LynxOS-178 Device Driver Writer's Guide

LynxOS-178

DOC-2208-00

Printed in the United States of America.

Table of Contents

Preface

Typographical Conventions

The typefaces used in this manual, summarized below, emphasize important concepts. All references to filenames and commands are case-sensitive and should be typed accurately.

Kind of Text	Examples
Body text; *italicized* for emphasis, new terms, and book titles	Refer to the *LynxOS-178 Device Driver Writer's Guide*
Environment variables, filenames, functions, methods, options, parameter names, path names, commands, and computer data	`ls -l myprog.c /dev/null`
Commands that need to be highlighted within body text or commands that must be typed as is by the user are **`bolded`**.	`login:` **`myname`** `#` **`cd /usr/home`**
Text that represents a variable, such as a filename or a value that must be entered by the user, is *italicized*.	`cat` *<filename>* `mv` *<file1> <file2>*

Blocks of text that appear on the display screen after entering instructions or commands	`Loading file /tftpboot/shell.kdi into 0x4000`
	`.....................................`
	`File loaded. Size is 1314816`
	`© 2019 Lynx Software Technologies, Inc. All rights reserved.`

Keyboard options, button names, and menu sequences	**`Enter, Ctrl-C`**

Technical Support

Lynx Software Technologies handles support requests from current support subscribers. For questions regarding Lynx Software Technologies products, evaluation CDs, or to become a support subscriber; our knowledgeable sales staff will be pleased to help you. Please visit us at:
http://www.lynx.com/training-support/contact-support/

How to Submit a Support Request

When you are ready to submit a support request, please include *all* of the following information:

- First name, last name, your job title
- Phone number, e-mail address
- Company name, address
- Product version number
- Target platform (for example, PowerPC)
- Board Support Package (BSP), Current Service Pack Revision, Development Host OS version
- Detailed description of the problem that you are experiencing:
- Is there a requirement for a US Citizen or Green Card holder to work on this issue?
- Priority of the problem - Critical, High, Medium, or Low?

Where to Submit a Support Request

Support, Europe	tech_europe@lynx.com +33 1 30 85 93 96
Support, worldwide except Europe	support@lynx.com +1 800-327-5969 or +1 408-979-3940 +81 33 449 3131 [for Japan]
Training and Courses	USA: training-usa@lynx.com Europe: training-europe@lynx.com USA: +1 408-979-4353 Europe: +33 1 30 85 06 00

CHAPTER 1 Driver Basics

This chapter describes the LynxOS-178 driver types, major and minor numbers, and how to reference devices and drivers.

Driver Types

A *Device Driver* is defined as a software interface to hardware devices. It enables operating systems to access hardware functions without knowing the precise details of the hardware that is being used. Further, it is designed to separate the characteristics of a particular device from the Kernel.

In LynxOS-178, device drivers are either statically or dynamically linked to the Kernel, reside in kernel space, and have complete and unrestricted access to operating system structures. Therefore, it is the developer's responsibility to restrict the access of a device driver to only the necessary structures.

The standard functions (called entry points) device drivers present to the operating system follow a predefined format. LynxOS-178 supports two types of drivers:

- Character Driver
- Block Driver

Driver type selection is determined by the physical device's requirements. For example, if the Kernel buffer cache need not be used, then it makes sense to use a character driver as opposed to a block driver.

The major differences between a character driver and a block driver are described in the next sections.

Character Driver

A *Character Driver* has the following characteristics:

- A character driver has the `read()` and `write()` entry points and no `strategy()` entry point.

- A character driver can exist on its own and operates in *raw* mode only.

The following physical devices are usually handled by a Character Driver:

- Serial port driver
- Parallel port driver
- Analog-to-digital card driver
- Network card driver (TCP/IP portion of the driver)

Block Driver

A *Block Driver* is the combination of a character driver and a block driver. Both sets of entry points refer to the same statics structure, but each set is accessed through a different major number. If the character entry points are used, the block driver is operating in *raw* mode. If the block driver entry points are used, the block driver is operating in *block* mode.

NOTE: The major difference between raw mode and block mode is that raw mode is not cached, while block mode is cached.

A Block Driver has the following characteristics:

- A block driver has a `strategy()` entry point and no `read()` or `write()` entry points.
- A block driver is tied in to the Kernel disk cache. All requests using the `strategy()` routine will go through the disk cache. As a result, only a block driver can support a file system.

The following physical devices are usually handled by a block driver:

- Disk drivers including the following devices:
  - SCSI
  - IDE
  - ramdisk
  - Flash Memory

Device Classifications

LynxOS-178 identifies devices using major and minor numbers. A major device corresponds to the set of entry points of a driver and a set of resources. A minor number is used to divide the functionality of the major number. The major number and minor number take up 24 bits. The minor number uses the lower 8 bits and the major number the next 16 bits. The upper 8 bits of the 32-bit value containing the major and minor numbers are reserved for block size

For example, assume the major number *x* corresponds to an Adaptec 2940 SCSI card. All SCSI devices controlled by that card will have major number *x* because they use the same interrupt vector and I/O ports. The minor number is used to select the SCSI device and partition. The minor number uses 4 bits for the SCSI device and 4 bits to represent the partition. This gives a maximum of 16 SCSI devices and 16 partitions per major number.

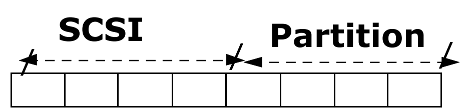


Figure 1-1: Minor Number for the Adaptec 2940 Driver

Another example is a timer card driver. It has a major number of *y*. Because it has no extra functionality, it has a minor number of 0. Therefore, the timer card driver can be accessed by using only the major number (major number + 0 = major number).

In general, when communicating with drivers the major number selects the appropriate entry points to use in the driver, and the minor number selects different capabilities within each major number.

Referencing Drivers and Devices

Drivers and Devices in LynxOS-178 can be referenced by using their identification numbers.

Drivers

Drivers are referenced using a unique driver identification number. This number is assigned automatically during Kernel configuration for statically linked drivers. Drivers supporting raw (character) and block interfaces have separate driver identification numbers for each interface. The `drivers` command displays the drivers currently installed in the system and their unique driver identification numbers.

Table 1-1 shows a sample output of the `drivers` command.

Table 1-1: Sample Output of the drivers Command

ID	Type	Major Devices	Start	Size	Name
0	char	1	0	0	null
1	char	1	0	0	mem
2	char	1	0	0	ctrl driver
3	char	1	0	0	Raw floppy
4	block	1	0	0	Floppy
5	char	1	0	0	SIM1542 RAW SCSI
6	block	1	0	0	SIM1542 BLK SCSI
7	char	1	0	0	kdconsole
8	char	2	0	0	serial

Devices

Each device is identified by a pair of major/minor numbers. LynxOS-178 automatically assigns the major numbers for static devices during Kernel generation. Dynamic devices are assigned major numbers when they are loaded. Character and block interfaces for the same device are indicated by different major numbers.

To view major devices installed on the system, use the `devices` command.

The ID column of Table 1-2 gives the major number of the device.

Table 1-2: Sample Output of the devices Command

ID	Type	Driver	Use Count	Start	Size	Name
0	char	0	2	0	0	null device
1	char	1	1	0	0	memory
2	char	2	0	0	0	ctrl dev
3	char	3	0	db0d7008	0	raw Floppy 0–3
4	char	5	1	db0d8a70	0	SIM1542 RAW SCSI
5	char	7	9	db0d8fd8	0	kdconsole
6	char	8	0	db0dc260	0	com 1
7	char	8	0	db0dce40	0	com 2
0	block	4	0	db0d7008	0	Floppy 0–3
1	block	6	2	db0d8a70	0	SIM1542 SCSI

Minor devices are identified by the minor device number. These numbers may be used to indicate devices with different attributes. Minor device numbers are necessary only if there are multiple minor devices per major device. The meaning of the minor device number is selected and interpreted only by the device driver. The Kernel does not attach a special meaning to the minor number. For example, different device drivers will use the minor device number in different ways: device type, SCSI target ID (for example, a SCSI disk controller driver), or a partition (for example, an IDE disk controller driver).

Application Access to Drivers and Devices

Like UNIX, LynxOS-178 is designed so that devices and drivers appear as files in the file system. Applications can access devices and drivers using these special device files. These files usually reside in the `/dev` directory (although they can be put anywhere) and are viewable, like other files, through the `ls -l` command. The device special files are named the same way as regular files and are identified by the device type (character [c] or block [b]) in the first character of the first column of the listing. Special device files have a file size of 0; however, they do occupy an inode and take up directory space for their name.

The size column of `ls -l` listing of such files shows the device's major and minor numbers, respectively (see Table 1-3). Special device files are created with the `mknod` utility.

Table 1-3: Sample `ls -l` Output of the /dev Directory

Permissions	Links	Owner	Major, Minor	Modification Date	Device File Name
crw-rw-rw-	1	root	0,0	Mar 29 01:57	null
crw-r--r--	1	root	1,0	Mar 29 01:52	mem
crw-rw-rw-	1	root	2,0	Mar 29 01:52	tty
crw-rw-rw-	1	root	3,12	Mar 29 01:52	rfd1440.0
crw-------	1	root	4,0	Mar 29 01:52	rsd1542.0
crw-------	1	root	4,16	Mar 29 01:52	rsd1542.0a
crw--w--w-	1	root	5,0	Mar 29 01:58	atc0
crw--w--w-	1	root	5,1	Mar 29 01:57	atc1
crw-rw-rw-	1	root	6,0	Mar 29 01:52	com1
crw-rw-rw-	1	root	7,0	Mar 29 01:52	com2
brw-rw-rw-	1	root	0,12	Mar 29 01:52	fd1440.0
brw-------	1	root	1,0	Mar 29 01:52	sd1542.0
brw-------	1	root	1,16	Mar 29 01:52	sd1542.0a

Mapping Device Names to Driver Names

The following method can be used to map a device name to a driver:

- Use the `ls -l` command on the `/dev` directory to obtain the listing of all the device names in the system. Determine the major and minor numbers associated with the name of the device. For example, in the above table, the device `com1` could be a character device with a major device number of 6 and a minor device number of 0.
- Use the `devices` command to get a listing of all the devices in the system. The value in the 'id' column corresponds to the major device number obtained above. If there is more than one entry with the same ID, the device type (character or block) eliminates any ambiguity involved. After locating the entry for the driver in question, look in the third

column with the heading ‘driver.’ This is the driver ID. For example, in Table 1-2 on page 5, `com1` has the driver ID of 8.

- Use the `drivers` command to get a listing of all the drivers in the system; with the driver ID obtained in the above step, obtain the name of the driver. For `com1`, the driver name is `serial`, which is the driver with ID 8 in Table 1-1 on page 4.

CHAPTER 2 Driver Structure

This chapter describes device driver structures and entry points.

Throughout this chapter, the LynxOS-178 RS-232 device driver is used to give illustrative examples to the ideas described herein. In order to save space and for the sake of making the explanation transparent, some simplifications have, however, been made in this chapter. The reader should refer to the source code of the RS-232 driver for the complete version of the driver code.

Device Information Structure

Each device driver consists of entry points, other routines, and various data structures. Entry points are functions within the driver that are standard methods of calling the driver. The important data structures are as follows:

- The *Device Information Structure*
- The *Statics Structure*

The device information structure provides the means for users to configure parameters for each major device.

Device Information Definition

The *Device Information Definition* contains the device information necessary to install a major device. This includes information that may vary between instances of a physical device if the hardware is replicated in the system. Examples of information that may change include the following items:

- Physical Addresses
- Interrupt Vectors
- Available Resources

The device information definition may also contain information about configuration defaults. There is one device information definition for each instantiation of the driver.

The implementation of the device information definition is done through a C structure, which consolidates the information specific to a particular major device. The structure is called typically called *<dev>*info, where *<dev>* corresponds to the device name (note that the example outlined below does not completely follow the exact naming convention).

For example, an RS-232 port device driver has a device information definition named RS232_InfoRec_s, which is given below. The C structure definition is put in a file typically named *<dev>*info.h, where *<dev>* is a prefix that corresponds to a particular device. The header file may also contain definitions of device-specific constants as well as various convenient macro definitions. The device information definition files are located in the driver specific directories (for example, $ENV_PREFIX/sys/drivers/rs232/rs232info.h).

In the example in this chapter, the header file rs232info.h contains the following data:

```
// rs232info.h -- Device information definition for the RS232 port.

#include <arch_mem.h> // kaddr_t

typedef struct RS232_InfoRec_s {
    kaddr_t port; // com1
    unsigned long default_config_reg; // Default configuration
    unsigned int time_out; // Time-out for polling loop
    unsigned int speed; // Com1 baud rate
    unsigned long freq; // Com1 frequency
} RS232_InfoRec_t, *RS232_InfoPtr_t;
```

Device Information Declaration

The *Device Information Declaration* is an instance of the device information definition. The data is passed to the appropriate driver upon installation. Each device information declaration must have a corresponding device information definition.

The device information declaration can be implemented by declaring a C structure in the device information definition file. By convention the device information declaration is named *<dev>*info*<n>*, where *<dev>* is the appropriate device name and *<n>* is a device sequence number, starting at 0. The declaration is put in a C source file named *<dev>*info.c, where *<dev>* stands for the device name. The device information declaration files are located in the driver specific directories (for example, `$ENV_PREFIX/sys/drivers/rs232/rs232info.c`).

