QNX® Neutrino® RTOS

Instant Device Activation
User's Guide

For QNX® Neutrino® 6.3.0 SP1 or later

Published under license by:

QNX Software Systems International Corporation
175 Terence Matthews Crescent
Kanata, Ontario
K2M 1W8
Canada
Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: `info@qnx.com`
Web: `http://www.qnx.com/`

Publishing history

Electronic edition published 2007

Contents

List of Figures

About This Guide

What you'll find in this guide

The Instant Device Activation *User's Guide* will help you set up a "minidriver" to start devices quickly when the system boots.
This guide is intended for software developers who want to develop instant device activation code.

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

For information on:	Go to:
How to write instant device activation code	Using Minidrivers for Instant Device Activation
API and datatypes	API and Datatypes
Sample drivers for writing code for different platform	Sample Drivers for Instant Device Activation
Hardware interaction within the minidriver	Hardware Interaction within the Minidriver

Typographical conventions

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

Reference	Example
Code examples	`if( stream == NULL )`
Command options	`-lR`
Commands	**`make`**
Environment variables	**PATH**
File and pathnames	`/dev/null`
Function names	*exit()*
Keyboard chords	Ctrl-Alt-Delete
Keyboard input	**`something you type`**
Keyboard keys	Enter
Program output	**`login:`**
Programming constants	NULL
Programming data types	**`unsigned short`**
Programming literals	`0xFF`, **`"message string"`**

continued...

Reference	Example
Variable names	*stdin*
User-interface components	**Cancel**

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.

Technical support

To obtain technical support for any QNX product, visit the **Support + Services** 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

In this chapter. . .

Instant Device Activation (also known as minidriver) allows you to provide higher levels of integration for your hardware. Advanced CPUs are providing higher levels of hardware integration than ever before. For example, the main CPU can now directly control CAN, J1850, and MOST interfaces.

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 is powered on. The problem is, the complex software running on such a telematics device can easily take hundreds of milliseconds, or more, to boot up.

For 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:

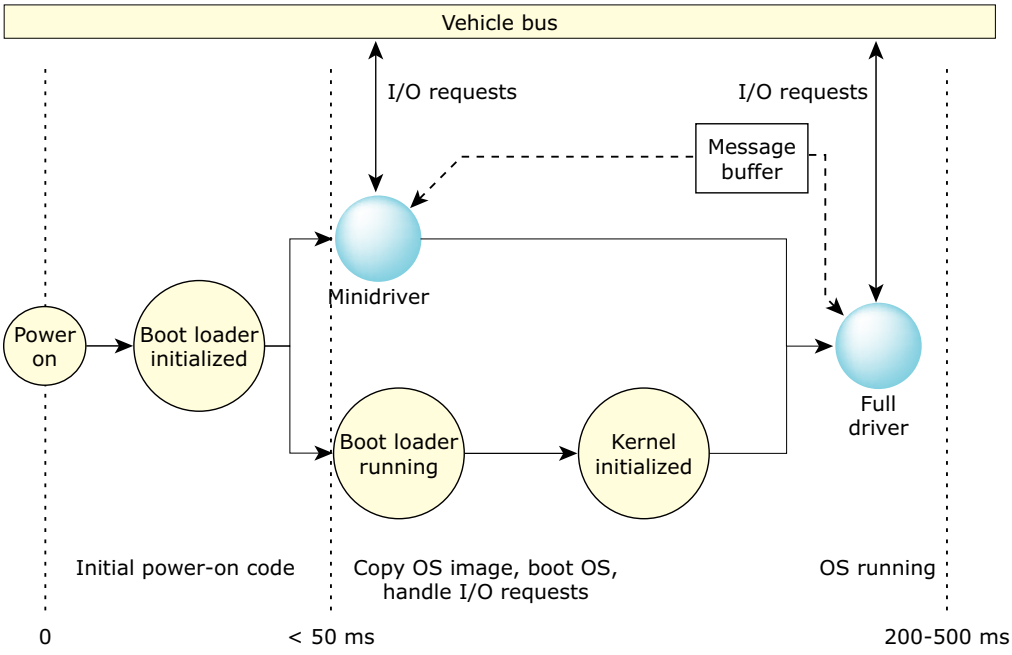
- receive CAN messages within 30 to 100 milliseconds after power is turned on
- respond to the above messages within 100 milliseconds of receiving them
- read Class 2 messages from a vehicle bus and responding to wake events
- initialize a MOST transceiver and responding to MOST requests
- animate a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

In order to address the timing requirements described above, 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 due to innovative software from QNX Software Systems. 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 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.



Booting process using instant device activation.

The minidriver architecture

A minidriver consists of two fundamental components:

- handler function
- message data storage

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.

How does the minidriver work?

A minidriver is a function (piece of code) that you link to the Neutrino startup program, so that it runs Run this code 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 (that is added to the Neutrino startup) 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 will be available and this handler function will be interrupt driven. The handler is called with a *state* variable.

See below for a prototype of this minidriver handler function:

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

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 will continue to run. The full driver also 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 will be polled (i.e. not invoked at a constant interval) during startup, the driver will need to use something to measure the time between the calls to get the proper interval.

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

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
- Whether or not the minidriver requires data storage and how much it 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

Hardware platform

Get the BSP associated with your hardware platform; it includes the source code to the board's **startup** program. You must link your minidriver to this **startup** program.

For more information, see the BSP documentation.

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

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 states, you must follow steps below:

- use global variables
- use static variables.

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 **startup** contains a global variable *mdriver_max*, which is the size of data (in bytes) that is copied from flash to RAM between calls of your minidriver. The default size is 16 KB. The appropriate data size should be based on the timing requirements of your device, processor speed, and the flash. The file that contains this variable is called **mdriver_max.c** and can be found in the startup library.

In order to change this value, there are two options:

- You can copy this file to your specific board directory and then the value. The **mdriver_max.c** file contains the following variable:

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

- Modify the **startup** program to initialize this variable to a new value, possibly in the *main()* function for your particular startup. Since this is a global variable, you can initialize it to a new value in *main()* before the minidriver handler is added.

Data storage

The minidriver program requires a space to buffer the received hardware data that is later passed to the full driver at process time. You will need to determine the amount of data you require and allocate the memory.

You have the choice of specifying where the memory is located or having startup choose a safe location. In order to allocate the memory, you make the following function call:

```
paddr_t alloc_ram(phys_addr, size, alignment);
```

The address returned is the physical address of your memory area. This is used when you register your minidriver with the **startup** program. Since you are reserving an area of RAM, make sure you call this function after RAM has been setup, i.e. after calling *init_raminfo()*.

This area of memory isn't internally managed by the system, it's your drivers responsibility to make sure it doesn't overwrite system memory and cause the startup to crash.

Hardware initialization

If your driver requires hardware initialization, you should place code in the MDRIVER_INIT handler of the minidriver. The MDRIVER_INIT state is the first state of the minidriver and it's set only once.

Hardware Access

The minidriver program most likely requires hardware access, meaning it needs to read and write hardware registers. In order to access hardware registers, the **startup** library provides function calls to map and unmap physical memory.

