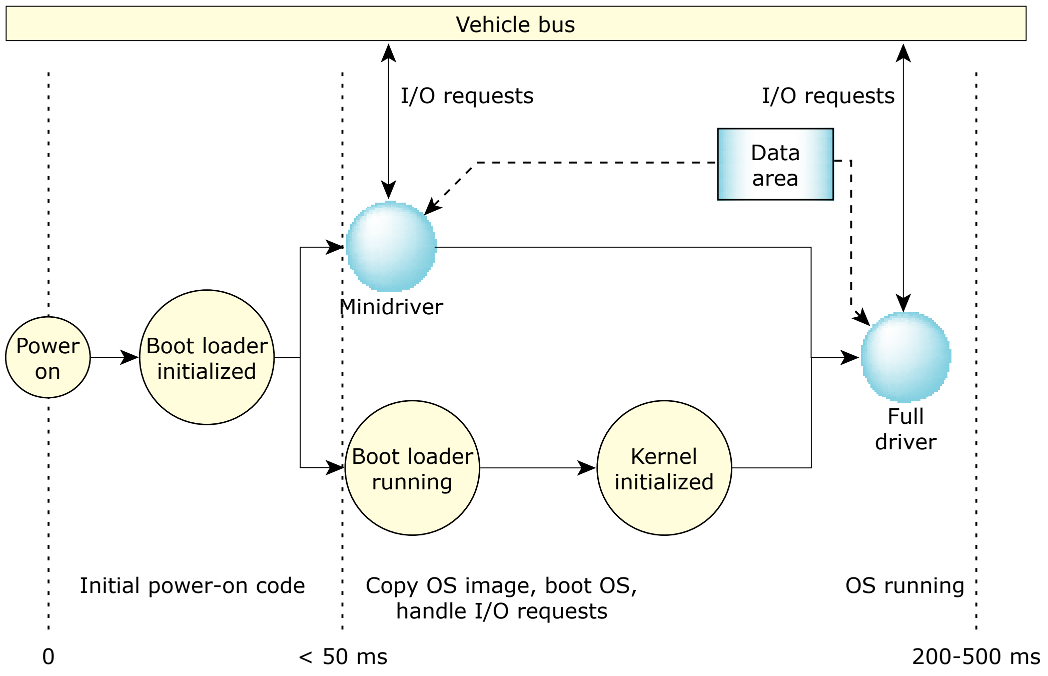

Figure 1: Booting process using instant device activation.

The minidriver architecture

A minidriver consists of these fundamental components:

- a handler function
- an optional data area
- startup code to create the data area and register the handler

Once a minidriver is created, its handler function is called throughout the booting process. The handler function is initially triggered from a timer (polling). Once CPU interrupts are enabled, the handler function is triggered by the real hardware interrupt. Note that the timers can also generate interrupts, allowing for a polled approach to be used for hardware that doesn't generate interrupts.

You can use the data area to store any information that the handler function needs to keep and potentially share with the full driver.

How does the minidriver work?

A minidriver is a function that you link to the QNX Neutrino startup program, so that it runs before the system becomes operational and the kernel is initialized. A minidriver can access hardware and store data in a RAM buffer area where a full (process-time) driver can then read this buffered information.

During system startup, a minidriver handler function is periodically called (or polled). You can adapt this periodic/polled interval to suit your device's timing requirements with minor changes to the startup program. At some point in the system startup, interrupts become available, and this handler function becomes interrupt driven. The handler is called with a state variable, so the handler knows why it was called.

Seamless transition

As soon as a full driver process is running in a fully operational system, transition takes place from the minidriver to a full driver. This transition is seamless and causes no blackout times. The full driver merely attaches to the device interrupt, which causes the minidriver to be notified that another process is attaching to its interrupt. The minidriver can then gracefully exit, and the full driver continues to run. The full driver has access to any buffered data that the minidriver chooses to store.

Running multiple handler functions

The minidriver can run multiple handler functions. For devices that must do something every *n* milliseconds, you could attach two handler functions:

- a minidriver for the actual device interrupt
- a minidriver for the system timer tick

Since the timer minidriver is intermittently polled (i.e., not invoked at a constant interval) during startup, the driver needs to use something to measure the time between the calls to get the proper interval.