The following code segment shows an example of a device information declaration for the RS-232 port:

```
// rs232info.c -- Device information declaration for the RS232 port.

#include “rs232info.h” // RS232 device information structure definition
#include <baudrate.h> // B115200
RS232_InfoRec_t rs232info0 = {
      .port = 0,
      .default_config_reg = 0x0400,
      .time_out = 500000,
      .speed = B115200,
      .freq = 0
};
RS232_InfoRec_t rs232info1 = {
      .port = 0,
      .default_config_reg = 0x0400,
      .time_out = 500000,
      .speed = B115200,
      .freq = 0
};
```

Statics Structure

The routines within the driver must operate on the same set of information for each instance of the driver. In LynxOS-178, this information is known as the driver *statics structure*. This structure is considered to be the device driver’s Internal Logical Resource, but it can contain (pointers to) other types of resources (Class 3 or Class 4). For more information, refer to the LynxOS-178 (RSC) Partitioning and RSC Interface Analysis Document (available for purchase as part of the DO-178 Artifacts Package). The install entry point returns the address of the statics structure, and the remaining entry points are passed a pointer to the statics structure by the Kernel. If the statics structure is not globally declared, then the driver routines can be used with more than one statics structure simultaneously.

The following is an example of a statics structure:

The format of the statics structure for most drivers can be defined in any way the user wishes. However, for some drivers, namely `TTY` and network drivers, there are certain rules that must be followed.

Entry Points

A driver consists of a number of *Entry Points*. These routines provide a defined interface with the rest of the kernel. The kernel calls the appropriate entry point when a user application makes a system call. LynxOS-178 dictates a predefined list of entry points, though a driver does not need to have all the entry points. It is recommended that empty routines for all unused entry points be provided.

install Entry Point

Before a device can be used, a driver must allocate and initialize the necessary data structures and initialize the hardware if it is present. This is the responsibility of the `install` entry point in a LynxOS-178 driver.

The `install` entry point may be called multiple times, once for each of the driver's major devices configured in the system. The `install` entry point is passed the address of a user-defined device information structure containing the user-configured parameters for the major device. Each invocation of the `install` entry point must receive a different address. Otherwise, different master numbers will correspond to the same device information structure, which almost always creates device/driver conflicts and represents a bug.

Character Drivers

As an example of the character driver `install` entry point, the `RS232_install()` routine from the LynxOS-178 RS-232 driver is used here:

```
// rs232drvr.c -- The RS-232 driver install entry point.
#include "rs232info.h"

void *
RS232_install(RS232_InfoPtr_t info)
{
    RS232_StaticsPtr_t s; // Statics structure.
    const dev_info_t *uart
    // Allocate statics
    s = (RS232_StaticsPtr_t)sysbrk(sizeof(RS232_Statics_t));

    if (s == NULL) {
      pseterr(ENOMEM);
      return void )SYSERR;
    }

    uart = bsp_get_dev_by_type(uart_found++, UART_DEV_TYPE);

    if (uart == NULL) {
      return (void *)SYSERR;
    }

    // Clean statics.
    memset(s, 0, sizeof(RS232_Statics_t));
```

```
    ...

    // Fill-in statics.
    s->info = info;
    s->info->time_out = tickspersec * s->info->time_out / 1000;

    pi_init(&s->write_sem_A);
    pi_init(&s->write_sem_B);

    pi_init(&s->read_sem_A);
    pi_init(&s->read_sem_B);

    return s;
}
```

In this example, an error in memory allocation for the driver statics structure leads to setting the `errno` variable to `ENOMEM` and returning the `SYSERR` constant. Otherwise, the fields of the driver statics structure are initialized in a manner specific to the RS-232 hardware.

Commonly, a device driver will use hardware-specific internal functions to verify the physical presence of the hardware device prior to any further initialization steps, such as allocating memory for the driver statics structure. Refer to Chapter 7, “Interrupt Handling” for an example of a device driver implemented in this way.

Block Drivers

The following code snippet gives the general structure of the `install` entry point of a block device driver.

```
void *
devinstall(struct devinfo *info, struct statics *status)
{
    // check for existence of device.
    ...
    // Do initialization of the devices.
    ...
    // Set interrupt vectors.
    return status;
}
```

The `install` entry point allocates memory for the block driver’s statics structure and returns a pointer to its statics structure. The input statics structure is the one that was returned from the block driver’s associated “raw” character mode driver. If an error occurs during the install, the `install` entry point must free all resources used and return `SYSERR`.

open Entry Point

The `open` entry point performs the initialization generic to a minor device. For every minor device accessed by the application program, the `open` entry point of the driver is accessed. Thus, if synchronization is required between minor devices (of the same major device), the `open` entry point handles it. Every `open` system call on a device managed by a driver results in the invocation of the `open` entry point.

Note that the `open` entry point is not reentrant, though it is preemptive. Only one user task can be executing the entry point code at a time for a particular device. Therefore, synchronization between tasks is not necessary in this entry point. The general structure of the `open` entry point is shown in the following section.

Character and Block Drivers

This chapter uses the `RS232_open()` function to illustrate the structure of a typical `open()` entry point as well as to highlight some standard methods used for implementing LynxOS-178 device drivers.

```
#include <file.h>
#include <io.h>

DRVENTPT int RS232_open(

    RS232_StaticsPtr_t s,          /* device statics structure */
    int dev,                      /* device number */
    struct file *f                 /* device node information */
)
{

    int      retval;               /* return value OK or SYSERR */
    uid_t    current_vm;           /* current user id */
    uint8_t  minor_dev;            /* minor device */
    int      ps;                   /* variable for interrupt disabling
                                      and restoring */

   retval = OK;                     /* return value */

/*
    Get minor device number.
    Based on the minor device number open the specified port. Minor devices
    0 and 1 correspond to the first RS-232 port. Minor devices 2 and 3
    correspond to the second RS-232 port.
*/

/* calculate minor device number we are trying to open */
minor_dev = minor(f->dev);

    if(minor_dev > RS232B_BLCKING)
{
    pseterr(EINVAL);
    retval = SYSERR;
}
else
{
    if(minor_dev > RS232A_BLCKING)
{
        /* check to see if the RS-232 Port B has already been
```

```
            * claimed by another VM
            */
         /* if it has, set errno to EBUSY and return SYSERR */
         /* else claim the port */

         current_vm = getuid(); disable(ps);
         if(s->portb_owner == NOTUSED)
         {
            s->portb_owner = current_vm;
         }
         restore(ps);
         if((current_vm != s->portb_owner) && (current_vm != 0))
         {
            pseterr(EBUSY);
            retval = SYSERR;
         }

      }
    }
    return(retval);
   }
```

The `open` entry point returns either `OK` or `SYSERR`, depending on whether or not the device was successfully opened.

To obtain the major device number, use the macro `major()`. For the minor device number, use the macro `minor()` as shown in the code snippet above. The `s` parameter is the address of the device information structure passed to the device driver entry point by the operating system Kernel. The `f` parameter is a pointer to the file structure defined in `file.h`.

The `open` entry point returns either `OK` or `SYSERR`, depending on whether or not the device was successfully opened. If the return value is `SYSERR` then `pseterr()` is used to set `errno` for the caller.

close Entry Point

The `close` entry point is invoked only on the “last close” of a device. This means that if multiple tasks have opened the same device node, the close entry point is invoked only when the last of these tasks closes it. Memory allocated in the `open` routine is deallocated in the `close` entry point. As with the `open` entry point, the `close` entry point is not reentrant.

Character Drivers

```
#include <file.h>
#include <io.h>

DRVENTPT int RS232_close(
   RS232_StaticsPtr_t s,   /* device statics structure pointer */
   struct file *f)         /* device node information */
```

```
{
    int retval;              /* return value OK or SYSERR */
    uint8_t  minor_dev;      /* minor device */
    int ps;  /* variable for interrupt disabling and restoring */

        retval = OK;

        /* calculate minor device number we are trying to close */
        minor_dev = minor(f->dev);

        if(minor_dev > RS232A_BLCKING) {
            disable(ps);
            s->portb_owner = NOTUSED;
            restore(ps);
        }
        return(retval);
    }
```

The `close` entry point returns either `OK` or `SYSERR`, depending on whether or not the device was successfully closed.

Block Drivers

```
#include <file.h>
#include <io.h>

int devclose(struct statics *s, int devno)
{
    /* perform any deallocation done previously in open */
    return (OK);
}
```

read Entry Point

The read entry point copies data from the device into a buffer in the calling task. The structure of the read entry point in the LynxOS-178 RS-232 device driver appears as follows:

Character Drivers Only

```
int RS232_read(
    struct RS232_Statics_s *s,   /* device statics structure pointer */
    struct file *f,              /* device node information */
    char *buffer,                /* dst buffer */
    int count)                   /* number of bytes to read */
{

    /* reading from the RS-232 ports is not implemented. */
    pseterr(ENOSYS);
```

```
        return(SYSERR);

    }
```

The s parameter is the address of the statics structure passed to the read() entry point by the operating system Kernel. The f parameter is a pointer to the file structure. The parameters buffer and count refer to the address and size, in bytes, of the buffer in which the data should be placed. This entry point should return the number of bytes actually read or SYSERR.

LynxOS-178 device drivers support read functions. More information regarding the read entry point can be found in the section entitled “read Entry Point” on page 98.

write Entry Point

The write entry point copies data from a buffer in the calling task to the device. The code snippet below is a fragment of the write entry point in the LynxOS-178 RS-232 device driver. It is given here to demonstrate the general structure of a write entry point in a LynxOS-178 character device driver.

Character Drivers Only

```
int RS232_write(
    struct RS232_Statics_s *s, /* device statics structure pointer */\
    struct file *f,             /* device node information */
    char *buffer,               /* src buffer */
    int count)                  /* number of bytes to write */
{
    int bytes = 0;
    uint8_t port;               /* minor device number */
    kaddr_t trreg;
    kaddr_t streg;
    uint32_t TimeOut;           /* variable to store time-out limit */

    port = minor(f->dev);       /* get minor device number from the
                                   file structure */
   /* Check that the port is enabled */
   if(RS232IsPortDisabled(s, port)) {
       pseterr(EDISABLED);
       return (SYSERR);
   }

    /*
      Minor device 0 is the device node for the non-blocking RS-232 portA.
    */
    /*
       Minor device 1 is the device node for the blocking RS-232 portA.
    */
    /*
      Minor device 2 is the device node for the non-blocking RS-232 portB.
```

```
        */
        /*
           Minor device 3 is the device node for the blocking RS-232 portB.
        */

        if(port == RS232A_NONBLCKING)
        {

            /* first RS-232 port, non-blocking case */
            trreg = s->info->port1;
            streg = s->info->port1 + U_LSR;
            bytes = RS232ANonBlockWrite(s, trreg, streg, buffer, count);

        }
        else if(port == RS232A_BLCKING)
        {
            /* do the processing of the rest of port number/blocking state
                combinations here */
        }
    ...


        return(bytes);
    }
```

The `s` parameter is the address of the statics structure passed by the Kernel. The `f` parameter is a pointer to the file structure. The parameters `buffer` and `count` refer to the address and size, in bytes, of the buffer containing the data to be written to the device. This entry point should return the number of bytes actually written or `SYSERR`.

The `RS232_write()` routine given in the above example calls an internal function `RS232ANonBlockWrite()`, the latter providing the actual implementation of the write functionality in the case of the first RS-232 port in the non-blocking case.

select Entry Point

The LynxOS-178 RS-232 device driver does not implement the `select` entry point. This is why an abstract entry point is shown here instead of giving an example from the specific device driver.

The `select` entry point supports I/O polling or multiplexing. The `select` entry point structure looks like this:

Character Drivers

```
select (s, f, which, ffs)
struct  statics  *s;
struct file *f;
int which; struct sel *ffs;
{
```

```
switch (which)
    {
        case SREAD:
            ffs->iosem = &s->n_data;
            ffs->sel_sem = &s->rsel_sem;
            break;
        case SWRITE:
            ffs->iosem = &s->n_spacefree;
            ffs->sel_sem = &s->wsel_sem;
            break;
        case SEXCEPT:
            ffs->iosem = &s->error;
            ffs->sel_sem = &s->esel_sem;
            break;
    }
    return (OK);
}
```

The `which` field is either `SREAD`, `SWRITE`, or `SEXCEPT`, indicating that the `select` entry point is monitoring a read, write, or exception condition. The `select` entry point returns `OK` or `SYSERR`.

In addition to the `select` entry point code, a number of fields are typically in the statics structure are required to support the `select` system call:

```
struct statics
{
    ...
    ksem_t *rsel_sem;    /* sem for select read */
    ksem_t *wsel_sem;    /* sem for select write */
    ksem_t *esel_sem;    /* sem for select exception */
    int n_spacefree;     /* space available for write */
    int n_data;          /* data available for read */
    int error;           /* error condition */
};
```

The `iosem` field in the `sel` structure passed into the `select()` function is a pointer to a flag (the term `iosem` may be considered misleading as it is not a semaphore) that indicates whether the condition being polled by the user task is true or not. The `sel_sem` field of the `sel` structure is a pointer to a semaphore that the driver signals at the appropriate time (see below). A driver must always set the `iosem` and `sel_sem` fields in the select entry point.

A driver that supports select must also test and signal the select semaphores at the appropriate points in the driver, usually the interrupt handler or Kernel thread. This should be done when data becomes available for reading, when space is available for writing, or when an error condition is detected:

```
/* data input
*/s->n_data++;

disable  (ps);
if (s->rsel_sem)
```

```
    ssignal (s->rsel_sem);
restore (ps);

/* data output */

s->n_spacefree++;
disable (ps);
if (s->wsel_sem)
    ssignal (s->wsel_sem);
restore (ps);

/* errors, exceptions */

if (error_found)
{
    s->error++;
    disable (ps);
    if (s->esel_sem)
        ssignal (s->esel_sem);
    restore (ps);
}
```

ioctl Entry Point

The `ioctl` entry point is called when the ioctl system call is invoked for a particular device. This entry point is used to set certain parameters in the device or obtain information about the state of the device.