When the minidriver handler is called with MDRIVER_STARTUP_INIT, you call:

```
uintptr_t startup_io_map(size, phys_addr);
startup_io_unmap(paddr);
void * startup_memory_map(size, phys_addr);
startup_memory_unmap(paddr);
```

After the minidriver handler is called with MDRIVER_STARTUP_PREPARE, the above functions are no longer available and your driver must use:

```
uintptr_t callout_io_map(size, phys_addr);
void * callout_memory_map(size, phys_addr);
```

At different times in the boot process, some calls may or may not be available. If your driver requires hardware access, it must do the following:

MDRIVER_INIT

- Call *startup_io_map()* or *startup_memory_map()* to gain hardware access
- Store this pointer away in the minidriver data area and use it to access hardware (*ptr1*).

MDRIVER_STARTUP

Use the stored pointer, *ptr1*, to do all hardware access. No memory map calls are needed.

MDRIVER_STARTUP_PREPARE

At this point, it's safe to call *callout_io_map()* or *callout_memory_map()*, but don't use the pointer returned.

The minidriver should call one of the above functions and store the pointer in the minidriver data area or in a static variable, separate from the previously stored value (*ptr2*). Use the stored pointer, *ptr1*, to do all hardware access.

MDRIVER_STARTUP_FINI

Use the stored pointer, *ptr1*, to do all hardware access. Any further calls will require the minidriver to use the pointer mapped with *callout_io_map()* or *callout_memory_map()* mapped pointer, so your driver must maintain this information in the data area.

MDRIVER_KERNEL

Use the stored pointer, *ptr2*, to do all hardware access.

MDRIVER_PROCESS

Use the stored pointer, *ptr2*, to do all hardware access.

MDRIVER_INTR_ATTACH

Use the stored pointer, *ptr2*, to do all hardware access.

Transition to full driver

Once the kernel is running, your full driver process can run a transition from the minidriver to the full driver. The full driver can take control of the hardware device and the minidriver can then gracefully exit. In order to perform the transition, the full driver does the following:

- locates the minidriver entry in the system page
- maps the minidriver data area into its memory space
- attaches to the device's interrupt.

The minidriver is called with a state of MDRIVER_INTR_ATTACH**>**. At this point, the minidriver should do any cleanup necessary and disable the device interrupt. The minidriver handler then can return a nonzero value, which indicates that it should exit. The successful transition occurs when:

- the full driver's interrupt attach succeeds and the full driver reenables the device interrupt
- the full driver begins to handle the device and any stored device data is read from the minidriver data area.

Sample minidriver

The sample minidriver program in this example is a simple implementation that can be used for debug purposes. In this implementation, the minidriver counts the number of

times it is called for each phase of the boot process and store that information in its data area. Once the system is booted, a program can then read this data area and retrieve this information.

Implementation notes

Timing data will be stored in a shared memory area. In this example, the size of this memory is set to 64 KB. If you decrease the *mdriver_max* value from 16 KB to a lower value, then you may need to increase this 64 KB value (e.g. to 128 KB). This is due to a larger number of callouts made when *mdriver_max* is decreased.

You should use the default *mdriver_max* value of 16 KB. So, the data storage required will be 64 KB. The assumption here is that there is no hardware access required for this implementation.

The minidriver handler function

The prototype for the minidriver handler function is as follows:

int *mdriver_handler*(**int** *state*, **void** ***data**);

See the description for *mdriver_add()* for the definition of *state* and the *data* pointer variable.

For this sample driver, the source code for the handler function would look 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);
}
```

In this example, the handler function stores call information, so a structure has been created to allow easier access to the data area.

Since the data area is set as 64 KB, we ensure that we don't write any data outside this area. If your handler function writes outside of its data space, a system failure can occur, and the operating system may not boot. Always be sure to properly bound memory reads and writes.

During the MDRIVER_INIT state, the data area is initialized. this is only called once. If your handler is called with MDRIVER_INTR_ATTACH, it returns a value of 1, requesting an exit of the handler. 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

Once you have written a handler function, you need to register it with startup and allocate the required system RAM for your data area. This can be accomplished with the following functions:

```
paddr_t alloc_ram(phys_addr, size, alignment);
int mdriver_add(name, interrupt, handler, data_paddr, data_size);
```

Since you're allocating memory and passing an interrupt, these functions must be called after RAM is initialized by calling *init_raminfo()*, and after the interrupt information is added to the system page by calling *init_intrinfo()*.

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

```
...
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 which
* may need to be changed.
*/
init_intrinfo();
mdrvr_addr = alloc_ram(~0L, 65535, 1); /* make our 64 k  data area */
mdriver_add("mini-data",  0,  mini_data, mdrvr_addr, 65535);
...
```

In this example, we have allocated 64 KB of memory and registered our minidriver handler function with the name **mini-data**.

Build startup

Once your minidriver is complete, you must rebuild startup code and the boot image. In order to verify that the minidriver is running properly, you may want to add debug information in the handler function or write an application to read the minidriver data area and print the contents.

Test application: `mini-peeker.c`

Below is 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;
}
```

Compile this code for your target system using regular compile technique. For example:

```
qcc -Vgcc_ntoppcbe mini-peeker.c -o mini-peeker
```

Transition from minidriver to full driver

Once the system is booted and your full driver is running, a transition must take place where the full driver takes over from the minidriver. The full driver should attach to the minidriver's interrupt and then it should read the minidriver's data area in order to retrieve any stored information.

Once your full driver does an *InterruptAttach()* or *InterruptAttachEvent()*, the kernel calls the minidriver with a state value of MDRIVER_INTR_ATTACH. When your minidriver handler receives this state, it should do any cleanup necessary and then exit.

Once your full driver is attached to the interrupt it can deal with any buffered data and continue to provide hardware access. A sample of this code would look like:

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

For safety, your full driver should always disable the device interrupt before doing the *InterruptAttach[Event]()* and then enable the interrupt upon success.

Minidriver implementation notes

Let's examine in more detail the steps for developing, running and debugging a minidriver.

These are the main phases when developing a minidriver:

- customizing the **`startup`** program that contains your minidriver code
- making a boot image that includes your minidriver
- debugging from within the minidriver
- displaying information about minidriver status (after the kernel boots)
- making the transition to the real (full) driver

Note that the last phase "making the transition to the real (full) driver" is optional. Until you turn off the minidriver by attaching an interrupt handler (*InterruptAttach*()*), your handler continues to receive interrupts and can be called even after the Neutrino system is booted and running. This may be a desired design.

Let's look at each of the phases of development and some tips and techniques for each phase.

Customizing the `startup` program that contains your minidriver code

This is the core of the minidriver development.

To implement a minidriver, the following steps are required:

1 Modify **`mdriver_max.c`** in **`libstartup`**.

(On your development host, this file is located in your workspace as part of your BSP files in a directory that ends with **`libstartup`**).

By default, this value is set to 16KB (with a standard QNX 6.3.0 Board Support Package) and is the amount of data that is copied at one time when the boot image is copied from flash to RAM. The minidriver callout will be called after each copy. For example:

```
minidriver callout
copy next 16K
minidriver callout
copy next 16K
etc.
```

It may be necessary to modify this value. For example, on a MPC5200 running at 396 MHz, the time needed to copy 16KB is around 8 milliseconds. This will vary depending on the speed of the processor and the speed of the flash. If the *mdriver_max* is modified to be a 1KB copy value, then the time needed to copy this 1KB drops to less than 1 millisecond. The value of *mdriver_max* will need to be set based on experimentation.