This architecture allows device drivers to start very early in the system startup and allows the device to continue to function during all boot phases. If a full driver doesn't choose to take over device control, the minidriver continues to run when the system is fully operational.

Chapter 2
Writing a Minidriver

In order to write a minidriver, you must first decide on the following:

- the hardware platform you'll work with
- the timing requirements of your driver
- how much data storage (if any) the minidriver needs
- whether or not your minidriver needs to initialize the hardware
- whether or not your minidriver requires hardware access
- how the transition to the full driver is to be accomplished

The BSP associated with your hardware platform includes the source code to the board's `startup` program. You must link your minidriver to this program. For more information, see the BSP documentation, as well as *Building Embedded Systems*.

You'll need to modify these files:

- `mdriver_max.c` — defines the amount of data copied from flash to RAM between calls to your minidriver
- `main.c` — where you'll set up the minidriver's data area and register your handler
- *my_mdriver.c* — contains your handler function; choose an appropriate name for this file. You can have multiple C and header files as part of your minidriver.

Don't modify the following files unless you're directed to do so by QNX Software Systems, but make sure they're included in your startup code directory:

- `cpu_mdriver.c`
- `mdriver.c`

Timing requirements

Since the minidriver code is polled during the startup and kernel-initialization phases of the boot process, you need to know the timing of your device in order to verify if the poll rate is fast enough. Most of the time in startup is spent copying the boot image from flash to RAM, and the minidriver is polled during this time period. The sequence might look like this:

- Call the minidriver.
- Copy the next 16 KB.
- Call the minidriver.
- Copy the next 16 KB.
- ...

The startup library contains a global variable *mdriver_max* (p. 36), which is the amount of data (in bytes) that's copied from flash to RAM between calls to your minidriver. This variable is defined in `mdriver_max.c`, and its default value is 16 KB.

You might have to experiment to determine the best value, based on the timing requirements of your device, processor speed, and the flash.

In order to change this value, you can:

- Copy `mdriver_max.c` to your specific board directory and then set the value. Be sure to recompile the `libstartup.a` library and relink your startup code with this new library.

  or:

- Set the value in your startup's *main()* before you register the minidriver handler.

Data storage

The minidriver program usually requires a space to store the received hardware data and any other information that the full driver will later need at process time. You need to determine the amount of data you require and allocate the memory.

As we'll see, you allocate the data area in the startup's *main()* function and provide the area's *physical* address when you register the handler function. When the handler is invoked, it's passed the area's *virtual* address.

This area of memory isn't internally managed by the system; it's your driver's responsibility to avoid overwriting system memory. If your handler function writes outside of its data area, a system failure could occur, and the operating system might not boot.

Handler function

The prototype of the minidriver handler function is:

```
int my_handler (int state, void *data);
```

The arguments are:

state

Indicates when the handler is being called. It can have one of the following values, defined in `<sys/syspage.h>`, and presented here in chronological order:

- `MDRIVER_INIT`: The driver is being initialized. The handler is called with this state only once, when you register the handler by calling *mdriver_add()* (p. 32).
- `MDRIVER_STARTUP`: The driver is being called from somewhere in startup.
- `MDRIVER_STARTUP_PREPARE`: Preparations must take place for minidriver operation outside the startup environment.
- `MDRIVER_STARTUP_FINI`: The last call to the driver from within the startup.
- `MDRIVER_KERNEL`: The driver is being called during kernel initialization.
- `MDRIVER_PROCESS`: The driver is being called during process manager/system initialization.
- `MDRIVER_INTR_ATTACH`: A process is calling *InterruptAttach()* or *InterruptAttachEvent()* with the same interrupt number as the minidriver is attached to. If you want the full driver to take over processing the interrupt, the minidriver handler should return 1 to indicate that it wants to exit.

data

The virtual address of the data area, converted from the physical address that you provided as the *data_paddr* parameter to *mdriver_add()*.

If you're working with an ARM platform, your minidriver handler function must be written as Position Independent Code (PIC). This means that when your handler is in the `MDRIVER_KERNEL`, `MDRIVER_PROCESS`, or `MDRIVER_INTR_ATTACH` state, you must not use global or static variables.