Character Drivers

The following example is a shortened version of the `RS232_ioctl()` function, which implements the `ioctl()` entry point in the LynxOS-178 RS-232 device driver, provided here to show what an `ioctl()` entry point looks like:

```
int
RS232_ioctl (RS232_Statics_s *s,
     struct file *f,
     int command,
     char *arg)
{
  uint8_t port;           /* minor device number */
  int retval;             /* return value */
  int ps;                 /* variable to enable and disable interrupts */

  retval = OK;
  port = minor (f->dev); /* get the minor device number */

  /*
     Minor device 0 corresponds to the non-blocking first RS-232 port.
   */
  /*
     Minor device 1 corresponds to the blocking first RS-232 port.
   */
  /*
     Minor device 2 corresponds to the non-blocking second RS-232 port.
```

```
   */
  /*
     Minor device 3 corresponds to the blocking second RS-232 port.
   */

  switch (command)
    {
    case FIOPRIO:             /* Adjust priority tracking */
    case FIOASYNC:            /* FNDELAY or FASYNC flag has changed */
      break;
    case RS232_GET_CONFIG:    /* returns contents of the
                                 configuration register */
    case RS232_GET_PAR_ENABL:
                               /* returns the parity enable status */
    case RS232_GET_PARITY:     /* returns the parity status */
    case RS232_GET_STOP_BITS:  /* returns the number of stop bits */
    case RS232_GET_TEST:       /* returns the test mode status */
    case RS232_SET_CONFIG:     /* set all attributes of the RS232 */
      pseterr (ENOSYS);
      retval = SYSERR;
      break;
    case RS232_SET_PAR_ENABL:   /* sets or clears the parity enable bit */
      if (rbounds ((unsigned long) arg) < sizeof (RS232_ConfigRec_t)) {
        pseterr (EFAULT);
        retval = SYSERR;
      } else {
        /* enable or disable parity for the RS-232 port
           using hardware-specific methods */
      }
      break;
    default:
      pseterr (EINVAL);
      retval = SYSERR;
      break;

    }

  return (retval);
}
```

Block Drivers

Presented below is a skeleton of the `ioctl()` entry point for an abstract block device driver in LynxOS-178.

```
int devioctl(
    struct statics *s,
    int devno,
    int command,
    char *arg)
{
```

```
        /* depending on the command copy relevant
           information to or from the arg structure */
    }
```

The driver defines the meaning of the `command` and `arg` parameters except for when `FIOPRIO` and `FIOASYNC` are used for the command, which are predefined and used by LynxOS-178 to communicate with the drivers. If the `arg` field is to be used as a memory pointer, it should first be checked for validity with either `rbounds` or `wbounds`. The `RS232_ioctl()` function, for instance, does this if the `RS232_SET_PAR_ENABL` command is issued by the user space program making the `ioctl()` system call.

The Kernel uses `FIOPRIO` to signal the change of a task priority to the driver that is doing priority tracking. `FIOASYNC` is invoked when a task invokes the `fcntl()` system call on an open file, setting or changing the value of the `FNDELAY` or `FASYNC` flag. The Kernel might change the priority of an I/O task in the case of *priority inheritance* to elevate the priority of a task that has locked a resource that another higher priority task is blocked on (See “Priority Inheritance Semaphores” on page 50.).

The `ioctl()` entry point should return OK in the case of successful `ioctl` operation or `SYSERR` if either an unknown `ioctl` command was issued or if some other error condition was met during the `ioctl` command. The calling thread’s `errno` can be set using `pseterr()`.

Standard UNIX practice allows all devices to be controlled using `ioctl()`, but the POSIX standard specifies `ioctl()` as only applicable to controlling System V STREAMS devices. Due to that fact, the `posix_devctl()` routine was created for the purpose of controlling non-STREAMS devices.

The `posix_devctl()` routine specified in the POSIX 1003.26 standard is required by the FACE standard, and support for this routine has been included in LynxOS-178.

The `posix_devctl()` routine implements additional functionality beyond what is provided by the `ioctl()` routine. This includes additional arguments that give the size of the buffer passed in the `ddptr` argument (`nbyte`) as well as a pointer (`diptr`) that can be used to pass additional data (an int) back to the caller in addition to the standard pass/fail return value and possible `errno`.

If a device driver does not specifically implement support for `posix_devctl()`, this routine will simply call the `ioctl()` interface to ensure backward compatibility. The additional functionality available using the `posix_devctl()` interface will be ignored in this case.

In order to implement support for the additional functionality available with the posix_devctl() routine, a device driver writer needs to implement a specific ioctl() command (POSIX_DEVCTL_CMD). This command takes a single argument of type struct posix_devctl_args, defined in devctl.h. This structure contains the five arguments passed to the posix_devctl() routine (int fildes, int dcmd, void *ddptr, size_t nbyte, and int *diptr).

The POSIX_DEVCTL_CMD ioctl() command should cause the ioctl command specified in the dcmd field to be executed, passing either the entire posix_devctl_args structure or the individual structure fields as arguments.

The requested command can then implement the full posix_devctl() level of functionality rather than the slightly more limited level of functionality available with the ioctl() interface.

uninstall Entry Point

The uninstall entry point is called when a major device is uninstalled dynamically from the system (via the devinstall command or cdv_uninstall and bdv_uninstall system calls). It is not called by a normal reboot of the system. Interrupt vectors set in the install routine and dynamically-allocated data are freed in the uninstall entry point. The entry point should also perform any necessary actions to put the hardware device into an inactive state. Once the interrupt handler is detached, any further interrupts from the device would cause the system to crash. The format of the uninstall entry point is shown below.

Character Drivers

The LynxOS-178 RS-232 device driver just returns OK from the RS232_uninstall() function, the latter being an implementation of the uninstall entry point in this particular device driver:

```
DRVENTPT int RS232_uninstall(RS232_StaticsPtr_t s)
{
return(OK);
}
```

Block Drivers

This is the general structure of the uninstall entry point of a block device driver:

```
void devuninstall(struct statics *status)
```

```
{
  ...
  /* clear interrupt vectors */
  ...
  ...
}
```

As a rule, if the `install` entry point has allocated some system resources (such as memory for the driver statics structure), these resources are deallocated in the `uninstall` routine. If the `sysbrk()` call was used to allocate memory, the `sysfree()` call is used to deallocate the memory. In addition, such things as waiting for the hardware to become quiescent and optional hardware state clean-up may be carried out by the `uninstall` entry point of a device driver. Refer to the example in Chapter 7, "Interrupt Handling."

Interrupt vectors are cleared here using the `iointclr()` call or DRM's `drm_unregister_isr()`. The return value is ignored.

strategy Entry Point

The `strategy` entry point is valid only for block devices. Instead of a `read` and `write` entry point, block device drivers have a `strategy` routine that handles both read and write. The format of the `strategy` entry routine is as follows:

```
#include <disk.h>
devstrategy(status, bp)
struct statics *status;
struct buf_entry *bp;
{
    /* do read or write depending on whether a read
    or write operation is specified in the buf_entry */
}
```

The structure `buf_entry` is defined in `disk.h` and is reproduced below.

```
struct buf_entry
{
   struct buf_entry *b_forw; /* double link list for owner device */
   struct buf_entry *b_back;
   struct buf_entry *av_forw; /* free list pointers*/
   struct buf_entry *av_back; /* can be used by driver*/
   char *memblk;               /* data block */
   int w_count;
   int rw_count;
   int b_rwsem;               /* block read/write semaphore*/
   int b_error;               /* disk error */
   int b_sem;                 /* block semaphore */
   int b_status;              /* block status */
   int b_device;              /* block device number */
   long b_number;             /* block number */
};
```

The size of the block request is returned by the macro `BLKSIZE(b_device)`. `BLKSIZE` takes as its argument the block device number, `b_device`.

Every block to be read or written has an associated buffer entry. The encoded major and minor device numbers are stored in `b_device`. The memory source or destination is denoted by `memblk`. The source or destination number of the block on the device is `b_number`.

The `b_status` is used to indicate the status and type of transfer. If the value `B_READ` is set, the transfer is a read; otherwise, the transfer is a write.

The transfer status is indicated by `B_DONE` or `B_ERROR`. The `b_error` field is set to nonzero if the transfer fails. The `b_rwsem` semaphore is signaled once the transfer completes. `av_forw` points to the next buffer or `NULL` if the end of the list has been reached. `w_count` is available to the driver for its own purposes. For example, LynxOS-178 drivers use `w_count` for priority tracking.

CHAPTER 3 Memory Management

This chapter describes the LynxOS-178 memory model, supported address types, how to allocate memory, DMA transfers, memory locking, address translation, and how to access user space from interrupt handlers and kernel threads.

LynxOS-178 Virtual Memory Model

LynxOS-178 uses a *Virtual Memory* architecture. All memory addresses generated by the CPU are translated by the hardware Memory Management Unit (MMU).

Each user task has its own protected virtual address space that prevents tasks from inadvertently (or maliciously) interfering with each other. The Kernel, which includes device drivers, and the currently executing user task exist within the same virtual address space — the user task (i.e. the application) is mapped into the lower part and the Kernel into the upper part. During a context switch, only the application's portion of the virtual address space is remapped.

Applications cannot access the portion of the address space occupied by the Kernel and its data structures. The constant OSBASE defines the upper limit of the user accessible space. Addresses above this limit are accessible only by the Kernel.

Kernel code, on the other hand, has access to the entire virtual address space. This greatly facilitates the passing of data between drivers and user tasks. A device driver, as part of the Kernel, can read or write a user address as a direct memory reference without the need to use special functions. However, this also means that it is the device driver developer's responsibility to restrict the access of his or her device driver to only the necessary structures. Kernel code, on the other hand, has access to the entire virtual address space. This greatly facilitates the passing of data between drivers and user tasks. A device driver, as part of the kernel, can read or write a user address as a direct memory reference without the need to use special functions. However, this also means that it is the user’s responsibility as a developer to restrict the access of his or her device driver to only the necessary structures.

LynxOS-178 Address Types

A LynxOS-178 driver supports several different address types. These address types include the following:

- User virtual
- Kernel virtual
- Kernel Direct Mapped
- Physical

In addition, the device driver may have to deal with device addresses such as I/O port addresses, PCI addresses, VME addresses, and so on.

User Virtual

These addresses are passed to the driver from a user application — typically addresses of buffers or data structures used to transfer data or information between the application and a device or the driver. They are valid only when the user task that passed the address is the current task.

Kernel Virtual

These are the addresses of Kernel functions and variables that can be used by a driver. The mapping of Kernel virtual addresses is permanent so they are valid from anywhere within a driver, regardless of the user task currently running. They are not accessible from an application.

Kernel Direct Mapped

This is a region of the Kernel virtual space that is directly mapped to physical memory (that is, contiguous virtual addresses mapped to contiguous physical addresses). The base of this region is defined by the constant `PHYSBASE`, which maps to the start of RAM. The size of the region is platform dependent.

NOTE: Memory that exists on devices or nonsystem buses (for example, PCI and VME) is not accessible via `PHYSBASE`.

Physical

Physical memory is the nontranslated address for memory. A driver will need to set up pointers to physical memory for DMA controllers because they bypass the MMU.

Allocating Memory

The global memory pool includes memory pages available to the system. The total number of system memory pages is platform dependent. The global memory pool is designated at boot time and not associated with any partition.

The following techniques can be used to allocate and free memory:

- `sysbrk(),sysfree()`
- `alloc_cmem(),free_cmem()`
- `get1page(),free1page()`

Each memory allocation will get memory; however, each technique has its advantages and disadvantages.

The sysbrk(), sysfree() Technique

These service calls have they own heap-based memory allocator and use a partition's Kernel heap pool. The memory allocated using these service calls is retrieved from the current partition's Kernel heap pool. The freed memory is returned to the current partition's Kernel heap pool. Therefore, special care should be taken to invoke these calls from the appropriate partition contexts. Each partition's Kernel heap pool starts from zero size and grows transparently as needed. If there is not enough memory in a partition's Kernel heap pool to satisfy an allocation request, additional memory is transferred from the global memory pool (the size is rounded up to be a multiple of a page). Once memory is added to a partition's Kernel pool, it is never returned to the global memory pool.

This technique has the following advantages and disadvantages:

- It is not guaranteed to be physically contiguous.
- It is virtually contiguous.
- It does not return memory to the global memory pool, but can be reused by future `sysbrk()`.

- It returns a value that is aligned on an 8-byte boundary.
- It does not require a physically contiguous memory hole to satisfy the request.

```
char * sysbrk(long size);
void sysfree(char *p, long size);

char *s;
if (!(s = sysbrk(100)))
    return SYSERR;
sysfree (s, 100)
```

The alloc_cmem(), free_cmem() Technique

These service calls allocate memory from the global memory pool.

This technique has the following advantages and disadvantages:

- It is physically contiguous.
- It is virtually contiguous.
- It can return memory to the global memory pool.
- It returns a value that is aligned on a 4k boundary.
- It fails a request if there is no physical contiguous memory hole large enough to satisfy the requests.

```
char *alloc_cmem (int size);
void free_cmem(char *p, int size);

char *s;
if (!(s = alloc_cmem(1000))) return SYSERR;
free_cmem(s, 1000);
```

The get1page(), free1page() Technique

These service calls allocate memory from the global memory pool.

This technique has the following advantages and disadvantages:

- It is physically and virtually contiguous for 1 memory page (4KB in size).
- It can return memory to the global memory pool.
- It returns a value that is aligned on a memory page (4KB) boundary.
- Its memory usage is more complicated than the other two methods.

```
char *get1page();
void free1page(char *addr);

char *s;
    if (!(s = get1page()))
     return SYSERR;
free1page(s);
```

If the amount of memory freed does not correspond to the amount of memory allocated, a Kernel crash will be the most likely result.