If **mdriver_max.c** is modified, be sure to recompile the **libstartup.a** library. Also, relink your startup code with this new library.

2 Modify the necessary startup files and add in your minidriver code.

The following files are modified in your startup code. They all exist in the same directory as your BSP startup code. For example, if you are building a BSP for the Media5200b board, you will have imported the BSP into QNX Momentics IDE (in to your workspace). The following files will change:

```
main.c        // to call mdriver_add() for your callout

cpu_mdriver.c // This file is added to your system.
                           // QNX provides this. Don't change the file.

mdriver.c     // This file is added to your system.
                           // QNX provides this. Don't change the file.

mini-driver.c // The rest of the code is your mini-driver code
                           // including the main handler function.
```

See the example files in the Sample Drivers for Instant Device Activation chapter of this guide.

Here are the highlights:

main.c

- Set the prototype for your minidriver callout function e.g.

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

- Allocate a memory area for your data e.g.

```
//Global
paddr_t mdriver_addr; // allocate 64K of memory
                      // for use by the minidriver

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

- Set your minidriver callout to be called. Do this after you call *init_intrinfo()* e.g.

```
//Code to add a sample minidriver function
//called "mini-data" for irq=81
mdriver_add("mini-data", 81, mini_data, mdrvr_addr, 65536);
```

cpu_mdriver.c
- Don't modify this file unless directed by QNX Software Systems.
- Make sure this C file is included in your startup code directory

mdriver.c
- Don't modify this file unless directed by QNX Software Systems.
- Make sure this **c** file is included in your startup code directory.

mini-driver.c
- This is your minidriver code. You can name this C file anything you wish. What is required is that the minidriver function defined in the *mdriver_add()* function be part of your startup code.
- You may have multiple C and header files as part of your minidriver

Try out one of the samples that is included with this package and build the new startup program (e.g. **startup-mgt5200**).

Making a boot image that includes your minidriver

Until this point, you will have a **startup** program (including your minidriver code) that has been compiled. Now include this startup program in the Neutrino boot image and try out the minidriver.

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

Compression

The boot image shouldn't be compressed. Decompression of the boot image modify the timings defined by the *mdriver_max()* copy size. Your boot image should have the following image type defined:

```
[virtual=ppcbe,binary] .bootstrap = {
```

Note that the keyword **+compress** isn't included in this line. You should change the **ppcbe** or binary entry to reflect your hardware and image format.

Startup program

After you compile your startup program that includes the minidriver, make sure to specify this startup program in your build file, e.g.

For the Media5200b board,

if you compile your startup program as **startup-mgt5200**, then you can do the following:

```
copy startup-mgt5200 to $(QNX_TARGET)/ppcbe/boot/sys/startup-mgt5200-mdriver
change my build file to include this startup-mgt5200-mdriver program
```

See the Sample Drivers for Instant Device Activation chapter of this guide for more information and sample build files.

Debugging from within the minidriver

Use the following techniques to debug your minidriver:

- *kprintf()*

  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, e.g. in your minidriver code:

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

- Memory area

  Include any data that you wish to collect in the shared memory area allocated for your minidriver. After the kernel has booted, you can examine the data inside the shared memory area. See the **`mini-peeker.c`** program for an example of doing this.

- JTAG or hardware indication

  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.

Displaying information about minidriver status (after the kernel boots)

After the kernel has booted, examine the shared memory that you allocated for your minidriver, by calling *alloc_ram()*. Pass the **`paddr_t`** to the *mdriver_add()* function so that there is a link between your minidriver and the shared memory area.

See the example **`mini-peeker.c`**.

Making the transition to the real driver

The minidriver is called based on the following sequences:

- When first attached (with *mdriver_add()*), the minidriver function is polled. You can change the poll rate can be changed by setting the value in *mdriver_max()* in **`libstartup`**.
- Once the kernel is running and interrupts are enabled, the minidriver is called when the interrupt that it is attached to is triggered.
- This action can continue for the life of the system. In other words, the minidriver can behave like a tiny interrupt handler that is always active.

  Usually, there is a need to do more with the hardware than what the minidriver is set up to do. There will be a transition or handoff to the real driver that will attach to the real interrupt.

  To do this properly, here are the sequences of events that should be done:

- The real driver starts and attaches to shared memory area.

  The real driver attaches to the shared memory area that is owned by the minidriver. For example:

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

- the real 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 until the real driver attaches to the same interrupt. 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 real driver takes control.

- the real driver attaches to the interrupt

  The real driver calls the function *InterruptAttach()* or *InterruptAttachEvent()* (the latter is preferred over *InterruptAttach()*). When this call is made, the minidriver receives a message of type MDRIVER_INTR_ATTACH. The minidriver returns a value of 1.

  From now onwards, the minidriver will no longer be called on the interrupt. Only the real driver will receive the interrupt. The shared memory area will continue to exist until you remove it.

Chapter 2

APIs and Datatypes

This chapter contains the following APIs and datatypes for the Instant Device Activation:

- *mdriver_add()*
- *mdriver_handler()*
- *mdriver_max*
- **`mdriver_entry`**

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 same value that is passed into *InterruptAttach()* to attach to a particular interrupt. It's first checked to make sure it's a valid number, so you must do an *init_intrinfo()* to add the interrupt information to the system page before you can register a minidriver. (See the Customizing Image Startup Programs chapter of Building Embedded Systems).
handler	This minidriver function is called at various times for the user to check the device. Do not assume that just because the handler has been called that the device actually needs servicing.
data_paddr	A physical address of a block of memory that is made available to the minidriver. It can be a predetermined location, or the user could use the *alloc_ram()* startup function to obtain the memory.
data_size	The size of the block of memory denoted by *data_paddr*.

Library:

libc

Description:

This function registers the minidriver with the system.

mdriver_handler()

It's the prototype of your handler function. This function registers the minidriver with the system. The prototype is:

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

The arguments are:

state Indicates when the handler is being called. It can have one of the following values:

MDRIVER_INIT	The driver is being initialized, called from *mdriver_add()*. The handler is called with this state only once.

MDRIVER_STARTUP The driver is being called from somewhere in startup.

MDRIVER_STARTUP_PREPARE
Preparations must take place for minidriver operation outside of 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. The *InterruptAttachEvent()* with the same interrupt number as the minidriver.

data A pointer that points to virtual address of the block of data provided as the *data_paddr* parameter of *mdriver_add()*.

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:

Safety	
Cancellation point	Yes
Interrupt handler	No
Signal handler	No
Thread	Yes

Synopsis:

```
unsigned mdriver_max;
```

Description:

The *mdriver_max* is a global variable in startup in the file **mdriver_max.c**. Since most of the time spent in startup is copying the OS image from flash to RAM, this value is the amount of data (in bytes) to copy between iterations of your minidriver handler function. You may need to modify this value based on the timing requirements of your driver.

The entry looks like:

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

Classification:

QNX Neutrino

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:

Each entry for a minidriver is stored in the system page. In order for a full (process time) driver to find a minidriver and gain access to its data area, it must access the system page. The members of this structure are:

intr — The interrupt, which the minidriver is attached to.

handler — The minidriver handler function.

data — The physical address of the minidriver's data area.

data_paddr

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

name — The name of the minidriver, added by *mdriver_add()*. This value is the offset into the system page strings section where the name is stored.

Access to this information is done through 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 can be used to locate a specific minidriver if there are multiple ones running in the system, possibly attached to the same interrupt. Sample code used to access this information would look like:

```
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

Chapter 3

Sample Drivers for Instant Device Activation

In this chapter. . .