The handler function should return:

- 0 if the handler function is still needed
- 1 to request that the kernel remove the minidriver

Don't assume that just because the handler has been called that the device actually needs servicing.

Hardware access

The minidriver program most likely requires hardware access, meaning it needs to read and write hardware registers. In order to help you access hardware registers, the `startup` library provides function calls to map and unmap physical memory. At different times in the boot process, some calls may or may not be available:

- When the minidriver handler is called with `MDRIVER_INIT`, these functions are available:
  - *startup_io_map()*
  - *startup_io_unmap()*
  - *startup_memory_map()*
  - *startup_memory_unmap()*
- After the minidriver handler is called with `MDRIVER_STARTUP_PREPARE`, the above functions are no longer available, and your driver must use these instead:
  - *callout_io_map()*
  - *callout_memory_map()*

For more information, see the Customizing Image Startup Programs chapter of *Building Embedded Systems*.

Here's a summary of what you need to do at each state if your minidriver needs to access hardware:

`MDRIVER_INIT`

- Initialize the hardware, if necessary.
- Call *startup_io_map()* or *startup_memory_map()* to gain hardware access.
- Store this pointer (which we'll call *ptr1*) in the minidriver data area and use it to access hardware.

`MDRIVER_STARTUP`

Use *ptr1* to do all hardware access. No memory map calls are needed.

`MDRIVER_STARTUP_PREPARE`

At this point, The minidriver should call *callout_io_map()* or *callout_memory_map()*, Store the returned pointer (which we'll call *ptr2*) in the minidriver data area or in a static variable, separate from the previously stored value. Don't use *ptr2* yet; continue to use *ptr1* to do all hardware access.

MDRIVER_STARTUP_FINI

This is the last call to your handler from within startup, so it's the last state in which you'll use *ptr1* to do all hardware access. After this state, you'll use *ptr2* instead.

MDRIVER_KERNEL, **MDRIVER_PROCESS**, **MDRIVER_INTR_ATTACH**

In these states, use *ptr2* to do all hardware access.

For an example, see the *Hardware Interaction Within the Minidriver* (p. 37) appendix.

Debugging from within the minidriver

Use the following techniques to debug your minidriver:

- If your startup code is able to print data to a serial port or other debug device, then you can use *kprintf()* to print any variable you wish to see. For example, in your minidriver code:

```
kprintf("I am the minidriver!\n");
kprintf("Global variable mcounts=%d\n", mcounts);
```

  For more information, see the Customizing Image Startup Programs chapter of *Building Embedded Systems*.
- Include any information that you wish to collect in the data area you allocated for your minidriver. After the kernel has booted, you can examine the data inside this area. See the `mini-peeker.c` (p. 27) program for an example of doing this.
- Depending on your hardware, you could use JTAG. If LEDs or other diagnostics are available, your minidriver could output values to hardware registers or ports to indicate certain conditions.

Customizing the `startup` program to include your minidriver

You need to modify the startup code in the BSP's `main.c` file in order to set up the minidriver's data area, register the handler function, and so on. Depending on what your minidriver needs to do, you might have to do the following:

- Declare the prototype for your minidriver's handler function. For example:

  ```
  extern int mini_data(int state, void *data);
  ```

- Call *init_raminfo()*, to determine the location and size of available system RAM.
- Allocate the required system RAM for your data area by calling *alloc_ram()*. For example:

  ```
  /* Global variable: */
  paddr_t mdriver_addr;

  /* Allocate 64 KB of memory for use by the minidriver */

  mdriver_addr = alloc_ram(~0L, 65536, 1);
  ```

- Call *init_intrinfo()* to add the interrupt information to the system page.
- Call *mdriver_add()* (p. 32) to register your minidriver handler. For example:

  ```
  /* Add a minidriver function called "mini-data" for IRQ 81. */

  mdriver_add("mini-data", 81, mini_data, mdrvr_addr, 65536);
  ```

For more information about the startup library functions, see the Customizing Image Startup Programs chapter of *Building Embedded Systems*.

Making the transition to a full driver

Once the kernel is running and interrupts are enabled, the minidriver continues to be called when the interrupt that it's attached to is triggered. This action can continue for the lifetime of the system; in other words, the minidriver can behave like a tiny interrupt handler that's always active.

Usually the hardware needs more attention than the minidriver is set up to give it, so you'll want the minidriver to hand off to a full driver.