Most of the time, the memory usage by the install entry point is negligible and it does not matter which technique is used. There are occasions when vast quantities of memory are needed (e.g. RAM disk memory). In this case, a driver that uses the `sysbrk()` function will permanently decrease the amount of memory in the global memory pool. The `sysbrk()` routine looks through a partition's Kernel heap pool for the requested amount of memory. If it cannot locate enough memory in a partition's Kernel heap pool to fulfill the request, it will transfer enough memory pages from the global memory pool to the partition's Kernel heap pool to fulfill the request. The corresponding `sysfree()` routine will not return any memory to the global memory pool. It will remain in the partition's Kernel heap pool. Therefore when the driver is uninstalled or closed, the memory will not appear in the global memory pool. That memory will only be available to the Kernel or another driver that calls `sysbrk()` at a subsequent time in the context of the same partition.

The two other memory allocation routines, `alloc_cmem()` and `get1page()`, return memory back to the global memory pool when `free_cmem()` and `free1page()` are called, respectively. Using these allocation functions is preferable if memory should be returned to the global memory pool after the memory is freed.

DMA Transfers

The fact that LynxOS-178 uses the CPU's MMU makes the programming of DMA transfers slightly more complicated. All addresses generated by the CPU are treated as virtual and are converted to physical addresses by the MMU. Memory that is contiguous in virtual space can be mapped to noncontiguous physical pages. DMA devices, however, typically work with physical addresses. Therefore, a driver must convert virtual addresses to their corresponding physical addresses before passing them to a DMAcontroller.

Address Translation

A virtual address can be converted to its physical addresses using `get_phys()`.

```
#include <kernel.h>
physaddr = (get_phys (virtaddr) - PHYSBASE + drambase);
```

The address returned is, in fact, a Kernel direct mapped address. To convert this address to its physical address, the constant `PHYSBASE` must be subtracted and `drambase` added. The variable `drambase` contains the physical address of the start of RAM. On most platforms, this will be 0, but for maximum portability, it should be used in the calculation. A driver should never modify the value of `drambase`!

The returned address is valid only up to the next page boundary as contiguous virtual addresses do not necessarily map to contiguous physical addresses. To convert virtual address ranges that cross page boundaries, `mmchain` should be used.

```
#include <mem.h>
#include <kernel.h>

struct dmachain array[NCHAIN];

mmchain (array, virtaddr, nbytes);
for (i = 0; i < nsegments; i++) {
    physaddr = array[i].address - PHYSBASE + drambase;
    length = array[i].count;
    ...
}
```

The virtual memory region is defined by its address (`virtaddr`) and size (`nbytes`). `mmchain()` fills in the fields in the `dmachain` array with the physical addresses and sizes of the physical memory segments that make up the specified virtual address range. The first and last segments in the list may be less than a whole page in size, as illustrated in Figure 3-1.

.

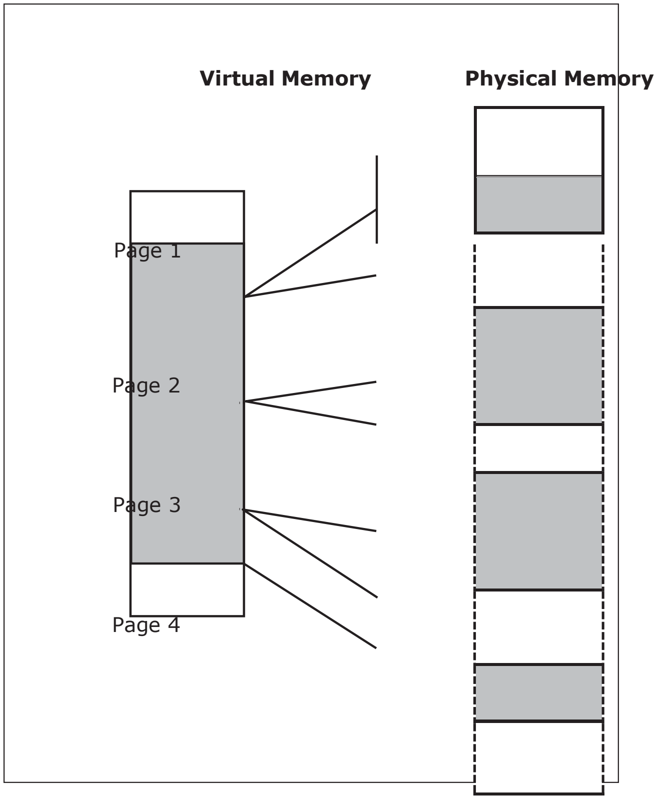


Figure 3-1: Mapping Virtual to Physical Memory

It is the responsibility of the driver to ensure that the `dmachain` array is large enough to hold all the segments that make up the specified address range. The maximum number of segments can be calculated as follows:

```
nsegments = (nbytes + PAGESIZE - 1) / PAGESIZE + 1;
```

`mmchain()` returns the actual number of segments.

Both `get_phys()` and `mmchain()` translate addresses in the context of the current process.

NOTE: As explained in the Kernel Threads section on p.104, kernel threads execute in the virtual memory context of a partition's null process. The `get_phys()` and `mmchain()` routines are therefore available for use by kernel threads as well as threads calling the POSIX driver APIs from userspace processes.)

In the case where no physical memory is mapped to a virtual address, both `get_phys()` and `mmchain()` set the converted address to 0. To translate addresses of a different process, `mmchainjob()` can be used. This takes an additional argument called job, which identifies the pid of the target process.

```
mmchainjob (job, array, virtaddr, nbytes);
```

CHAPTER 4 Synchronization

This chapter describes synchronization issues and the synchronization methods that can be used when writing the device driver.

Introduction

Synchronization is probably the most important aspect of a device driver. It is also the most intricate and therefore the most difficult to master. It requires careful attention and is often the cause of the most frustrating bugs which take hours or days to locate, but only seconds to fix. This aspect of a driver can make or break not only the driver but the whole system. Poorly designed synchronization, while it may not crash the system, can nevertheless have a disastrous effect on the system's real-time performance.

For users new to writing device drivers or unfamiliar with pre-emptive Kernels, the next sections review problems that give rise to the need for synchronization. This discussion is not specific to LynxOS-178, so if the reader is already familiar with the subject, they can skip to the "Synchronization Mechanisms" section on page 44.

Event Synchronization

The most frequent type of synchronization found in device drivers is the need to wait for a device to complete a data transfer or other operation. Another typical event is waiting for a resource, such as a buffer, to become free.

The simplest technique is to use polling, but polling can waste CPU time unnecessarily. However, in some cases, it is unavoidable.

Polling can be used in the following cases:

- Short waits - a few microseconds, where the overhead of an interrupt is not worth setting up.
- Hardware does not have an interrupt capability - forcing the use of polling.

Some disadvantages of polling are as follows:

- CPU time may be wasted.
- When the CPU is switched to a higher priority process, the event that the polling was waiting for could occur and the event could be missed.

The use of interrupts overcomes these drawbacks, but a driver needs a way to wait for an interrupt to occur without using CPU cycles. Kernel semaphores are the mechanism used to allow a driver to be blocked and to be woken up when the event occurs, as illustrated in the Figure 4-1.

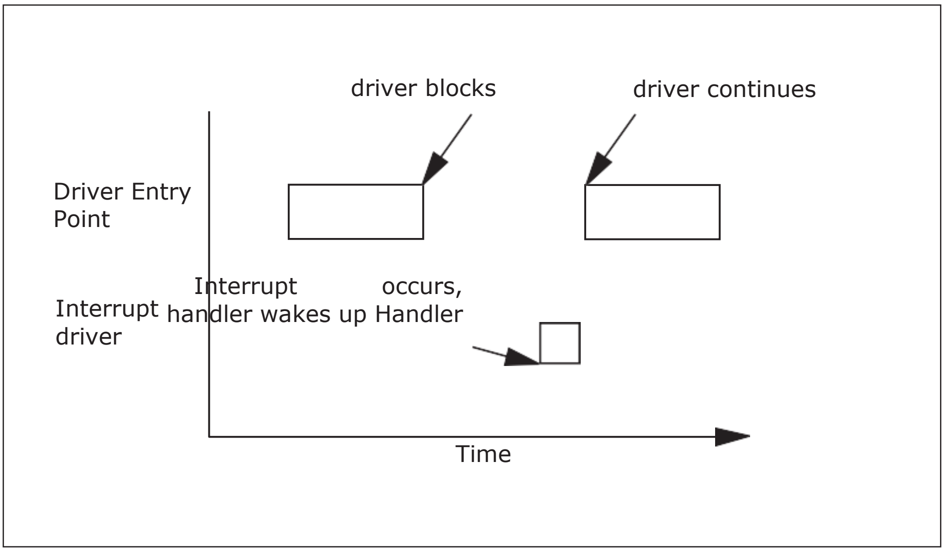


Figure 4-1: Event Synchronization

Critical Code Regions

The LynxOS-178 Kernel, being designed for real-time, is completely pre-emptive including device drivers. This means that one task can pre-empt another, even if the pre-empted task happens to be in the middle of a system call or well into a driver.

Consider the following code, which is an attempt to provide exclusive access to a device:

```
if (device_free == TRUE) {
 device_free = FALSE;
...                  /* use device */
 device_free = TRUE;
```

```
}
```

Imagine a system where two tasks are using the device. The first task executes line 1 of the above code and finds the device to be free. But, before being able to execute line 2, it is pre-empted by task 2, which then executes the same code. The second task also finds the device free (the first task has not executed line 2 yet) and so proceeds to use it. While task 2 is using the device, it is pre-empted by task 1 (maybe because its time ran out). Task 1 continues where it left off and proceeds to access the device as well. So, both tasks are accessing the device concurrently, just the situation that the code was supposed to prevent.

This type of situation is known as a race condition (that is, the result of the computation depends on the timing of task execution). A race condition is a bug. Concurrent accesses to a device may well lead to a system crash. The example also serves to illustrate the insidious nature of this type of problem. The probability that a preemption occurs between lines 1 and 2 is obviously quite low. So, the problem will occur very infrequently, and can be extremely difficult to track down or reproduce systematically.

Code that accesses shared resources, such as a device, is known as a c*ritical code region*. The mechanism used to synchronize access to the code is often referred to as a *mutex* as it provides mutual exclusion.

In order to avoid race conditions, a driver must protect the critical regions of code. There are a number of mechanisms for protecting critical code regions that will be discussed in the following sections.

When considering synchronization, the interrupt handler must also be taken into account. It, too, can pre-empt the current task at any moment and may access static data within the driver. But, because an interrupt handler is not scheduled in the same way as other tasks, the mechanisms used to synchronize with it are usually different from those used to synchronize between user tasks.

Figure 4-2 summarizes the problems of synchronizing critical code regions in a multitasking kernel.

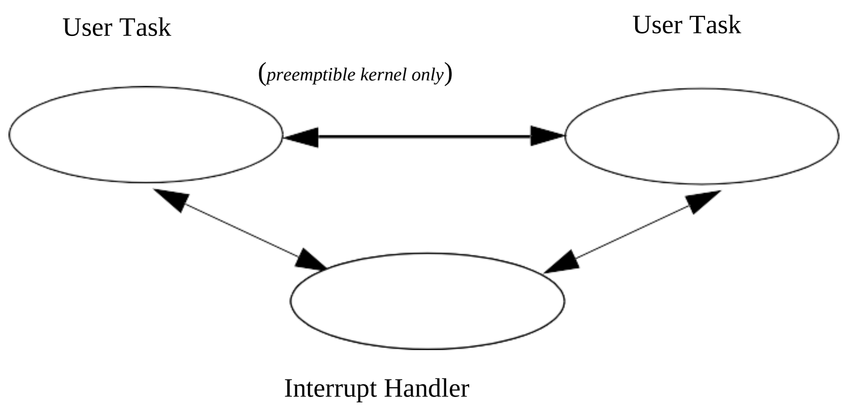


Figure 4-2: Problems in Synchronizing Critical Code Regions in a Multitasking Kernel

Resource Pool Management

A third type of synchronization involves the dynamic management of a shared pool of resources, such as a set of data buffers. A counting semaphore is used to atomically remove items from and return items to the central pool. The remove operation may block if no resources are currently available and the return operation wakes up any tasks waiting for resources.

Synchronization Mechanisms

There are a number of synchronization mechanisms that can be used in a LynxOS-178 device driver:

- Kernel Semaphores
- Disabling Preemption
- Disabling Interrupts

Broadly speaking, we can place the mechanisms on a scale of s*electivity*, which refers to the level of impact the use of the mechanism has on other activities in the system. Kernel semaphores are the most selective and only affect tasks that are using the driver. There is no impact on other nonrelated tasks or activities in the system.

Disabling preemption is obviously less selective as it will have an impact on all tasks running in the system, regardless of whether they are using the driver. Higher priority tasks will be prevented from executing until preemption is reenabled. However, the system will continue to handle interrupts.

Disabling interrupts is the least selective method. Both task preemption and interrupt handling are disabled so the only activity that can occur in the system is the execution of the driver that disabled interrupts.

Which mechanism to use depends on a number of factors. The different synchronization mechanisms and the situations in which each should be used will be discussed in detail below. First, let's describe the mechanisms themselves.

Kernel Semaphores

General Characteristics

These are the most versatile and widely used driver synchronization mechanism. A Kernel semaphore is simply an integer variable that is declared by the driver. All semaphores must be visible in all contexts. This means that the memory for a semaphore must not be allocated on the stack.