FreeScale Media5200b sample minidriver

This sample is a minidriver for the Media5200b board. The driver initializes the serial port at a known baud rate and buffers the characters into the data area. Once the system is booted, a program can then read this data area and retrieve the buffered characters. For this implementation, we require:

- The value of *mdriver_max* is 1 KB bytes for 14400 baud rate. Note that timing is dependent on the baud rate.
- Storage is required and 64 KB bytes will be used.
- Hardware access to the serial port registers is required.

The minidriver handler function

The prototype for the minidriver handler function is as follows:

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

The *state* value informs the handler function where in the boot process it's being called from. See the *mdriver_add()* documentation in the API and Datatypes chapter of this guide. The *data* parameter is a virtual pointer to the minidriver's allocated data area. For this sample driver, the source code for the handler function would look like:

```
struct mserial_data
{
   psc_regs *psc;
   psc_regs *psc_k;
   uint16_t intr_calls;
   uint16_t ncalls;
   uint16_t errors;
   uint16_t last_count;
   uint16_t  data_len;
};

/*
 * Hardware initialization function which only gets called
 * the first time the mini-driver is run.
 * The Baud, Clk and port are hardcoded for this example
 *   baud = 14400
 *   clk = 132000000
 *   port = PSC1
 */
static psc_regs *
init_hw()
{
   /* hardcode to PSC1, baud and clk */
   int  baud = 14400;
   int  clk = 132000000;
   paddr64_t psc_base = MGT5200_MBAR_BASE + 0x2000;
   psc_regs *psc;
   gpio_std_regs *gpio;

   if ((psc = (psc_regs *)(startup_memory_map(0xA0,psc_base,
    PROT_READ|PROT_WRITE|PROT_NOCACHE))) == 0)
       return (0);

   if ((gpio = (gpio_std_regs*)(startup_memory_map(0x40,(MGT5200_MBAR_BASE + 0x0b00),
    PROT_READ|PROT_WRITE|PROT_NOCACHE))) == 0)
   {
      startup_memory_unmap((void *)psc);
      return (0);
   }
```

```
   baud = ( (clk / baud) + 16 ) / 32 ; /* calculate baud rate */
   psc->CR = PSC_CR_TXD | PSC_CR_RXD;
   psc->SICR = 0x00000000;         /* write to SICR: SIM2 = uart mode,dcd
                                   does not affect rx */
   psc->SR_CSR = 0xdd00;           /* write to CSR: RX/TX baud rate
                                   from timers */
   psc->CTUR = (baud >> 8) & 0xff; /* write to CTUR: divide counter
                                   upper byte */
   psc->CTLR = baud & 0xff;        /* write to CTLR: divide counter
                                   lower byte */
   psc->CR = 0x20;                 /* write to CR: reset RX and
                                   RXFIFO */
   psc->CR = 0x30;                 /* write to CR: reset TX and
                                   TXFIFO */
   psc->CR = 0x40;                 /* write to CR: reset
                                   error status */
   psc->CR = 0x50;                 /* write to CR: reset
                                   break change int */
   psc->CR = 0x10;                 /* write to CR: reset MR pointer */
   psc->IPCR_ACR = 0x00;           /* write to ACR: disable state
                                   change CTS/ DCD interrupts */
   psc->ModeReg = 0x73;            /* write to MR1: 8bit data,
                                   no parity */
   psc->ModeReg = 0x07;            /* write to MR2: normal
                                   mode,1stop bit */
   psc->OP1 = 1;                   /* Write to OP1: enable
                                   RTS to send */
   psc->ISR_IMR = 0x0000;          /* write to IMR: Mask RxRDY and
                                   TxRDY from causing an interrupt */
   psc->RFALARM = 511;
   psc->TFALARM = 0xF8;
   psc->RFCNTL = 0x04;

   gpio->port_config = (gpio->port_config & GPIO_PSC1_MASK) |
                       GPIO_PSC1_UART_ENABLE;

   /* Enable the transmitters and receivers */
   psc->CR = 0x05; /* write to CR: enable TX and RX.*/
   psc->ISR_IMR = PSC_ISR_RxRDY;

   startup_memory_unmap((void *)gpio);

   /* return mapped register pointer */
   return (psc);
}

int mini_serial(int state, void *data)
{
   uint8_t     *bp = 0;
   paddr64_t psc_base = MGT5200_MBAR_BASE + 0x2000;
   uint16_t count;
   uint8_t *dptr;
   struct mserial_data *mdata;

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

   if (state == MDRIVER_INTR_ATTACH)
   {
  /* disable the serial interrupt */
  mdata->psc->ISR_IMR = 0x0000;
  * make sure there are no characters in the hardware fifo */
  count = 0;
  while (mdata->psc->SR_CSR & PSC_SR_RxRDY)
  {
      bp = (uint8_t*) &mdata->psc->RB_TB;
      dptr[mdata->data_len+count] = *bp;
      count++;
  }
      mdata->last_count = count;
      count = count + mdata->data_len;
        mdata->data_len = count;

      /* reset any error conditions */
      if ((mdata->psc->SR_CSR & PSC_SR_ORERR) ||
```

```
          (mdata->psc->SR_CSR & PSC_SR_RB) ||
         (mdata->psc->SR_CSR & PSC_SR_FE) ||
    (mdata->psc->SR_CSR & PSC_SR_PE))
      {
    mdata->psc->CR = PSC_CR_RESET_ERR;
    count = mdata->errors + 1;
    mdata->errors = count;
      }

      return (1);
    }
    else if (state == MDRIVER_INIT)
    {
       /* the first time called initialize the hardware and do data setup */
       mdata->psc = init_hw();
       if (mdata->psc == 0)
  return (1);
       mdata->intr_calls = 0;
       mdata->ncalls = 0;
       mdata->errors = 0;
       mdata->data_len = 0;
    }
    else if (state == MDRIVER_STARTUP_PREPARE)
    {
       /* once we are out of startup use callout_io_map */
       if ((mdata->psc_k = (psc_regs *)(callout_memory_map(0xA0,psc_base,
        PROT_READ|PROT_WRITE|PROT_NOCACHE))) == 0)
       {
   /* something bad so disable the driver */
   mdata->psc->ISR_IMR = 0x0000;
   return (1);
   }
       }

    /* count the number of times the mini-driver is called */
    count = mdata->ncalls + 1;
    mdata->ncalls = count;
    if (state == MDRIVER_PROCESS)
    {
/* called because of an interrupt */
count = mdata->intr_calls + 1;
mdata->intr_calls = count;
    }

    count = 0;
    while (mdata->psc->SR_CSR & PSC_SR_RxRDY)
    {
bp = (uint8_t*) &mdata->psc->RB_TB;
     dptr[mdata->data_len+count] = *bp;
count++;
     }

     mdata->last_count = count;
     count = count + mdata->data_len;
     mdata->data_len = count;

     /* reset any error conditions */
     if ((mdata->psc->SR_CSR & PSC_SR_ORERR) ||
       (mdata->psc->SR_CSR & PSC_SR_RB) ||
      (mdata->psc->SR_CSR & PSC_SR_FE) ||
 (mdata->psc->SR_CSR & PSC_SR_PE))
     {
 mdata->psc->CR = PSC_CR_RESET_ERR;
 count = mdata->errors + 1;
 mdata->errors = count;
     }

     if (state == MDRIVER_STARTUP_FINI)
mdata->psc = mdata->psc_k;

     return (0);
}
```

In this example, the handler function stores call information, so a structure has been created to allow easier access to the data area. The data area is filled with the received characters.