Here's the sequence of events for doing this:

- The full driver locates the minidriver's entry in the system page by using the *SYSPAGE_ENTRY()* macro. For an example, see the entry for `mdriver_entry` (p. 34) in this guide.
- The full driver maps the minidriver's data area into its memory space. For example:

  ```
  dptr = mmap_device_memory( 0, 65536,
            PROT_READ | PROT_WRITE | PROT_NOCACHE,
            0, SYSPAGE_ENTRY(mdriver)->data_paddr );
  ```

- The full driver can do post-processing of existing data.

  Since the minidriver is still running at this point, it continues to run whenever the interrupt is triggered. Depending on the design, it may be necessary to do some processing of the existing data that has been stored by the minidriver before the full driver takes control.
- The full driver attaches to the interrupt by calling *InterruptAttach()* or *InterruptAttachEvent()*.

  For safety, the full driver should always disable the device interrupt before calling *InterruptAttach()* or *InterruptAttachEvent()*, and then enable the interrupt upon success.
- When the full driver attaches to the interrupt, the kernel calls the minidriver with a state of `MDRIVER_INTR_ATTACH`. The minidriver should do any cleanup necessary, disable the device interrupt, and then return a value of 1 to request that the kernel remove it.

  After this, the minidriver is no longer called, and only the full driver receives the interrupt.
- The full driver begins to handle the device and process any device data that was stored in the minidriver data area.

Making a boot image that includes your minidriver

At this point, you have a `startup` program (including your minidriver code) that's been compiled. Now include this startup program in the QNX Neutrino boot image and try out the minidriver.

There are some basic rules to follow when building a boot image that includes a minidriver:

- The boot image shouldn't be compressed. Decompression of the boot image affects the timings defined by the *mdriver_max* copy size. Your boot image should have an image type like the following:

  ```
  [virtual=armle-v7,binary] .bootstrap = {
  ```

  Note that the keyword `+compress` isn't included in this line. You should change the `armle-v7` or `binary` entry to reflect your hardware and image format.
- After you compile your startup program that includes the minidriver, make sure to specify this startup program in your buildfile.

  For example, if you compile your startup program as `startup-`*my_board*, you should copy it to the appropriate directory (e.g., `${QNX_TARGET}/armle-v7/boot/sys/startup-`*my_board*`-mdriver`), and then change your buildfile to include `startup-`*my_board*`-mdriver`.

Chapter 3
Sample Minidriver

The minidriver program in this example is a simple implementation that you can use for debugging purposes. It counts the number of times it's called for each phase of the boot process and stores that information in its data area. Once the system is booted, a program can read the data area and retrieve this information. No hardware access is required for this minidriver.

In this example, the size of the data area is 64 KB. If you decrease the value of *mdriver_max* (the amount of data that's copied from flash to RAM between calls of your minidriver) from its default 16 KB, then you may need to increase the size of the data area because the handler function will be called more times.

The minidriver handler function

For this sample driver, the source code for the handler function looks like this:

```
struct mini_data
{
   uint16_t nstartup;
   uint16_t nstartupp;
   uint16_t nstartupf;
   uint16_t nkernel;
   uint16_t nprocess;
   uint16_t data_len;
};

/*
 * Sample minidriver handler function for debug purposes
 *
 * Counts the number of calls for each state and
 * fills the data area with the current handler state
 */
int
mini_data(int state, void *data)
{
   uint8_t  *dptr;
   struct mini_data *mdata;

   mdata = (struct mini_data *) data;
   dptr = (uint8_t *) (mdata + 1);

   /* on MDRIVER_INIT, set up the data area */
   if (state == MDRIVER_INIT)
   {
      mdata->nstartup = 0;
      mdata->nstartupf = 0;
      mdata->nstartupp = 0;
      mdata->nkernel = 0;
      mdata->nprocess = 0;
      mdata->data_len = 0;
   }

   /* count the number of calls we get for each type */
   if (state == MDRIVER_STARTUP)
      mdata->nstartup = mdata->nstartup + 1;
   else if (state == MDRIVER_STARTUP_PREPARE)
      mdata->nstartupp = mdata->nstartupp + 1;
   else if (state == MDRIVER_STARTUP_FINI)
      mdata->nstartupf = mdata->nstartupf + 1;
   else if (state == MDRIVER_KERNEL)
      mdata->nkernel = mdata->nkernel + 1;
   else if (state == MDRIVER_PROCESS)
      mdata->nprocess = mdata->nprocess + 1;
   else if (state == MDRIVER_INTR_ATTACH)
   {
      /* normally disable my interrupt */
      return (1);
   }

   /* put the state information in the data area
   after the structure if we have room */

   if (mdata->data_len < 60000 ) {
      dptr[mdata->data_len] = (uint8_t) state;
      mdata->data_len = mdata->data_len + 1;
   }
```