There are two types of kernel semaphores in LynxOS-178, a counting semaphore and a Priority Inheritance semaphore. The type of semaphore is defined by the value the semaphore is initialized to.

Counting Semaphores

A counting semaphore has an internal counter. A counting semaphore can be initialized to any nonnegative integer value being not larger than the KSEM_CNT_MAX constant. This is useful to regulate access to multiple resources, with the semaphore value initialized to the number of free resources.

```
sem = 1;
```

After the initial semaphore value is set, the semaphore can be acquired using the `swait()` function.

```
swait (&sem, flag);
```

Note that the address of the semaphore is passed to `swait()`. The `flag` argument is explained below.

If the semaphore value is greater than zero, it is simply decremented and the task continues. If the semaphore value is less than or equal to zero, the task blocks and is put on the semaphore's wait queue. Tasks on this queue are kept in priority order.

A semaphore is signaled using the `ssignal()` function, similar to `swait()`, takes the semaphore address as argument.

```
ssignal (&sem);
```

If there are tasks waiting on the semaphore's queue, the highest priority task is woken up. Otherwise, the semaphore value is incremented.

Kernel semaphores have state. The semaphore's value remembers how many times the semaphore has been waited on or signaled. This is important for event synchronization. If an event occurs but there are no tasks waiting for that event, the fact that the event occurred is not forgotten.

Kernel semaphores are not owned by a particular task. Any task can signal a semaphore, not just the task that acquired it. This is necessary to allow Kernel semaphores to be used as an event synchronization mechanism but requires care when the semaphore is used for mutual exclusion.

Semaphores are the slowest mutual exclusion mechanism available to the driver; however, they incur the penalty only if blocking or waking is involved. If the semaphore is free, the execution time for `swait()` is very fast. Similarly, `ssignal()` incurs significant overhead only if there are tasks blocked on the semaphore.

The `flag` argument to the `swait()` function allows a task to specify how signals are handled while it is blocked on a semaphore. If the task does not block, this argument is not used. There are three possibilities for `flag`, specified using symbolic constants defined in `kernel.h`:

`SEM_SIGIGNORE`	Signals have no effect on the blocked task. Any signals sent to the task while it is waiting on the semaphore remain pending and will be delivered at some future time.
`SEM_SIGRETRY`	Signals are delivered to the task. If the task's signal handler returns, the task automatically waits again on the semaphore. Signal delivery is transparent to the driver as the `swait()` function does not indicate whether any signals were delivered.

`SEM_SIGABORT`	If a signal is sent to the task while it is blocked on the semaphore, the `swait()` is aborted. The task is woken up, and `swait()` returns a nonzero value. The signal remains pending.

Priority Inheritance Semaphores

In a multitasking system that uses a fixed priority scheduler, a problem known as *Priority Inversion* can occur. Consider a situation where a task holds some resource. This task is pre-empted by a higher priority task which requires access to the same resource. The higher priority task must wait until the lower priority task releases the resource. But the lower priority task may be prevented from executing (and, therefore, from releasing the resource) by other tasks of intermediate priority. One solution to this problem is to use *Priority Inheritance* whereby the priority of the task holding the resource is temporarily raised to the priority of the highest priority task waiting for that resource until it releases the resource. LynxOS-178 Kernel semaphores support priority inheritance. In order to function with priority inheritance, the semaphore's value must be initialized by the Kernel function `pi_init()`.

```
pi_init (&s->mutex);
```

The use of this feature is only meaningful in the context of a Kernel semaphore being used as a mutex mechanism.

A Priority Inheritance semaphore can be acquired and signaled using the same functions `swait()` and `ssignal()`.

Other Counting Kernel Semaphore Functions

There are a number of other functions used to manipulate counting kernel semaphores.

`ssignaln(int *sem, int n)`	Used to signal a semaphore *n* times. This is equivalent to calling `ssignal()` *n* times.

`sreset(int *sem)`	Resets the semaphore value to 0 and wakes up all tasks that are waiting on the semaphore.
`scount(int *sem)`	Returns the semaphore value. A negative count indicates the number of tasks that are waiting for the semaphore. This function is rarely used in a driver but if the need arises, it should always be used rather than using the value of the semaphore variable directly. The latter technique would give erroneous results because the value is not a simple count when there are tasks blocked on the semaphore.

The `ssignaln()` and `sreset()` functions can be used only with counting semaphores. The `scount()` function can be used with both types of semaphores. The use of these routines is further illustrated in the examples discussed below. Table 4-1 compares kernel semaphore types.

Table 4-1: Kernel semaphore types comparison

Semaphore	Counting	Priority Inheritance
Variable type	int	int
Initialization procedure	assign semaphore variable to initial value of counter (nonnegative value)	call `pi_init()` function with pointer to semaphore variable
Has a counter	yes	no
Priority Inversion can occur	yes	no
Available methods	`swait()`, `ssignal()`, `ssignaln()`, `sreset()`, `scount()`	`pi_init()`, `swait()`, `ssignal()`,`scount()`

Disabling Preemption

The Kernel routines `sdisable()` and `srestore()` control task preemption and are used in much the same way as `disable()` and `restore()`.

```
int ps;
sdisable (ps);  /* disable task preemption */
    ...
    ...
srestore (ps);  /* restore preemption state */
```

The variable `ps` must be a local variable and should never be modified by the driver. Each call to `sdisable()` must have a corresponding call to `srestore()`, using the same variable. Note that `srestore()` does not necessarily reenable preemption. Rather, it restores the state that existed before `sdisable()` was called. So if preemption was already disabled when `sdisable()` was called, the (first) call to `srestore()` will not reenable it. The Kernel continues to handle interrupts while preemption is disabled.

Care should be taken not to disable preemption for too long as this will delay the inter-VM context switch, affecting VM time slices defined by the inter-VM scheduler. The maximum time for which preemption can be disabled should be less than or equal to the system integrator-defined Jitter Time. For more information, refer to the LynxOS-178 (RSC) Partitioning and RSC Interface Analysis Document (available for purchase as part of the DO-178 Artifacts Package).

Critical Code Regions

If the shared resource is accessed by the interrupt handler, then `disable()/restore()` must be used to protect critical code regions. If not, then any of the synchronization mechanisms can be used.

Using Kernel Semaphores for Mutual Exclusion

Both types of semaphores, a counting semaphore and a Priority Inheritance semaphore, can be used to protect a critical code region. When using a counting semaphore like a mutex, the semaphore's value should be initialized to 1.

```
s->mutex = 1;
```

In order to function with priority inheritance, the semaphore's value should be initialized by the Kernel function `pi_init()` instead.

```
pi_init (&s->mutex);
```

The difference between counting semaphores and Priority Inheritance semaphores is described in section "Kernel Semaphores" on page 39.

This allows the first task to lock the semaphore and enter the region. Other tasks (including a Kernel thread) that attempt to enter the same region will block until the semaphore is unlocked. Each call to `swait()` must have a corresponding call to `ssignal()`.

```
swait (&s->mutex, SEM_SIGIGNORE); /* enter critical
                                      code region */
    ...
    ...                           /* access resource */
    ...
ssignal (&s->mutex);              /* leave critical
                                        code region */
```

Signals can normally be ignored when using a Kernel semaphore as a mutex. Compared to waiting for an I/O device, a critical code region is relatively short so there is little need to be able to interrupt a task that is waiting on the mutex. Unlike an event, which is never guaranteed to occur, execution of a critical code region cannot “fail”. The task holding the mutex is bound, sooner or later, to get to the point where the mutex is released

CAUTION! `sreset()` and `ssignaln()` should never be used on a kernel semaphore that is used for mutual exclusion as in both cases this could lead to more than one task executing the critical code concurrently.

Using Interrupt Disabling for Mutual Exclusion

The `disable()` and `restore()` routines must be used for synchronization with an interrupt handler. If the interrupt handler needs to access critical regions, then it will also have to use `disable()` and `restore()`. This is because interrupts are enabled when the driver is executing the interrupt routine. The reason for this is to allow higher priority interrupts to occur. It is rare, but the interrupt routine can be pre-empted by a higher priority interrupt. This behavior is necessary to minimize delays in servicing events requiring real-time response.

```
disable (ps);
if (ptr = freelist)      /* take item off free list */
    freelist = ptr->next;
restore (ps);

disable (ps);
ptr->next = freelist;    /* put item back on free list */
freelist = ptr;
restore (ps);
```

If a driver allows multiple tasks to use a device concurrently and the critical region is in the part of the driver that will be executed very often, then there may be some advantage in using `disable()` and `restore()` in order to avoid an excessive number of context switches. But if the driver is written so that only one user task can use the device at a time, then `swait()` and `ssignal()` will probably be sufficient for synchronization with the driver’s Kernel thread.

When using `disable()` and `restore()`, it is essential that the time during which interrupts are disabled is kept to a minimum to avoid having an impact on the system's response times. It is sometimes possible to split a seemingly long critical code section into two or more pieces by introducing calls to `restore()` and `disable()` at appropriate points. Care must be taken in selecting the point where interrupts are reenabled to avoid creating a race condition.

```
disable (ps);
    ...
    ...
restore (ps);    /* allow interrupts and pre-emption */

disable (ps);
    ...
    ...
restore (ps);
```

If the critical code region is still too long then the driver should be redesigned so that `swait()`/`ssignal()` can be used and another means found for the interrupt handler to communicate with the rest of the driver.

Using Preemption Disabling for Mutual Exclusion

If the shared resource is not accessed by the interrupt handler, `sdisable()` and `srestore()` can be used instead of `disable()` and `restore()`. This allows the system to continue to handle interrupts during the critical code region. The same remarks and considerations that apply to the use of `disable()` and `restore()` apply equally to `sdisable()` and `srestore()`.

Event Synchronization

A *Kernel Semaphore* is the mechanism used to implement event synchronization in a LynxOS-178 driver. The value of a semaphore used to count pending events should be initialized to 0, indicating that no events have occurred.

Waiting for an event:

```
if (swait (&s->event_sem, SEM_SIGABORT))
{
    pseterr (EINTR);
    return (SYSERR);
}
```

Signaling an event:

```
ssignal (&s->event_sem);
```

Handling Signals

Often times there is no guarantee that an event will occur; therefore, signals should be allowed to abort the `swait()` using `SEM_SIGABORT`. This way, a task can be interrupted if the event it is waiting for never arrives. If signals are ignored, there will be no way to interrupt the task in the case of problems, so the task could remain in the blocked state indefinitely. The driver must check the return code from `swait()` to determine whether a signal was received. As an alternative to `SEM_SIGABORT`, timeouts can be used if the timing of events is known in advance.

It is sometimes useful for an application to be able to handle signals while it is blocked on a semaphore but without aborting the wait. This is possible using the `SEM_SIGRETRY` flag to `swait()`. Signals are delivered to the application and the `swait()` automatically restarted. There is no way for the driver to know whether any signals were delivered while the task was blocked on the semaphore.

A word of caution is necessary concerning the use of `SEM_SIGRETRY`. If the signal handler in the application calls `exit(3)`, then the `swait()` in the driver will never return. This could cause problems if the task had blocked while holding some resources. These resources will never be freed. To avoid this type of problem, a driver can use `SEM_SIGABORT` in conjunction with the Kernel function `deliversigs()`. This allows the application to receive signals in a timely fashion, but without the risk of losing resources in the driver.

```
if (swait (&s->event_sem, SEM_SIGABORT)
{
    cleanup (s); /* tidy up before delivering signals */
    deliversigs (); /* may never return */
}
```

Using sreset with Event Synchronization Semaphores

Two examples of using `sreset()` discussed below are:

- In handling error conditions
- Variable length data transfers (with multiple consumers)

Handling Error Conditions

A driver must handle errors that may occur. For example, what should it do if an unrecoverable error is detected on a device? A frequent approach is to set an error flag and wake up any tasks that are waiting on the device:

```
if (error_found) {
    s->error++;
    sreset (&s->event_sem);
}
```

But now the driver cannot assume that when `swait()` returns, the expected event has occurred. The `swait()` could have been woken up because an error was detected. So some extra logic is required when using the event synchronization semaphore:

```
if (swait (&s->event_sem, SEM_SIGABORT))
{
    pseterr (EINTR);
    return (SYSERR);
}
if (s->error)
{
    pseterr (EIO);
    return (SYSERR);
}
```

Variable Length Transfers

The second example of using `sreset()` is somewhat more esoteric but is an interesting example nevertheless. Imagine the following scenario: A device or “producer” process generates data at a variable rate. Data can also be consumed in variable sized pieces by multiple tasks. At some point, a number of consumer tasks may be blocked on an event synchronization semaphore, each waiting for different amounts of data, as illustrated in Figure 4-3.

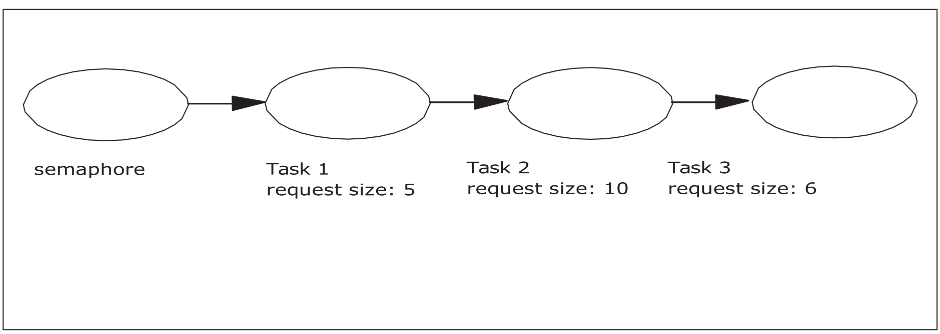


Figure 4-3: Synchronization Mechanisms

When some data becomes available, what should the driver do? Without adding extra complexity and overhead to the driver, there is no easy way for the driver to calculate how many of the waiting tasks it can satisfy (and should, therefore, wake up). A simple solution is to call `sreset()` which will wake all tasks which will then consume the available data according to their priorities. Tasks that are awakened but find no data left will have to wait again on the event semaphore.