Access to the serial port registers is necessary. At initialization time, the registers are mapped with *startup_memory_map()*, and the pointer is stored in the data area during the startup phase. The registers are mapped in with *startup_memory_map()*. Once the startup phase is complete, the handler must use *callout_memory_map()* to access the registers.

During the MDRIVER_INIT state, the data area is initialized, and the serial hardware is set up; this is called only once. At this time, the *startup_memory_map()* is called to map in the hardware registers, and the pointer is stored in the data area. This pointer is used for hardware access until the handler is called with a state of STARTUP_MDRIVER_PREPARE. At this time *callout_memory_map()* can be called and the pointer is stored in the data area, however the *startup_memory_map()* pointer should still be used. Once the handler is called with a state of MDRIVER_STARTUP_FINI, the handler starts using the pointer returned by *callout_memory_map()* for all hardware access. This pointer is then used for all further invocations of the minidriver handler.

When the handler is called with a state of MDRIVER_INTR_ATTACH, it disables the device interrupt and returns a value of 1 requesting an exit of the handler.

Adding your minidriver to the system

Once you have written a handler function, you need to register it with startup and allocate the required system RAM for your data area. This can be accomplished with the following functions:

```
paddr_t alloc_ram(phys_addr, size, alignment);
int mdriver_add(name, interrupt, handler, data_paddr, data_size);
```

Since we are allocating memory and passing an interrupt these functions must be called after RAM is initialized by calling *init_raminfo()*, and after the interrupt information is added to the system page by calling *init_intrinfo()*. The *main()* function of our startup **`main.c`** will look 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 which
* may need to be changed.
```

```
*/
init_intrinfo();

mdrvr_addr = alloc_ram(~0L, 65536, 1); /* grab 64k */
mdriver_add("mini-serial",  0,  mini_serial, mdrvr_addr, 65536);
...
```

In this example, we have allocated 64 KB of memory and registered our minidriver handler function with the name **mini-serial**.

Build startup

The minidriver is complete now, and you must rebuild startup and boot image. In order to verify that the minidriver is running properly, you may want to add debug information into the handler function or write an application to read the minidriver data area and print the contents.

Testing your minidriver

The Media5200b board has a single serial port that this minidriver example uses to receive bytes on. In order to test this driver, you should build an OS image that sets up TCP/IP and lets you **telnet** to the board. Since you'll be using the serial port as a data source; don't run the serial driver and don't put debug printouts or verbosity on **procnto** in your image. Once you have an image, you should burn it onto the onboard flash where you can boot from it.

Connect a NULL-modem cable between your Media5200b board and a host machine. The minidriver is running at 14400 baud, with no flow control. You must make sure that the host machine's serial port is also configured in this manner. On the host machine, begin sending characters to the serial port. Sample code should look like:

```
...
uint8_t test_data[20] = {"012345678987654321"};
...

if ((fd = open("/dev/ser1", O_RDWR)) == -1)
{
   fprintf(stderr, "Unable to open device: %s (errno=%d\n", device, errno);
   return (-1);
}

for (;;)
{
write(fd, test_data + index, 1);
   index++;
   if (index == 19)
   index = 0;
}
```

This code sends a pattern of characters to the serial port so that the minidriver can capture data.

Now boot the Media5200b board and **telnet** to it.

Once connected, you can run the sample full driver, **mdrvr-serpsc** that does the following:

- attaches to the serial interrupt

- stops the minidriver
- reads the minidriver data area
- captures 64 characters from the serial port
- displays the minidriver data and the captured data, which should show the pattern of characters your sample application sent

Since the **`mdrvr-serpsc`** application is a sample serial driver you don't need to run **`devc-serpsc`** driver.

Sample timings

These timings are based on sample boot image with the following properties:

- boot from Flash
- uncompressed image size: 904092 bytes
- example minidriver running, data driver only

mdriver_max	Number of calls	Average calling interval
1 KB	855	514 microseconds
4 KB	231	2.04 ms
16 KB	75	8.13 ms

Boot time	First invocation time
453.2 ms	1 ms

Boot time identifies the time from startup to the start of the first application process. The first invocation time is the time from IPL to the first invocation of the minidriver (MDRIVER_INIT).

Renesas Biscayne minidriver

This sample is a minidriver for the Biscayne board. The driver initializes the serial port at a known baud rate and buffers the characters into the data area. Once the system is booted, a program can then read this data area and retrieve the buffered characters.

For this implementation we require:

- the *mdriver_max* value of 4KB
- data storage is required and 64KB is assumed

- hardware access to the serial port registers

The minidriver handler function

The prototype for the minidriver handler function is as follows:

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

The *state* value informs the handler function where in the boot process it's being called from. See the *mdriver_add()* documentation in the API and Datatypes chapter of this guide. The *data* parameter is a virtual pointer to the minidriver's allocated data area. For this sample driver, the source code for the handler function would look like this:

```
struct mserial_data
{
   uintptr_t port;
   uintptr_t port_k;
   uint16_t intr_calls;
   uint16_t ncalls;
   uint16_t errors;
   uint16_t err;
   uint16_t last_count;
   uint16_t  data_len;
};


void Delay(int n)
{
   unsigned int i;
   for(i=0;i<n;i++);
}

uintptr_t
init_hw()
{
   int  cnt;
   uint64_t base = SH_SCIF_BASE;
   int baud = 14400;
   int clk = 33333333;
   unsigned port;

   if ((port = startup_io_map(120, base)) == NULL)
       return (NULL);

    cnt = clk / (baud * 64 / 2) - 1;  // assumes SCSMR1.CKS = 0

    // set clock selection
    out16(port + SH_SCIF_SCSCR_OFF, 0x0);
    Delay(1);

    // Enable reset state of the FIFO register
    out16(port + SH_SCIF_SCFCR_OFF, SH_SCIF_SCFCR_TFRST | SH_SCIF_SCFCR_RFRST);

    // data transfer format
    // 8 bits no parity
    // CKS = 0
    out16(port + SH_SCIF_SCSMR_OFF,0);
    Delay(1);

    // baud rate
    out8(port + SH_SCIF_SCBRR_OFF,cnt);
    Delay(1);

    //Disable the reset state of the fifo
    set_port16(port + SH_SCIF_SCFCR_OFF, SH_SCIF_SCFCR_TFRST | SH_SCIF_SCFCR_RFRST, 0);