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

A few things to note:

- The handler function stores call information, so a structure has been created to allow easier access to the data area.
- When the state is `MDRIVER_INIT`, the handler initializes the data area. The handler is called only once with this state.
- Before we store the state information, we make sure that we're not about to write outside the data area, lest we crash the startup.
- If the handler is called with `MDRIVER_INTR_ATTACH`, it returns a value of 1, requesting that the kernel remove the minidriver. However, due to the asynchronous nature of the system, there might be several more invocations of the handler after it has indicated that it wants to stop.

Adding your minidriver to the system

The *main()* function of startup `main.c` looks like this:

```
...
paddr_t  mdrvr_addr;
...

/*
* Collect information on all free RAM in the system.
*/
init_raminfo();

/* In a virtual system we need to initialize the page tables */

if(shdr->flags1 & STARTUP_HDR_FLAGS1_VIRTUAL)
{
   init_mmu();
}

/* The following routines have hardware or system dependencies that
   may need to be changed. */
init_intrinfo();

/* Allocate a 64 KB data area. */
mdrvr_addr = alloc_ram(~0L, 65535, 1);

/* Register the minidriver and its handler function. */
mdriver_add("mini-data",  0,  mini_data, mdrvr_addr, 65535);
...
```

The name stored in the system page for our minidriver is `mini-data`.

Test application: mini-peeker.c

Here's the source code for a test application called mini-peeker.c that maps in the minidriver data area and prints the contents:

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <stdint.h>
#include <sys/mman.h>
#include <sys/neutrino.h>
#include <sys/syspage.h>
#include <hw/inout.h>
#include <inttypes.h>

struct mini_data
{
   uint16_t     nstartup;
   uint16_t     nstartupp;
   uint16_t     nstartupf;
   uint16_t     nkernel;
   uint16_t     nprocess;
   uint16_t     data_len;
};

int main(int argc, char *argv[])
{
   int               i, count;
   int               dump_data = 0;
   uint8_t           *dptr;
   struct mini_data  *mdata;

   if (argv[1])
       dump_data = 1;

   ThreadCtl(_NTO_TCTL_IO, 0);

   /* map in minidriver data area */
   if ((dptr = mmap_device_memory(0, 65535, PROT_READ |
               PROT_WRITE | PROT_NOCACHE, 0,
               SYSPAGE_ENTRY(mdriver)->data_paddr)) == NULL)
   {
        fprintf(stderr, "Unable to get data pointer\n");
        return (-1);
   }

   mdata = (struct mini_data *) dptr;
   dptr = dptr + sizeof(struct mini_data);

   /* dump mini-driver data */
   printf("---------------- MDRIVER DATA -------------------\n");
   printf("\tMDRIVER_STARTUP         calls = %d\n", mdata->nstartup);
   printf("\tMDRIVER_STARTUP_PREPARE calls = %d\n", mdata->nstartupp);
   printf("\tMDRIVER_STARTUP_FINI    calls = %d\n", mdata->nstartupf);
   printf("\tMDRIVER_KERNEL          calls = %d\n", mdata->nkernel);
   printf("\tMDRIVER_PROCESS         calls = %d\n", mdata->nprocess);
   printf("\tData Length             calls = %d\n", mdata->data_len);
   count = mdata->data_len;

   if (dump_data)
   {
       printf("State information:\n");
       for (i = 0; i < count; i++)
            printf("%d\n", dptr[i]);
    }
    printf("\n---------------------------------\n");

   return EXIT_SUCCESS;
}
```

Transition from minidriver to full driver

Here's an example of the code in the full driver that arranges the transition from the minidriver:

```
if ((id == InterruptAttachEvent(intr, event,
              _NTO_INTR_FLAGS_TRK_MSK)) == -1)
{
   perror("InterruptAttachEvent\n");
   return (-1);
}


if ((dptr = mmap_device_memory(0, data_size,
               PROT_READ | PROT_WRITE | PROT_NOCACHE,
               0, SYSPAGE_ENTRY(mdriver)->data_paddr)) == NULL)
{
   fprintf(stderr, "Unable to get data pointer\n");
   return (-1);
}

/* Your minidriver should now be stopped and you should
   have access to the interrupt and the data area */

/* Enable device interrupt (intr) */
```

Once the full driver is attached to the interrupt, it can process any buffered data and continue to provide hardware access.

Chapter 4
APIs and Datatypes

This chapter describes the APIs and datatypes for Instant Device Activation.

mdriver_add()

Register the minidriver with the system

Synopsis:

```
int mdriver_add( char *name,
                 int interrupt,
                 int (*handler)( int state,
                                 void *data ),
                 paddr32_t data_paddr,
                 unsigned data_size);
```

Arguments:

name

An arbitrary character string used for identification purposes.

interrupt

The interrupt that you want to attach the handler to.

handler

A pointer to the handler function for the minidriver. For more information, see "*Handler function* (p. 16)" in the Writing a Minidriver chapter.

data_paddr

The physical address of a block of memory that the minidriver can use to store any data. It can be:

- a predetermined location (e.g., one that you reserved beforehand by passing the `-r` *addr*,*size*[,*flag*] option to startup)
- a block that you allocated by calling the *alloc_ram()* startup function

The virtual address of this block is passed to the handler function as its *data* argument.

data_size

The size of the block of memory denoted by *data_paddr*.

Library:

`libc`

Description:

This function registers the minidriver with the system, as follows:

- It checks the interrupt number to make sure it's valid, so you must call *init_intrinfo()* to add the interrupt information to the system page before you can register a minidriver. For more information, see the Customizing Image Startup Programs chapter of *Building Embedded Systems*.

  If the interrupt number isn't valid, *mdriver_add()* crashes the system.

- It calls the handler with a state of `MDRIVER_INIT` and the virtual address that corresponds to the physical address given by *data_paddr*.
- It adds an entry of type `mdriver_entry` (p. 34) to the *mdriver* section of the system page.

You call *mdriver_add()* from *main()* in your board's startup code.

Returns:

The index into the *mdriver* section of the system page for the newly added minidriver, or -1 if the minidriver wasn't added.

Classification:

QNX Neutrino

mdriver_entry

Minidriver system page entry

Synopsis:

```
struct mdriver_entry
{
   uint32_t    intr;
   int         (*handler)(int state, void *data);
   void        *data;
   paddr32_t   data_paddr;
   uint32_t    data_size;
   uint32_t    name;
   uint32_t    internal;
   uint32_t    spare[1];
};
```

Description:

When you call *mdriver_add()* (p. 32), it adds an entry for your minidriver to the system page. The members of the `mdriver_entry` structure include:

intr

The interrupt that the minidriver is attached to.

handler

A pointer to the minidriver handler function.

data*, *data_paddr

The virtual and physical addresses of the minidriver's data area, respectively.

data_size

The size of the minidriver's data area, in bytes.

name

The offset into the system page's *strings* section where the minidriver's name is stored.

In order for a full (process-time) driver to find a minidriver and gain access to its data area, it must access the entry in the system page by using the *SYSPAGE_ENTRY()* macro:

```
SYSPAGE_ENTRY(mdriver)[i].data_paddr
```

where *i* is the index into the minidriver section. You can use the *name* field to locate a specific minidriver if there are multiple ones running in the system, possibly attached to the same interrupt. Here's some sample code that accesses this information:

```
int i, num_drivers = 0;
struct mdriver_entry    *mdriver;

mdriver = (struct mdriver_entry *) SYSPAGE_ENTRY(mdriver);
num_drivers = _syspage_ptr->mdriver.entry_size/sizeof(*mdriver);
printf("Number of Installed minidrivers = %d\n\n", num_drivers);

for (i = 0; i < num_drivers; i++)
{
    printf("Minidriver entry .. %d\n", i);
    printf("Name .............. %s\n",
       SYSPAGE_ENTRY(strings)->data + mdriver[i].name);
    printf("Interrupt ......... 0x%X\n", mdriver[i].intr);
    printf("Data size ......... %d\n", mdriver[i].data_size);
    printf("\n");
}
```

Classification:

QNX Neutrino

mdriver_max

Amount of data copied from flash to RAM between calls to the minidriver

Synopsis:

```
unsigned mdriver_max = KILO(16);
```

Description:

The *mdriver_max* is a global variable defined in the file `mdriver_max.c` in the startup code. It defines the amount of data (in bytes) that's copied from flash to RAM between calls to your minidriver. The default value is 16 KB, but you might have to change it, depending on the timing requirements of your device, the processor speed, and the flash; see "*Timing requirements* (p. 14)" in the Writing a Minidriver chapter.

The *KILO()* macro, along with *MEG()* and *GIG()*, is defined in `<startup.h>`.

Classification:

QNX Neutrino

Appendix A
Hardware Interaction Within the Minidriver

The following example shows how to interact with hardware from within a minidriver. It's for a fictional hardware device called “MYBUS” with the following characteristics:

- There's a series of 8-bit registers (status and data) at address 0xFF000000 (*MBAR_BASE*).
- Interrupt 56 is generated when a character arrives at the MYBUS port.
- There are registers at this address which will be read from and written to.

Remember that the mapping of hardware registers depends on where in the boot process that the minidriver is called. This transition is handled in the `MDRIVER_STARTUP_PREPARE` and `MDRIVER_STARTUP_FINI` stages.

```
#include "startup.h"      /* This is included with the BSP for your board */

   typedef unsigned char  U8;
   typedef unsigned short U16;
   typedef unsigned int   U32;

/**************    MYBUS Registers   ********************/

typedef struct MYBUS_register_set {
   volatile U8  interrupt_status;
   volatile U8  data_register;
   volatile U8  control_register;
   volatile U8  extra1;
   volatile U8  extra2;
   volatile U8  extra3;
   volatile U8  extra4;
   volatile U8  extra5;
}  MYBUS_regs_t;

/**************    GPIO Registers   ********************/

typedef struct GPIO_register_set {
   volatile U32  gpio0;
   volatile U32  gpio1;
}  GPIO_regs_t;

/**************    Minidriver data area   ************/

typedef struct
{
   MYBUS_regs    *MYBUS_REGS;                    /* This is the same as either
                                                    PREKERNEL or POSTKERNEL. */
   MYBUS_regs    *MYBUS_REGS_PREKERNEL_START;    /* Register mappings to
                                                    use before the kernel starts. */
   MYBUS_regs    *MYBUS_REGS_POSTKERNEL_START;   /* Register mappings to use
                                                    after the kernel starts. */
   U16           total_message_counter;          /* Total times the minihandler
                                                    is called. */
   U16           process_counter;                /* Times called after the kernel
                                                    is running. */
   U16           kernel_counter;                 /* Times called while the
                                                    kernel is booting. */
   U16           data_len;                       /* Length of data portion
                                                    stored in the data area. */
}MYBUS_data_t;


/* Physical memory locations and offsets */
```

```
#define MBAR_BASE 0xff000000
#define GPIO_OFFSET 0x0C00
#define MYBUS_OFFSET 0x2400

/* Control_register settings */
#define CTRL_INTERRUPT_ON       0x01
#define CTRL_INTERRUPT_OFF      0x00

/**********************************************************
void MYBUS_Init(void)

Hardware initialization function for MYBUS.
This routine is called only once, when the minidriver is started.