Caution when using sreset

To maintain coherency of the semaphore queue, `sreset` must synchronize with calls to `ssignal()`. Because `ssignal()` can be called from an interrupt handler, `sreset()` disables interrupts internally while it is waking up all the blocked tasks. Since the number of tasks blocked on a semaphore is not limited, this could lead to unbounded interrupt disable times if `sreset()` is used without proper consideration.

To avoid this problem, another technique must be used in driver designs where an unknown number of tasks could be blocked on a semaphore. One possibility is to wake tasks in a “cascade”. The call to `sreset()` is replaced by a call to `ssignal()`, which will wake up the first blocked task. This task is then responsible for unblocking the next blocked task, which will unblock the next one, and so on until there are no more blocked tasks. A negative semaphore indicates that there are blocked tasks. This is illustrated in the modified error handling code from the previous section:

```
if (error_found)
{
    s->error++;
    if (s->event_sem < 0)
        ssignal (&s->event_sem);
}
    ...
if (swait (&s->event_sem, SEM_SIGABORT))
{
    pseterr (EINTR);
    return (SYSERR);
}
if (s->error)
{
    if (s->event_sem < 0)
        ssignal (&s->event_sem);
    pseterr (EIO);
    return (SYSERR);
}
```

Tasks are queued on a semaphore in priority order, they will still be awakened and executed in the same order as when using `sreset()`. There is no penalty to using this technique.

Resource Pool Management

LynxOS-178 Kernel semaphores can also be used as counting semaphores for managing a resource pool. The value of the semaphore should be initialized to the number of resources in the pool. To allocate a resource, `swait()` is used. `ssignal()` is used to free a resource. The following code shows an example of using `swait()` to allocate and `ssignal()` to free a resource.

```
struct resource *
allocate
struct statics *s;
{
    struct resource *resource;
    int ps;

    swait (&s->pool_sem, SEM_SIGRETRY);
    sdisable (ps);
    resource = s->pool_freelist;
    s->pool_freelist = resource->next;
    srestore (ps);
    return (resource);
}
free (s, resource)
struct statistics *s;
struct resource *resource;
{
    struct resource *resource;
    int ps;
    sdisable (ps);
    resource->next = s->pool_freelist;
    s->pool_freelist = resource;
    srestore (ps);
    ssignal (&s->pool_sem);
}
```

The counting semaphore functions implicitly as an event synchronization semaphore too. When the pool is empty an attempt to allocate will block until another task frees a resource.

A mutex mechanism is still needed to protect the code that manipulates the free list. The combining of different synchronization techniques is discussed more fully in the following section.

Combining Synchronization Mechanisms

The examples discussed in the preceding sections have all been fairly straightforward in that they have only used one synchronization mechanism. In a real driver the scenarios are often far more complex and require the combining of different techniques. The following sections discuss when and how the synchronization mechanisms should be combined.

Manipulating a Free List

Refer to the code in “Using Interrupt Disabling for Mutual Exclusion” on page 50. This illustrates the use of interrupt disabling to remove an item from a free list, but does not address what the driver should do if the free list is empty.

One possibility is that the driver blocks until another task puts something back on the free list. This scenario requires the use of a mutex and an event synchronization semaphore. Two different approaches to this problem are illustrated in the following examples. The first example is deliberately complicated to demonstrate various synchronization techniques.

```
/* get_item : get item off free list, blocking if
list is empty  */
struct item *
get_item
struct statics *s
{
    struct item *p;
    int ps;

    do
    {
    disable (ps);        /* enter critical code */
    if (p = s->freelist) /* take 1st item on list */
        s->freelist = p->next;
    else
        /* list was empty, so wait */
        swait (&s->freelist_sem, SEM_SIGIGNORE);
    restore (ps);        /* exit critical code */
    } while (!p);
    return (p);
}

/* put_item : put item on free list, wake up waiting tasks */
put_item (s, p)
struct statics *s;
struct item *p;
{
    int ps;

    disable (ps);            /* enter critical code */
    p->next = s->freelist;   /* put item on list */
    s->freelist = p;
    if (scount (&->freelist_sem) < 0)
        ssignal (&s->freelist_sem); /* wake up waiter */
    restore (ps);            /* exit critical code */
}
```

There are a number of points of interest illustrated by this example:

1. The example uses `SEM SIGIGNORE` for simplicity. If `SEM SIGABORT` is used, the return value from `swait()` must be checked.
2. The example uses the `disable()` and `restore()` mechanism for mutual exclusion. This allows the free list to be accessed from an interrupt handler using `put_item()`. `get_item()` should never be called from an interrupt handler, though, as it may block. If the free list is not accessed by the interrupt handler, `sdisable()` and `srestore()` can be used instead.
3. The `get_item()` function uses the value of the item taken off the list to determine if the list was empty or not. Note that the `freelist_sem()` is being used simply as an event synchronization mechanism, not a counting semaphore. (Managing a free list with a counting semaphore is illustrated in the second approach). As a consequence, the code that puts items back on the free list must signal the semaphore *only* if there is a task waiting. Otherwise, if the semaphore was signaled every time an item is put back, the semaphore count would become positive and a task calling `swait()` in `get_item()` would return immediately, even though the list is still empty.
4. Blocking with interrupts disabled may seem at first sight like a dangerous thing to do. But this is necessary as restoring interrupts before the `swait()` would introduce a race condition. LynxOS-178 saves the interrupt state on a *per task* basis. So, when this task blocks and the scheduler switches to another task, the interrupt state will be set to that associated with the new task. But, from the point of view of the task executing the above code, the `swait()` executes atomically with interrupts disabled.
5. `swait()` and `ssignal()` cannot be used as the mutex mechanism in this particular example as this could lead to a deadlock situation where one task is blocked in the `swait()` while holding the mutex. Other tasks wishing to put items back on the list will not be able to enter the critical region. If a critical code region may block, care must be taken not to introduce the possibility of a deadlock. To avoid a deadlock, `sdisable()` and `srestore()` or `disable()` and `restore()` should be used as the mutex mechanism rather than `swait()` and `ssignal()`. But, once again, the critical code region must be kept as short as possible to avoid having any adverse effect on the system's real-time responsiveness. An alternative would be to raise an error condition if the list is empty rather than block. This would allow `swait()` and `ssignal()` to be used as the mutex mechanism.

6. A call to `ssignal()` in `put_item()` may make a higher priority task eligible to execute but the context switch won't occur until preemption is reenabled with `restore()`.

In the second approach to this problem, a kernel semaphore is used as a counting semaphore to manage items on the free list. The value of the semaphore should be initialized to the number of items on the list.

```
struct item *
get_item (s)
struct statics *s;
{
    struct item *p;
    int ps;

    swait (&s->free_count, SEM_SIGRETRY);
    disable (ps);
    p = s->freelist;
    s->freelist = p->next;
    restore (ps);
    return (p);
}

put_item (s, p)
struct statics *s;
struct item *p;
{
    int ps;

    disable (ps);
    p->next = s->freelist;
    s->freelist = p;
    restore (ps);
    ssignal (&s->free_count);
}
```

This code illustrates the following points:

1. A Kernel semaphore used as a counting semaphore incorporates the functionality of an event synchronization semaphore. `swait()` will block when no items are available and `ssignal()` will wake up waiting tasks.

2. The example uses the `disable()` and `restore()` mechanism for mutual exclusion. This allows the free list to be accessed from an interrupt handler using `put_item()`. `get_item()` should never be called from an interrupt handler, though, as it may block. If the free list is not accessed by the interrupt handler, `sdisable()` and `srestore()` can be used instead.

3. The event synchronization is outside of the critical code region so there is no possibility of deadlock. Therefore, `swait()` and `ssignal()` could

be used as the mutex mechanism if the code does not need to be called from an interrupt handler.

4. The function `put_item()` could be modified to allow several items to be put back on the list using `ssignaln()`. But items can only be consumed one at time, since there is no function `swaitn()`.

Signal Handling and Real-Time Response

"Handling Signals" on page 46 discussed the use of the `SEM_SIGRETRY` flag with `swait()`. It is not advisable to use `swait()` with this flag inside a critical code region protected with `(s)disable()` and `(s)restore()`. The reason for this is that, internally, `swait()` calls the Kernel function `deliversigs()` to deliver signals when the `SEM_SIGRETRY` flag is used. If the `swait()` is within a region with interrupts or preemption disabled, then the execution time for `deliversigs()` will contribute to the total interrupt or preemption disable time, as illustrated in the following example:

```
sdisable (ps);      /* enter critical region */
    ...
swait (&s->event_sem, SEM_SIGRETRY);
                   /* may call deliversigs internally */
    ...
srestore (ps);      /* leave critical region */
```

In order to minimize the disable times it is better to use `SEM_SIGABORT` and reenable interrupts/preemption before calling `deliversigs()`. The above code then becomes:

```
sdisable (ps);    /* enter critical region */
    ...
while (swait (&s->event_sem, SEM_SIGABORT))
{
    srestore (ps);  /* re-enable preemption
                         before delivering signals */
    deliversigs (); /* may never return */
    sdisable (ps);
}
    ...
srestore (ps);     /* leave critical region */
```

Nesting Critical Regions

It is also possible to nest critical regions. As a general rule, a less selective mechanism can be nested inside a more selective one. For instance, the following is permissible:

```
int sps, ps;
sdisable (sps);
```

```
    ...
disable (ps);
    ...
restore (ps);
    ...
srestore (sps);
```

Note that different local variables must be used for the two mechanisms. However, the converse is not true. It is not permitted to do the following:

```
disable (ps);
    ...
sdisable (sps);
    ...
srestore (sps);
    ...
restore (ps);
```

In any case, the inner `sdisable()` and `srestore()` is completely redundant, as preemption is already disabled by the outer `disable()` and the BKL is already locked by the outer `disable()`.

A spin lock critical region may be nested inside a kernel semaphore/mutex critical region, but not vice versa. As already mentioned, nesting spin lock critical regions within other spin lock critical regions is possible, but not recommended. A spin lock critical region should always be the innermost critical region.

Kernel semaphore/mutex critical regions can be nested within other kernel semaphore/mutex critical regions. Pay attention to keeping the order of semaphore/mutex locking constant; a varying locking order may lead to deadlocks.

Inter-VM Synchronization

In some cases it may be necessary to synchronize access to a Driver Logical Resource between different VMs. This can be achieved by one of the following mechanisms:

- Interrupt disabling. When the interrupts are disabled, the inter-VM context switch is also disabled.
- Preemption disabling. When preemption is disabled, the inter-VM context switch is also disabled (This behavior could be changed in subsequent releases).
- Counting semaphores. Priority inheritance semaphores are not allowed. Use of this mechanism allows a situation where a thread in one VM is

blocked on a semaphore owned by a thread in another VM. This kind of interaction between partitions must be analyzed and proved to be acceptable for the particular environment.

For more information, refer to the LynxOS-178 (RSC) Partitioning and RSC Interface Analysis Document (available for purchase as part of the DO-178 Artifacts Package).

CHAPTER 5 Accessing Hardware

Device drivers provide the ability to synchronize access and control of the operating system to the hardware. This chapter describes how device drivers access the hardware layer and illustrates the virtual memory mappings used by LynxOS-178 on different hardware platforms.

General Tasks

Synchronization

Lynx Software Technologies recommends protecting the access to device registers by disabling interrupts. For more information, refer to Chapter 7, “Interrupt Handling” on page 75.

Handling Bus Errors

An access to a memory location where no device is responding may cause the hardware to generate a bus error. By default, LynxOS-178 will halt, printing some diagnostic information on the debug port, when a bus error occurs. To change this behavior, a driver can catch bus errors using the `recoset`/`noreco` routines. These routines should surround the code that could potentially cause a bus error.

```
if (recoset ())
{
    pseterr (EFAULT);     /* a bus error occurred */
    noreco();             /* must restore before leaving */
    return (SYSERR);
}
...                       /* access device */
noreco ();                 /* restore default behavior */
```

When first called, recoset returns a zero value. If a bus error occurs subsequently, the Kernel does a nonlocal jump so that the execution flow resumes again where recoset returns, this time with a nonzero value. noreco restores the default system behavior. An exception to this general scheme is the driver install entry point that is discussed in Chapter 2, “Driver Structure” on page 9.

Probing for Devices

It is common for a driver to test for the presence of a device during the `install` entry point. For this reason, LynxOS-178 handles bus errors during execution of the install routine to relieve the driver of this responsibility. If a bus error occurs, the Kernel does not return to the install routine from the bus error handler. The error represents that the device is not present and user tasks will not be permitted to open it.

Device Access

The following sections contain platform-specific information about hardware device access from LynxOS-178. Each section contains memory map figures to illustrate the mapping of LynxOS-178 virtual memory to the hardware device.

In general, the kernel has permissions to access the full range of virtual memory while the user processes have restricted access. Table 5-1 shows a generalization of this concept. Keep this in mind when viewing the memory maps.

Table 5-1: Virtual Memory Access to User Processes

LynxOS Virtual Memory Area	Permissions
OSBASE and above	Kernel only; no user access
SPECPAGE	Read-only to user
Kernel Stack	Read-only to user
Shared Memory	Depends on mapping
User Stack	Read-write to user
User Data	Read-write to user
User Text	Read-only to user

Device Resource Manager (DRM)

Device Resource Manager is a LynxOS-178 module which manages device resources. The DRM assists device drivers in identifying and setting up devices, as well as accessing and managing the device resource space. Developers of new device drivers should use the DRM for managing PCI devices. Chapter 10, “Device Resource Manager” describes the DRM in detail.