    // FIFO control
    out16(port + SH_SCIF_SCFCR_OFF + 4, 0x0);
    Delay(1);
```

```
    out8(port + SH_SCIF_SCFTDR_OFF, 0);
    Delay(1);

    out16(port + SH_SCIF_SCSPTR_OFF + 4, 0x41);

    out16(port + SH_SCIF_SCFCR_OFF, 0x30 | 0x200 | SH_SCIF_SCFCR_MCE);

    out16(port + SH_SCIF_SCSCR_OFF, SH_SCIF_SCSCR_RE | SH_SCIF_SCSCR_TE |
                                    SH_SCIF_SCSCR_TIE | SH_SCIF_SCSCR_RIE);
    Delay(1);
    out16(port + SH7760_SCIF_SCSPTR_OFF, SH_SCIF_SCSPTR_RTSIO);

    out16(port + SH_SCIF_SCSCR_OFF, SH_SCIF_SCSCR_RE | SH_SCIF_SCSCR_RIE);

    return (port);
}


int
mini_serial(int state, void *data)
{
   uint8_t     *bp = 0;
   uint16_t count;
   uint8_t *dptr, c;
   struct mserial_data *mdata;
   int num, i;
   int          pending_interrupts;

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

   if (state == MDRIVER_INTR_ATTACH)
   {
      /* disable the serial interrupt */
      set_port16(mdata->port + SH_SCIF_SCSCR_OFF, SH_SCIF_SCSCR_RE | SH_SCIF_SCSCR_RIE, 0);
     return (1);
   }
   else if (state == MDRIVER_INIT)
   {
      /* the first time called initialize the hardware and do data setup */
      mdata->port = init_hw();
      if (mdata->port == 0)
 return (1);
      mdata->intr_calls = 0;
      mdata->ncalls = 0;
      mdata->errors = 0;
      mdata->data_len = 0;
   }
   else if (state == MDRIVER_STARTUP_PREPARE)
   {
      /* once we are out of startup use callout_io_map */
      if ((mdata->port_k = callout_io_map(0x120, SH_SCIF_BASE)) == 0)
      {
 /* something bad so disable the driver */
 set_port16(mdata->port + SH_SCIF_SCSCR_OFF, SH_SCIF_SCSCR_RE | SH_SCIF_SCSCR_RIE, 0);
 return (1);
       }
   }

   /* count the number of times the mini-driver is called */
   count = mdata->ncalls + 1;
   mdata->ncalls = count;
   if (state == MDRIVER_PROCESS)
   {
      /* called because of an interrupt */
      count = mdata->intr_calls + 1;
      mdata->intr_calls = count;
   }

   /* get the data from the serial port and clear any errors */

   // Handle the error messages: Overrun, Framing, Parity
   pending_interrupts = in16(mdata->port + SH_SCIF_SCFSR_OFF);
   if (in8(mdata->port + SH7760_SCIF_SCLSR_OFF) & SH_SCIF_SCLSR_ORER)
   {
```

```
        set_port16(mdata->port + SH7760_SCIF_SCLSR_OFF, SH_SCIF_SCLSR_ORER, 0);
    }
    if ((pending_interrupts & SH_SCIF_SCFSR_PER))
    {
        set_port16(mdata->port + SH_SCIF_SCFSR_OFF, SH_SCIF_SCFSR_PER |
            SH_SCIF_SCFSR_ER | SH_SCIF_SCFSR_DR, 0);
    }
    if (pending_interrupts & SH_SCIF_SCFSR_FER)
    {
        set_port16(mdata->port + SH_SCIF_SCFSR_OFF,
            SH_SCIF_SCFSR_FER | SH_SCIF_SCFSR_ER | SH_SCIF_SCFSR_DR, 0);
    }
    if (pending_interrupts & SH_SCIF_SCFSR_BRK )
    {
        set_port16(mdata->port + SH_SCIF_SCFSR_OFF, SH_SCIF_SCFSR_BRK | SH_SCIF_SCFSR_DR, 0);
    }

    /* grab data */
    if (in16(mdata->port + SH7760_SCIF_SCRFDR_OFF) )
    {
        c = in8(mdata->port + SH_SCIF_SCFRDR_OFF);
        set_port16(mdata->port + SH_SCIF_SCFSR_OFF, SH_SCIF_SCFSR_RDF, 0);

        dptr[mdata->data_len] = c;
        mdata->data_len = mdata->data_len + 1;
    }

    if (pending_interrupts & (SH_SCIF_SCFSR_TDFE|SH_SCIF_SCFSR_TEND) )
    {
        set_port16(mdata->port + SH_SCIF_SCSCR_OFF, SH_SCIF_SCSCR_TIE,0);
    }

    //Clear all the interrupts
    set_port16(mdata->port + SH7760_SCIF_SCLSR_OFF, SH_SCIF_SCLSR_ORER, 0);
    set_port16(mdata->port + SH_SCIF_SCFSR_OFF, 0xff, 0);

    if (state == MDRIVER_STARTUP_FINI)
        mdata->port = mdata->port_k;

    return (0);
}
```

In this example, the handler function stores call information, so a structure has been created to allow easier access to the data area. The data area is filled with the received characters.

During the MDRIVER_INIT state, the data area is initialized and the serial hardware is set up; this is called only once. At this time, the *startup_io_map()* is called to map in the hardware registers, and the pointer is stored in the data area.

This pointer is used for hardware access until the handler is called with a state of STARTUP_MDRIVER_PREPARE. At this time, *callout_io_map()* is called and the pointer is stored in the data area, however the *startup_io_map()* pointer should still be used. Once the handler is called with a state of MDRIVER_STARTUP_FINI, the handler starts using the pointer returned by *callout_io_map()* for all hardware access. This pointer is then used for all further invocations of the minidriver handler.

When the handler is called with a state of MDRIVER_INTR_ATTACH; it disables the device interrupt and returns a value of 1 requesting an exit of the handler.

Adding your minidriver to the system

Once you have written a handler function, you need to register it with startup and allocate the required system RAM for your data area. This is accomplished with the following functions:

```
paddr_t alloc_ram(phys_addr, size, alignment);
int mdriver_add(name, interrupt, handler, data_paddr, data_size);
```

Since you're allocating memory and passing an interrupt, these functions must be called after RAM is initialized by calling *init_raminfo()*, and after the interrupt information is added to the system page by calling *init_intrinfo()*. The *main()* function of our 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 which
* may need to be changed.
*/
init_intrinfo();

mdrvr_addr = alloc_ram(~0L, 65536, 1); /* grab 64k */
mdriver_add("mini-serial",  0,  mini_serial, mdrvr_addr, 65536);
...
```

In this example, you have allocated 64 KB of memory and registered the minidriver handler function with the name **`mini-serial`**.

Build startup

The minidriver is now complete, and you should rebuild startup and the boot image. In order to verify that the minidriver is running properly, you may want to add debug information into the handler function or write an application to read the minidriver data area and print the contents.

Testing your minidriver

The Biscayne board has a debug serial port that this minidriver example uses to receive bytes on. In order to test this driver, you should build an OS image that sets up TCP/IP and lets you **`telnet`** to the board. Since you're using the serial port as a data source, don't run the serial driver and don't put debug printouts or verbosity on **`procnto`** in your image. Once you have an image, you should burn it onto the onboard flash where you can boot from it.