INPUTS   None

OUTPUTS  None

 **********************************************************/

static MYBUS_regs_t * MYBUS_Init(void)
{
   GPIO_regs_t   *GPIO_REGS_P;
   MYBUS_regs_t    *MYBUS_REGS_P;
   U32 data_byte;

   if((GPIO_REGS_P =
        (GPIO_regs_t *) startup_memory_map(0x40, MBAR_BASE + GPIO_OFFSET),
        PROT_READ|PROT_WRITE|PROT_NOCACHE)) == 0 )
      {
          startup_memory_unmap((unsigned)GPIO_REGS_P);
          return (0);
      }

      /* Change GPIO as needed */
      Data = GPIO_REGS_P->gpio0;
      Data = Data & 0xFFF0FFFF;
      GPIO_REGS_P->gpio0 = Data;

      /* We are done with GPIO */
      startup_memory_unmap((void *)PORT_REGS_P);

      if((MYBUS_REGS_P
         = (MYBUS_regs_t *)startup_memory_map(0x10,
           (MBAR_BASE + MYBUS_OFFSET),
           PROT_READ|PROT_WRITE|PROT_NOCACHE)) == 0
         {
             startup_memory_unmap((unsigned)MYBUS_REGS_P);
             return (0);
         }

        /* Initialize MYBUS and turn on the interrupt. */

        /* Write any values to the MYBUS_REGS_P as needed, and
           then turn on the interrupt source. */

        MYBUS_REGS_P->control_register = CTRL_INTERRUPT_ON;

        kprintf("MYBUS is initialized\n" );
        return ( MYBUS_REGS_P );

}

/**********************************************************
   int mini_mybus_handler(void)
 **********************************************************/

int mini_mybus_handler(int state, void *data)
{
   U8             *dptr;
   U8 StatusReg;
   U8 notValid;
   MYBUS_data_t    *mdata;
   int             val;

   mdata = (MYBUS_data_t *) data;
   dptr = data + sizeof(MYBUS_data_t);

   if (state == MDRIVER_INTR_ATTACH)
   {
```

```
            kprintf("Real driver is attaching .. minidriver was called %d times\n",
                    mdata->total_message_counter);

            /* Disable MYBUS interrupt */
            mdata->MYBUS_REGS_POSTKERNEL->control_register = CTRL_INTERRUPT_OFF;
            return (1);
        }
        else if (state == MDRIVER_INIT)
        {
           /* The first time called, initialize the hardware and do data setup */
           mdata->MYBUS_REGS_PREKERNEL = MYBUS_Init();
           if (mdata->MYBUS_REGS_PREKERNEL == 0)
           {
              return (1);
           }

           /* Make our default register location reflect the fact that we are
              in PREKERNEL */
           mdata->MYBUS_REGS = mdata->MYBUS_REGS_PREKERNEL;

           /* Initialize the counters of messages received. */
           mdata->total_message_counter = 0;
           mdata->process_counter = 0;
           mdata->kernel_counter = 0;
       }
       else if (state == MDRIVER_PROCESS)
       {
           mdata->process_counter++;
       }
       else if (state == MDRIVER_KERNEL)
       {
           mdata->kernel_counter++;
       }
       else if (state == MDRIVER_STARTUP_PREPARE)
       {
           /* Once we are out of startup, use callout_io_map or callout_memory_map */

           kprintf("I am in STARTUP PREPARE %x\n", mdata->total_message_counter);
           if ((mdata->MYBUS_REGS_POSTKERNEL = (MYBUS_regs_t *)(callout_memory_map(0x10,
                     (MBAR_BASE + MYBUS_OFFSET),
                     PROT_READ|PROT_WRITE|PROT_NOCACHE))))
           {
              /* Something bad happened. Disable the interrupt and turn off
                 the minidriver */
              mdata->MYBUS_REGS_PREKERNEL->control_register = CTRL_INTERRUPT_OFF;
              return (1);
           }

        }

        /* At this point, we use MYBUS_REGS. We could either be in startup, in
           kernel loading or at process time. */

        /* Read the interrupt status register immediately upon entry to the handler. */
        StatusReg = mdata->MYBUS_REGS->interrupt_status;

        /* Increase the message counter. */
        mdata->total_message_counter++;

        switch( StatusReg )
        {
           /* Read my data and add to my data area (after data_len in MYBUS_data_t) *.

           /* Make sure that you clear the source of interrupt before you return *.

           /* ... */
        }

        if (state == MDRIVER_STARTUP_FINI)
        {
           val = mdata->total_message_counter;
           kprintf("I am in state STARTUP FINI. Total messages processed=%x \n", val);

           /* Startup has finished.. now I switch over to use the POSTKERNEL mapping */
           mdata->MYBUS_REGS = mdata->MYBUS_REGS_POSTKERNEL;
        }
        return (0);
   }
```

Index