CHAPTER 6 Elementary Device Drivers

This chapter discusses the details of some simple device drivers that do everything in a polled fashion. They are efficient in the sense that they do not interfere with the real-time response of the system.

It is useful to see a comparison between these drivers and their more efficient interrupt-based and thread-based implementations in order to understand the performance advantages of one over the other.

Character Device Drivers

The Production Mode RS-232 driver (the driver for dual ports 16550-compatible UART device) is taken as elementary. This driver does not use interrupts. Instead, it polls the port to check if the previous character was put on the port.

Device Information Definition

```
/* rs232info.h */
typedef struct RS232_InfoRec_s {
    kaddr_t         port1;                  /* Com1 */
    kaddr_t         port2;                  /* Com2 */
    unsigned long   default_config_reg;     /* Default configuration */
    unsigned int    channels;               /* Minor devices */
    unsigned int    time_out;               /* Time-out for polling loop */
    unsigned int    speed1;                 /* Com1 baud rate */
    unsigned int    speed2;                 /* Com2 baud rate */
} RS232_InfoRec_t, *RS232_InfoPtr_t;
```

The device information definition for the RS-232 driver contains the port addresses as well as the polling loop time-out value.

Device Information Declaration

```
/* rs232info.c */

#include <rs232info.h>      /* RS232_InfoRec_t */
#include <baudrate.h>       /* B115200 */

/* Device information structure for RS232 driver */
RS232_InfoRec_t rs232info0 = {
    0,
    0,
    0x0400,
    CHAN_ENABLE(0) | CHAN_ENABLE(1),
    500000,
    B115200,
    B115200
};
```

The device information declaration assigns the value of 500000 to the timeout for the blocking driver operations and sets baud rates for both ports.

NOTE: The `IOBASE` region is used to map the memory-mapped device registers in. It is assumed, therefore, that the `IOBASE` area is already mapped in by the BSP.

The `rs232info.c` file is compiled and executed as a standalone program to create a data file to be passed to the `install` routine during dynamic installation of the RS-232 driver.

Declaration for ioctl

```
/* rs232ioctl.h -- various constants for the ioctl entry point */

#define RS232_SET_BAUD 415

typedef struct {
   ...
   /* Other configuration parameters. */
   ...
   RS232_BaudRate_t   RS232_BaudRate;
} RS232_ConfigRec_t, RS232_ConfigPtr_t;

typedef enum {
    RS232_OFF = 0,
    RS232_9600 = 1,
    RS232_19200 = 2,
    RS232_38400 = 3,
    RS232_57600 = 4,
    RS232_115200 = 5
} RS232_BaudRate_t;
```

The `ioctl` routine in this driver provides a means to set up the port baud rate of the port of the UART device. Thus, the user can issue the `ioctl()` system call with the appropriate command (that is, `RS232_SET_BAUD`) and a pointer to the structure `RS232_ConfigRec_t`. The port will be configured in accordance with the values given in the passed structure.

Hardware Declaration

```
/* tty8250.h */

#define U_THR 0x00 /* WR: transmit buffer (LCR_DLAB=0) */
#define U_DLL 0x00 /* WR: divisor latch LSB (LCR_DLAB=1) */
#define U_DLM 0x01 /* WR: divisor latch MSB (LCR_DLAB=1) */
#define U_LCR 0x03 /* WR: line control register */
#define U_LSR 0x05 /* RD: line status register */

/* line control register (read/write) */
#define LCR_DLAB 0x80 /* divisor latch access bit */

/* line status register (read/write) */
#define LSR_TRE 0x20 /* transmit register empty */

/* baud rates using 1.8432 MHZ clock */
#define CILL -1
#define C0 0
#define C9600 12
#define C19200 6
#define C38400 3
#define C57600 2
#define C115200 1
```

The UART port registers used by the driver are described in the `tty8250.h` header file. There are the registers to output characters, configure the port, and check the port status.

Driver Source Code

```
#include <stdbool.h>            /* for bool type */
#include <common_dd_api.h>  /* for common device driver ioctl commands */
#include <kernel.h>                     /* for SYSERR, OK, etc. */
#include <errno.h>                      /* for POSIX error codes */
#include <sys/file.h>           /* for struct sel, FREAD, etc. */
#include <sys/ioctl.h>          /* for FIOPRIO, FIOASYNC, etc. */
#include <kern_proto.h>         /* for scount */

#if defined(__powerpc__)
#include <port_ops_ppc.h>
#elif defined(__x86__)
#include <port_ops_x86.h>
#else
#error Unsupported platform.
#endif

#include <tty8250.h>
#if !defined(NODL)
#include <dldd.h>                       /* Dynamic Device Drivers */
#endif
#include <rs232info.h>          /* RS-232 Device Information File */
#include <rs232ioctl.h>         /* RS-232 ioctl commands */
#include <environ.h>            /* kgetenv() */

#include <bsp_device.h>
#if defined(NODL)
#define DRVENTPT extern
#else
#define DRVENTPT static
#endif
```

```
/*-----------------------MACRO DEFINITIONS------------------------------
*/
#define UART_DIVISOR(bus_freq, baud_rate)       \
        (((bus_freq) * (1000000) * 2 + 32 * (baud_rate)) / (32 *
(baud_rate) * 2))

/* calculates the bit mask for the configure register */
#define NOTUSED   -1  /* RS-232 has not been opened */

#define RS232A_NONBLCKING 0  /* First RS-232 non-blocking device node */
#define RS232A_BLCKING 1     /* First RS-232 blocking device node */
#define RS232B_NONBLCKING 2  /* Second RS-232 non-blocking device node */
#define RS232B_BLCKING 3     /* Second RS-232 blocking device node */

#define RS232A_NODE 0        /* first RS-232 port used by install */
#define RS232B_NODE 1        /* second RS-232 port used by install */
```

Statics Structure

```
/* rs232drvr.c driver statics */

typedef struct RS232_Statics_s
{
  RS232_InfoPtr_t info; /* pointer to the RS232 info structure */
  int   write_sem_A;    /* write semaphore for port A */
  int   write_sem_B;    /* write semaphore for port B */
  int   read_sem_A;     /* read semaphore for port A */
  int   read_sem_B;     /* read semaphore for port B */
  uid_t portb_owner;    /* user ID of the owner for the second RS-232 port
*/
  int   port_disableA;  /* These variables are used to remember */
  int   port_disableB;  /* which port is disabled. */
} RS232_Statics_t, *RS232_StaticsPtr_t;
```

The statics structure is defined as shown. The `info` structure passed through the device information structure is stored in the structure. The variables `write_sem_A` and `write_sem_B` are semaphores used to prevent more than one process from writing to the ports A and B, respectively.

install Routine

```
/* re232drvr.c */
DRVENTPT char*
RS232_install(RS232_InfoPtr_t info)
{
        RS232_StaticsPtr_t      s;      /* statics structure */
        dev_info_t *uart0, *uart1;
        /* Allocate statics */
        s = (RS232_StaticsPtr_t)sysbrk(sizeof(RS232_Statics_t));

        if (s == NULL) {
                pseterr(ENOMEM);
                return (char *)SYSERR;
        }

        uart0 = bsp_get_dev_by_type(0, UART_DEV_TYPE);
        uart1 = bsp_get_dev_by_type(1, UART_DEV_TYPE);

        /* Clean statics */
        bzero((char *)s, sizeof(RS232_Statics_t));
```

```
        /* Update info */
        info->port1                = uart0->vaddr;
        info->port2                = uart1->vaddr;

        /* Set UART1/UART2 Baud Rates */
        if (RS232_setbaud(info->port1, info->speed1) != OK
                || RS232_setbaud(info->port2, info->speed2) != OK) {
                pseterr(EINVAL);
                return (char *)SYSERR;
        }

        /* Set UART2 Modem Control Register */
        if (__inb(info->port2 + U_MSR) & MSR_CTS) {
                __outb(info->port2 + U_MCR, MCR_RTS);
        }

        /* Read UART2 Line Status and Receiver Buffer Registers */
        (void)__inb(info->port2 + U_LSR);
        (void)__inb(info->port2 + U_RBR);

        /* Set UART2 Line Control Register */
        __outb(info->port2 + U_LCR, LCR_WL8);

        /* Set UART2 FIFO Control Register */
        __outb(info->port2 + U_FCR, FCR_EN);

        /* Fill-in statics */
        s->portb_owner = NOTUSED;
        s->info = info;
        s->info->time_out = tickspersec * s->info->time_out / 1000;

        pi_init(&s->write_sem_A);
        pi_init(&s->write_sem_B);

        pi_init(&s->read_sem_A);
        pi_init(&s->read_sem_B);

        return (char *)s;
}
```

The `install` routine allocates the statics structure and initializes it. It returns `SYSERR` with `errno` set to `ENOMEM` if the `static` structure allocation failed. Otherwise, it stores the pointer to the `info` structure and initializes semaphores for both ports.

uninstall Routine

```
DRVENTPT int RS232_uninstall(RS232_StaticsPtr_t s)
{
    return OK;
}
```

The `uninstall` routine does nothing.

open Routine

```
DRVENTPT int RS232_open(RS232_StaticsPtr_t s, int dev, struct file *f)
{
   int          retval;      /* return value OK or SYSERR */
   uid_t        current_vm;  /* current user id */
   uint8_t      minor_dev;   /* minor device */
   int          ps;          /* variable for interrupt disabling
                                and restoring
                             */
   retval = OK; /* return value */

   /*
      Get minor device number.
      Based on the minor device number open the specified port.
      Minor devices 0 and 1 correspond to the first RS-232 port.
      Minor devices 2 and 3 correspond to the second RS-232 port.
    */

   /* Calculate minor device number we are trying to open. */
      minor_dev = minor(f->dev);

   if(minor_dev > RS232B_BLCKING)
   {
      pseterr(EINVAL);
      retval = SYSERR;
   }
   else
   {
      if(minor_dev > RS232A_BLCKING)
   {
      /* check to see if the RS-232 Port B has already been claimed by
      another VM
      */
      /* if it has set errno to EBUSY and return SYSERR */
      /* else claim the port */

      current_vm = getuid();
      disable(ps);
      if(s->portb_owner == NOTUSED)
      {
           s->portb_owner = current_vm;
      }
      restore(ps);
      if((current_vm != s->portb_owner) && (current_vm != 0))
      {
           pseterr(EBUSY);
           retval = SYSERR;
      }
   }
}
   return(retval);
}
```

The `open` routine first checks the minor device number. If it is not valid, it returns an error.

close Routine

```
DRVENTPT int RS232_close(
  RS232_StaticsPtr_t s,        /* device statics structure pointer */
  struct file *f               /* device node information */
)
{
  int      retval;      /* return value OK or SYSERR */
  uint8_t  minor_dev;   /* minor device */
  int      ps;          /* variable for interrupt disabling and restoring
*/

  retval = OK;

  /* calculate minor device number we are trying to close */
  minor_dev = minor(f->dev);

  if(minor_dev > RS232A_BLCKING)
  {
     disable(ps);
     s->portb_owner = NOTUSED;
     restore(ps);
  }
  return(retval);
}
```

The `close` routine clears port owner field if it is required.

Auxiliary Routines

```
static int RS232TimeOut(kaddr_t reg, uint32_t time_out)
{
    bool PortTimeOut = false;
    bool Full;
    uint32_t PortLimit;
    int retval = OK;
    if ( port_disable & (1 << ((reg >> 8) & 1)) ) {
         return SYSERR;
    }

    PortLimit = Uptime + time_out;

    do {
        PortTimeOut = (Uptime >= PortLimit);
        Full =  !(__inb(reg) & LSR_TRE);

    } while(Full && !PortTimeOut);

    if(Full) {
        retval = SYSERR;
    }

    return(retval);
}
```

The `RS232TimeOut` function checks the status of the UART port transmit register for the specified time (measured in the ticks). If the port becomes available for writing (empty) within that time, the function returns success. Otherwise, it returns an error. Note that the `Uptime` variable is a global variable that represents the current tick count.

```
static int RS232ABlockWrite(RS232_StaticsPtr_t s, kaddr_t trreg,
                            kaddr_t streg, uint32_t time_out,
                            char *buffer, int count)
{
    int bytes = 0;

    swait(&s->write_sem_A,SEM_SIGIGNORE);

    while(count > 0)
    {
        /*
          Wait until there is room to write information into the fifo.
         */

        /* Re-init the time-out value */
        if((RS232TimeOut(streg,time_out)) != OK)
        {
            bytes = SYSERR;
            break;
        }

        /* no errors occurred so write the byte to the fifo buffer */
        __outb(trreg + U_THR, *buffer);

        buffer++;
        bytes++;
        count--;
    }

    ssignal(&s->write_sem_A);

    return(bytes);
}
```

The `RS232ABlockWrite` function locks a semaphore at the beginning to ensure the routine is reentrant. Then the function retrieves a character from the buffer and puts it in the UART port transmit register in the loop. It uses the `RS232TimeOut` function to check availability on the port transmit register. If a timeout happens in the `RS232TimeOut` function, then an error is returned.