Connect a NULL-modem cable between your Biscayne board and a host machine. The minidriver is running at 14400 baud, so make sure the host machine's serial port is also configured in this manner. On the host machine, begin sending characters to the serial port. Sample code should look like this:

```
...
uint8_t test_data[20] = {"012345678987654321"};
...
```

```
if ((fd = open("/dev/ser1", O_RDWR)) == -1)
{
   fprintf(stderr, "Unable to open device: %s (errno=%d\n", device, errno);
   return (-1);
}

for (;;)
{
write(fd, test_data + index, 1);
   index++;
   if (index == 19)
   index = 0;
}
```

This code sends a pattern of characters to the serial port so that the minidriver can capture data.

Now boot the Biscayne board and telnet to it. Once connected, you can run the sample full driver, **`mdrvr-sescif`** that does the following:

- attaches to the serial interrupt, stopping the minidriver
- reads the minidriver data area
- captures 64 characters from the serial port
- displays the minidriver data and the captured data, which should show the pattern of characters your sample application sent

Since the **`mdrvr-serscif`** application is a sample serial driver, you don't need to run the **`devc-sersci`** driver.

Sample Timings

These timings are based on sample boot image with the following properties:

- boot from Flash
- uncompressed image size: 748224 bytes
- sample minidriver running, data driver only
- no interrupt used

mdriver_max	Number of calls	Average calling interval
1 KB	715	59 microseconds
4 KB	139	194 microseconds
16 KB	65	934 microseconds

Boot time identifies the time from startup to the start of the first application process. The first invocation time is the time from IPL to the first invocation of the minidriver (MDRIVER_INIT).

OMAP minidriver

This sample is a minidriver for the OMAP 5912 board. The driver initializes the serial port at a known baud rate and buffers the characters into the data area. Once the system is booted, a program can then read this data area and retrieve the buffered characters. For this implementation we require:

- the *mdriver_max* value of 4 KB
- data storage of 64 KB
- hardware access to the serial port registers
- the minidriver handler function

The prototype for the minidriver handler function is as follows:

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

The *state* value informs the handler function where in the boot process it is being called from. See the *mdriver_add()* documentation in the API and Datatypes chapter of this guide. The *data* parameter is a virtual pointer to the minidriver's allocated data area. For this sample driver, the source code for the handler function should look like this:

```
struct mserial_data
{
   uintptr_t port;
   uintptr_t port_k;
   uint16_t intr_calls;
   uint16_t ncalls;
   uint16_t errors;
   uint16_t err;
   uint16_t last_count;
   uint16_t  data_len;
};

#ifndef write_omap
#define write_omap(__port,__val) out8(__port,__val)
#endif

#ifndef read_omap
#define read_omap(__port) in8(__port)
#endif

/*
 * Initialize the UART hardware
 * base address = 0xfffb0000
 * baud rate = 14400
 * data = 8 n 1
 */
uintptr_t
init_hw()
```

```
{
   unsigned value = 0;
   unsigned  msr, c;
   uintptr_t port;
   uint64_t base = 0xfffb0000;
   int         baud = 14400;

   if ((port = startup_io_map(OMAP_UART_SIZE, base)) == 0)
      return (0);

   // hit LCR with special byte to enable access to the Enhanced Feature Register (EFR)
   write_omap(port + OMAP_UART_LCR,0xbf);

   // turn off S/W flow control, enable writes to MCR[7:5], FCR[5:4], and IER[7:4]
   write_omap(port + OMAP_UART_EFR, 0x10);

   write_omap(port + OMAP_UART_LCR,0);

   // set MCR bit 6 to enable access to TCR and TLR registers
   write_omap(port + OMAP_UART_MCR, 0x40);

   value = 0x0;
   // set TCR - RX FIFO - start Rx at 0 bytes, halt at rx fifo value
   write_omap(port + OMAP_UART_TCR,value >> 4);

   write_omap(port + OMAP_UART_TLR, value);

   // disable access to TCR and TLR
   write_omap(port + OMAP_UART_MCR, 0x00);
   write_omap(port + OMAP_UART_SCR, 0x00);

   write_omap(port + OMAP_UART_LCR,0xbf);

   // disable access to MCR[7:5], FCR[5:4], and IER[7:4],
   // disable auto flow control and sw flow control
   write_omap(port + OMAP_UART_EFR, 0x00);

   // set Divisor Latch Enable - writing something other
   // than 0xbf to LCR puts it back in "normal" mode
   write_omap(port + OMAP_UART_LCR, 0x80);

   // baud and clk
   value = 48000000 / (16 * baud);
   write_omap(port + OMAP_UART_DLL, value);
   write_omap(port + OMAP_UART_DLH, (value >> 8) & 0xff);
   write_omap(port + OMAP_UART_LCR, 0x13);

   // turn on DTR, RTS
   c = read_omap(port + OMAP_UART_MCR);
   write_omap(port + OMAP_UART_MCR, (c & ~OMAP_MCR_DTR|OMAP_MCR_RTS) |
   OMAP_MCR_DTR|OMAP_MCR_RTS);

   // According to the National bug sheet you must wait for the transmit
   // holding register to be empty.
   do {
   } while((read_omap(port + OMAP_UART_LSR) & OMAP_LSR_TXRDY) == 0);

   // Clean the device so we get a level change on the intr line to the bus.
   // Enable out2 (gate intr to bus)
   c = read_omap(port + OMAP_UART_MCR);
   write_omap(port + OMAP_UART_MCR, (c & ~OMAP_MCR_OUT2) | OMAP_MCR_OUT2);

   write_omap(port + OMAP_UART_IER, 0); // Disable all interrupts
   read_omap(port + OMAP_UART_LSR); // Clear Line Status Interrupt
   read_omap(port + OMAP_UART_RHR); // Clear RX Interrupt
   read_omap(port + OMAP_UART_THR); // Clear TX Interrupt
   read_omap(port + OMAP_UART_MSR); // Clear Modem Interrupt

   // Enable interrupt sources.
   write_omap(port + OMAP_UART_IER, 0x01);

   // get current MSR stat
   msr = read_omap(port + OMAP_UART_MSR);

   return (port);
```

```
}

int
mini_serial(int state, void *data)
{
    uint16_t count;
    uint8_t *dptr, c;
    struct mserial_data *mdata;
    unsigned lsr, iir;

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

    if (state == MDRIVER_INTR_ATTACH)
    {
       /* disable the serial interrupt */
       write_omap(mdata->port + OMAP_UART_IER, 0x00);
       return (1);
    }
    else if (state == MDRIVER_INIT)
    {
       if ((mdata->port = init_hw()) == 0)
  return (1);

/* clear the data area counters */
mdata->intr_calls = 0;
mdata->ncalls = 0;
mdata->errors = 0;
mdata->data_len = 0;
}
else if (state == MDRIVER_STARTUP_PREPARE)
{
   /* once we are out of startup use callout_io_map */
   if ((mdata->port_k = callout_io_map(OMAP_UART_SIZE, 0xfffb0000)) == 0)
   {
      /* something bad so disable the driver */
      write_omap(mdata->port + OMAP_UART_IER, 0x00);
      return (1);
   }
}

/* count the number of times the mini-driver is called */
count = mdata->ncalls + 1;
mdata->ncalls = count;
if (state == MDRIVER_PROCESS)
{
   /* called because of an interrupt */
   count = mdata->intr_calls + 1;
   mdata->intr_calls = count;

   iir = read_omap(mdata->port + OMAP_UART_IIR) & 0x07;
   if (iir == OMAP_II_RX) /* receive interrupt */
   {
   do {
 dptr[mdata->data_len] = read_omap(mdata->port + OMAP_UART_RHR) & 0xff;
 mdata->data_len = mdata->data_len + 1;

 lsr = read_omap(mdata->port + OMAP_UART_LSR);
 /* check for errors */
 if((lsr & (OMAP_LSR_RCV_FIFO | OMAP_LSR_BI|OMAP_LSR_OE|
                   OMAP_LSR_FE|OMAP_LSR_PE)) != 0)
 {
    // Read whatever input data happens to be in the buffer to "eat" the
    // spurious data associated with break, parity error, etc.
    c = read_omap(mdata->port + OMAP_UART_RHR);
    c = c;
 }
    } while(lsr & OMAP_LSR_RXRDY);
      }
}
else
{
   /* poll for data */
   while (read_omap(mdata->port + OMAP_UART_LSR) & OMAP_LSR_RXRDY)
   {
```

```
        dptr[mdata->data_len] = read_omap(mdata->port + OMAP_UART_RHR) & 0xff;
        mdata->data_len = mdata->data_len + 1;
    }
}

if (state == MDRIVER_STARTUP_FINI)
    mdata->port = mdata->port_k;

return (0);
}
```

In this example, our handler function stores call information, so a structure is created to allow easier access to the data area. The data area is filled with the received characters.

During the MDRIVER_INIT state, the data area is initialized and the serial hardware is set up. This is called only once. At this time, *startup_io_map()* is called to map in the hardware registers, and the pointer is stored in the data area.

This pointer is used for hardware access until the handler is called with a state of STARTUP_MDRIVER_PREPARE. At this time, *callout_io_map()* is called, and the pointer is stored in the data area, however the *startup_io_map()* pointer should still be used. Once the handler is called with a state of MDRIVER_STARTUP_FINI, the handler starts using the pointer returned by *callout_io_map()* for all hardware access. This pointer is then used for all further invocations of the minidriver handler. When the handler is called with a state of MDRIVER_INTR_ATTACH, it disables the device interrupt and returns a value of 1 requesting an exit of the handler.

Adding your minidriver to the system

Once you have written a handler function, you need to register it with startup and allocate the required system RAM for your data area. This is accomplished with the following functions:

```
paddr_t alloc_ram(phys_addr, size, alignment);
int mdriver_add(name, interrupt, handler, data_paddr, data_size);
```

Since you're allocating memory and passing an interrupt, these functions must be called after RAM is initialized by calling *init_raminfo()*, and after the interrupt information is added to the system page by calling *init_intrinfo()*. The *main()* function of the startup **`main.c`** should look like this:

```
...
paddr_t mdrvr_addr;
...
init_intrinfo();
    init_qtime();

    /* allocate 64k of ram for the minidriver's use */
    mdrvr_addr = alloc_ram(~0L, 64*1024, 1);
    /* code to add a sample minidriver which merely does data collection */
    mdriver_add("mini-data", 0, mini_data, mdrvr_addr, 64*1024);
    /* code to add a sample minidriver for a serial port */
    /* mdriver_add("mini-serial", 46, mini_serial, mdrvr_addr, 64*1024); */
...
```

In this example, you have allocated 64 KB of memory and registered the minidriver handler function with the name **`mini-serial>`**

Build startup

Our minidriver is complete, and you should now rebuild startup and the boot image. In order to verify that the minidriver is running properly, you may want to add debug information to the handler function or write an application to read the minidriver data area and print the contents.

Testing your minidriver

The OMAP 5912 board has a debug serial port that this minidriver example uses to receive bytes on. In order to test this driver, you should build an OS image that sets up TCP/IP and lets you **`telnet`** to the board. Since you'll be using the serial port as a data source, don't run the serial driver and don't put debug printouts or verbosity on **`procnto`** in your image. Once you have an image, you should burn it onto the onboard flash where you can boot from it.

Connect a NULL-modem cable between your OMAP board and a host machine. The minidriver is running at 14400 baud, so make sure the host machine's serial port is also configured in this manner. On the host machine, begin sending characters to the serial port. Sample code should look like this:

```
...
uint8_t test_data[20] = {"0123456789876543 21"};
...

if ((fd = open("/dev/ser1", O_RDWR)) == -1)
{
   fprintf(stderr, "Unable to open device: %s (errno=%d\n", device, errno);
   return (-1);
}

for (;;)
{
write(fd, test_data + index, 1);
   index++;
   if (index == 19)
      index = 0;
}
```

This code sends a pattern of characters to the serial port so that the minidriver can capture data.

Now boot the OMAP 5912 board and **`telnet`** to it. Once connected, you can run the sample full driver, **`mdrvr-seromap`**, which does the following:

- attaches to the serial interrupt, stopping the minidriver
- reads the minidriver data area
- captures 64 characters from the serial port
- displays the minidriver data and the captured data, which should show the pattern of characters your sample application sent

Since the **`mdrvr-seromap`** application is a sample serial driver, you don't need to run the **`devc-seromap`** driver.

Sample timings

These timings are based on a sample boot image with the following properties:

- boot from Flash
- uncompressed image size: 801564 bytes
- sample minidriver running, data driver only
- no interrupt used

mdriver_max	Number of calls	Average calling interval
1 KB	782	93 microseconds
4 KB	229	352 microseconds
16 KB	91	1.38 ms

Boot time	First invocation time
170 ms	1 ms

Boot time identifies the time from startup to the start of the first application process. The first invocation time is the time from IPL to the first invocation of the minidriver (MDRIVER_INIT).

Appendix A

Hardware Interaction within the Minidriver

The following example shows how to interact with hardware from within a minidriver.The important message is that the mapping of hardware registers is different, depending on where in the boot process that the minidriver is called. This transition is handled in the MDRIVER_STARTUP_PREPARE and the MDRIVER_STARTUP_FINI stages.

```
/-------------------------------------------------------------------------/
Here is a more complex example of a mini driver for a fictional hardware
device called 'MYBUS'.
Characteristics of MYBUS are as follows:

- series of 8 bit registers (status and data) at address 0xFF000000  (MBAR)
- interrupt 56 is generated when a character arrives at the MYBUS port

There are registers at this address which will be read/written to.

/-------------------------------------------------------------------------/

#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;

/**************    Mini-driver data area   ************/

typedef struct
{
   MYBUS_regs    *MYBUS_REGS;                  //this value will either be
                                               //same as PREKERNEL or POSTKERNEL
   MYBUS_regs    *MYBUS_REGS_PREKERNEL_START;  //register  mappings to
                                               //use before kernel starts
   MYBUS_regs    *MYBUS_REGS_POSTKERNEL_START; //register mappings to use
                                               //after kernel starts
   U16           total_message_counter;        //total times mini-handler
                                               //is called
   U16           process_counter;              //times called after kernel
                                               //is running
   U16           kernel_counter;               //times called during
                                               //kernel 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 only called once when the mini-driver 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 ..
//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 .. mini-driver
                    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;

      mdata->total_message_counter = 0;    // Initialize messages received
             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 mini-driver */
           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 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);
}
```