The blocking write routine for the second RS-232 port that is called `RS232BBlockWrite` is designed in a similar way.

```
static int RS232ANonBlockWrite(RS232_StaticsPtr_t s,
                               kaddr_t trreg, kaddr_t streg,
                               char *buffer, int count)
{
     int bytes = 0;
     int ps;

     if ( port_disable & (1 << ((trreg >> 8) & 1)) ) {
            return bytes;
     }

if(RS232ANonBlockingLock(s))
{
     disable(Mps);

     while(((*(trreg + U_LSR)) & LSR_TRE)
            && (bytes < count))
     {
```

```
            /*
                    While true write a byte to the rs232 transmit register
                    increment the buffer passed as a parameter
                    check if there was a time-out
                    increment the number of bytes written to the port.
            */
            _ outb(trreg + U_THR, *buffer);
            restore(ps);
            buffer++;
            bytes++;
            disable(ps);
        }

            restore(ps);
            ssignal(&s->write_sem_A);
        }

        return(bytes);/* bytes written to the port. */
    }
```

While the transmitter is enabled and bytes to be sent remain, the `RS232ANonBlockWrite()` routine writes the characters from the `buffer` to the UART transmit register after a successful call to yet another auxiliary routine `RS232ANonBlockingLock()`, the latter locking the RS-232 device. The work with the hardware device is being done with the interrupts disabled, which is provided by the pair of calls to the `disable()/restore()` functions. Finally, the write semaphore is signaled to indicate the end of the non-blocking write operations. The function `RS232BNonBlockWrite` is implemented in a similar way.

```
static int RS232ANonBlockingLock(RS232_StaticsPtr_t s)
{
    bool didLock = false;
    int   ps;

    /* attempt to lock semaphore atomically */

    disable(ps);
    if (scount(&s->write_sem_A) > 0)
    {
        swait(&s->write_sem_A, SEM_SIGIGNORE);
        didLock = true;
    }
    restore(ps);

    return (didLock);
}
```

If the write semaphore can be `swait()`'ed on at least once without having to wait for this semaphore to be signaled, the function acquires the semaphore and returns true. If the function has to wait on the write semaphore, no acquisition takes place and a false value is returned immediately (thus, the nonblocking nature of this routine).

There is another nonblocking lock routine that is designed in a similar manner, but works with the second port. It is called `RS232BNonBlockingLock()`.

```
static int
RS232_setbaud(int com_port, int baud) {
        int                             ps;
        char                    c;
        unsigned long   bus_freq;
        unsigned long   div;
        unsigned long   speed;

        /* Calculate speed */
        switch (baud) {
        case B9600:
                speed = 9600;
                break;
        case B19200:
                speed = 19200;
                break;
        case B38400:
                speed = 38400;
                break;
        case B57600:
                speed = 57600;
                break;
        case B115200:
                speed = 115200;
                break;
        case B230400:
                speed = 230400;
                break;
        default:
                return SYSERR;
                break;
        }

        dev_info_t *platform = bsp_get_dev_by_type(0, PLATFORM_DEV_TYPE);
        /* Get Core Complex Bus (CCB) clock */
        bus_freq = platform->u.p.bus_freq;

        /* Calculate divisor */
        div = UART_DIVISOR(bus_freq, speed);

        /* Disable interrupts */
        disable(ps);

        /* Save Line Control Register */
        c = __inb(com_port + U_LCR);

        /* Select "Divisor Latch Access" bit */
        __outb(com_port + U_LCR, c | LCR_DLAB);

        /* Write Low Divisor */
        __outb(com_port + U_DLL, div % 256);

        /* Write High Divisor */
        __outb(com_port + U_DLM, div / 256);

        /* Restore Line Control Register */
        __outb(com_port + U_LCR, c);

        /* Restore Line Control Register */
        restore(ps);

        return OK;
}
```

The `RS232_setbaud()` function, which serves as a helper routine for the `ioctl` entry point of the RS-232 driver, sets the UART port baud rate to the value requested at the call to this routine by writing to the UART registers, such as the line control register and the divisor latch. Note that the interrupts are disabled for the period when the important hardware operations are being done.

There is another non-blocking lock routine that is designed in a similar manner, but works with the second port. It is called `RS232BNonBlockingLock()`.

read Routine

```
DRVENTPT int RS232_read(RS232_StaticsPtr_t s, struct file *f,
               char *buffer, int count)
{
    /* Reading from the RS-232 ports is not implemented. */
    pseterr(ENOSYS);
    return(SYSERR);
}
```

The `read` routine returns the `ENOSYS` error to indicate that this operation is not supported by the RS-232 driver.

write Routine

```
DRVENTPT int RS232_write(RS232_StaticsPtr_t s, struct file *f,
                char *buffer, int count)
{
    int bytes = 0;
    uint8_t port;           /* minor device number */
    kaddr_t trreg;
    kaddr_t streg;
    uint32_t TimeOut;       /* variable to store time-out limit */

    port = minor(f->dev);

/* Minor device 0 is the device node for the non-blocking RS-232 portA. */
/* Minor device 1 is the device node for the blocking RS-232 portA. */
/* Minor device 2 is the device node for the non-blocking RS-232 portB. */
/* Minor device 3 is the device node for the blocking RS-232 portB. */

        if(port == RS232A_NONBLCKING)
        {
            /* Non-blocking case first RS-232 port. */
            trreg = s->info->port1;
            streg = s->info->port1 + U_LSR;
            bytes = RS232ANonBlockWrite(s,trreg,streg,buffer,count);
        }
        else if(port == RS232A_BLCKING)
        {
            /* Blocking first RS-232 port. */

            /*
               Initializes the temp variables to the register addresses
```

```
                in the device info file.
            */
            trreg = s->info->port1;
            streg = s->info->port1 + U_LSR;

            /*
               Initializes the time-out value to the value in the dev info
               file.
             */

        TimeOut = s->info->time_out;

        if((bytes = RS232ABlockWrite(s,trreg,streg,TimeOut,buffer,count))
                < 0)
        {
             /*
                 Time-out occurred so set errno to EIO break
                 out of the loop.
              */
             pseterr(EIO);
             bytes = SYSERR;
        }
    }
    else if(port == RS232B_NONBLCKING)
    {
        /* Non-blocking case for the second RS-232 port. */

        /* Initializes temp variables to values in the device info file. */
        trreg = s->info->port2;
        streg = s->info->port2 + U_LSR;
        bytes = RS232BNonBlockWrite(s, trreg, streg, buffer, count);
    }
    else
    {
        /* Blocking case for the second RS-232 port. */
        /* Initialize register address variables. */

        trreg = s->info->port2;
        streg = s->info->port2 + U_LSR;

        /* Initialize time-out limit. */

        TimeOut = s->info->time_out;

        /* Lock the second RS-232 port for writing. */

        if((bytes = RS232BBlockWrite(s,trreg,streg,TimeOut,buffer,count))
            < 0)
        {
           /* Time-out occurred so set errno to EIO break out of the loop.
*/
             pseterr(EIO);
             bytes = SYSERR;
        }
    }

    return(bytes);
}
```

The `write` function first gets the minor number of the device. Depending on the minor device number, which determines the specific serial port (A or B) and the type of write operation (blocked or nonblocked), the `RS232_write()` routine calls `RS232ANonBlockWrite()`, `RS232ABlockWrite()`, `RS232BNonBlockWrite()`, or `RS232BBlockWrite()` to actually write the data.

```
int RS232_select(
    RS232_StaticsPtr_t s,     /* device statics structure pointer */
    struct file *f,           /* device node information */
    int which,                /* SREAD,SWRITE, or SEXCEPT */
    struct sel *ffs           /* kernel's ff->iosem and ffs->sel_sem */
)
{
    /* Select entry point is not implemented for the RS-232 ports. */
    pseterr(ENOSYS);
    return(SYSERR);
}
```

The select entry point is not implemented in this driver. The function `RS232_select()` simply sets the `ENOSYS` error to indicate that this operation is not supported by the RS-232 driver.

ioctl Routine

```
int RS232_ioctl(RS232_StaticsPtr_t s, struct file *f,
                int command, char *arg)
{
    uint8_t  port;    /* minor device number */
    int      retval;  /* return value */
    int      ps;      /* variable to enable and disable interrupts */

    retval = OK;
    port = minor(f->dev);  /* get the minor device number */

    /*
       Minor device 0 corresponds to the non-blocking first RS-232 port.
       Minor device 1 corresponds to the blocking first RS-232 port.
       Minor device 2 corresponds to the non-blocking second RS-232 port.
       Minor device 3 corresponds to the blocking second RS-232 port.
     */
    switch(command)
    {
        case FIOPRIO:  /* Adjust priority tracking. */
        case FIOASYNC: /* FNDELAY or FASYNC flag has changed. */
        break;

        case RS232_SET_TEST: /* Sets test mode to ENABLED or DISABLED. */
        {
            if(rbounds((unsigned long)arg) < sizeof(RS232_ConfigRec_t))
            {
                pseterr(EFAULT);
                retval = SYSERR;
            }
        else
        {


            int com_port =
               port > RS232A_BLCKING ? s->info->port2:s->info->port1;
```

```
            {
                case DISABLED:
                {
                    disable(ps);
                    __port_andb(com_port+U_MCR, ~MCR_LOOP);
                    restore(ps);
                    break;
                }
                case ENABLED:
                {
                    disable(ps);
                    __port_orb(com_port+U_MCR, MCR_LOOP);
                    restore(ps);
                    break;
                }
                default:
                {
                    pseterr(EINVAL);
                    retval = SYSERR;
                    break;
                }
            }
        }
        break;
    }
        default:
        {
            pseterr(EINVAL);
            retval = SYSERR;
            break;
        }
    }

    return(retval);
}
```

The `ioctl` routine operates based on the command issued by the user.

If the command is `SET_TEST`, depending on the value of the `ioctl` argument, the device enters or leaves the test (loopback) mode. The routine `rbounds` checks if the user’s pointer is valid. If the pointer is valid, the test mode of the RS-232 UART port is set depending on the argument value. Otherwise, the `EFAULT` error is returned.

Dynamic Installation

```
/* Character entry points for dynamic installation of the driver. */
struct dldd entry_points = {
   RS232_open,
   RS232_close,
   RS232_read,
   RS232_write,
   RS232_select,
   RS232_ioctl,
   RS232_install,
   RS232_uninstall,
   (char *) NULL
};
#endif
```

The above structure defines the entry points of the sample driver, which are required for dynamic installation.

CHAPTER 7 *Interrupt Handling*

Interrupts are external hardware exception conditions which are delivered to the processor to indicate the occurrence of a specific event. Some of the things for which they are useful are listed below:

- Indication of the completion of an operation.

  For example, an interrupt could be generated indicating the completion of a Direct Memory Access (DMA) transfer. The device driver would give a command to the DMA controller to transfer a block of data and set the vector for the interrupt generated by the controller to a specific driver function. This, in turn, would signal a semaphore to wake up any system or user threads waiting on the completion of the DMA transfer.

- Data availability.

  The availability of data at a port is often indicated by an interrupt. A `tty` driver receives an interrupt when a character is ready to be read from the port.

- Device ready for a command.

  A printer generates an interrupt when it has printed a character and is ready to print the next character.

LynxOS-178 Interrupt Handlers

Interrupt handlers in LynxOS-178 are specified in the `install` or `open` entry points and are cleared in the `uninstall` or `close` entry points. These interrupt handlers run before any other Kernel or application processing is completed.

Since interrupt handlers have the highest run priority, the minimization of the length of each interrupt service routine is paramount. This leads to better interrupt response time and task response time.

Interrupt handlers in LynxOS-178 are declared and reset using the functions `iointset()` and `iointclr()` or, in case the Device Resource Manager is used, `drm_register_isr()` and `drm_unregister_isr()` for Legacy Interrupts, `drm_msi_register_isr()` and `drm_msi_unregister_isr()` for Message Signaled Interrupts. See the man pages for a complete description of these functions. Interrupt handlers and other driver calls not invoked within a process' context cannot directly use application virtual addresses. The application virtual addresses must be translated to Kernel virtual addresses before they can be accessed by driver routines outside that processor context.

General Format

The general format of an interrupt-based device driver under LynxOS-178:

- The general driver entry points are referred to as the top-half of the device driver.
- The interrupt handler is known as the bottom-half of the driver.

Use of Queues

Queues are often used to communicate between the top and bottom halves. Examples of the use of queues to communicate between entry points and interrupt handlers follow:

- For communication from the `write()` entry point to the interrupt handler.

  A counting semaphore, initialized to the size of the queue, tracks the free space in the `write()` entry point. The `swait()` routine is called and if space is available in the queue, a character is enqueued. The interrupt handler subsequently dequeues the character and signals the semaphore using `ssignal()`.

- For communication from the interrupt handler to the `read()` entry point.

  The semaphore in the `read()` entry point tracks data availability in the queue. The `swait()` routine blocks until data is available in the queue. The interrupt handler posts the data to the queue, signaling the semaphore that data is available, if queue space is available. If queue space is unavailable, an error flag is set.

Typical Structure of Interrupt-based Driver

```
device_read()
{
    swait(&receive_data_available,SEM_SIGIGNORE);
    disable();
    dequeue_receive_data();
    restore();
}
device_write()
{
    disable();
    swait(&space_on_queue_available,SEM_SIGIGNORE);
    restore();
}
interrupt_handler()
{
    if (data_received) {
        enqueue_receive_data();
        ssignal(&receive_data_available);
    } else {
        if (dequeue_send_data()) {
            output_data();
        } else {
            no_interrupt_pending = 1;
        }
    }
}
```

Example

What follows is an example of the interrupt-based LynxOS driver for the 16550-compatible UART device (tty driver) that adheres to the format specified in Chapter 6, “Elementary Device Drivers”. Unlike Chapter 6, this chapter does not present all the routines the driver source code contains. Only those data structures and functions that are essential for understanding the driver internals are considered.

tty Manager

The LynxOS-178 tty driver employs a separate Kernel subsystem called the tty manager that contains various routines to handle such tasks as the character queue management, general tty I/O, opening and closing the tty device, and much more. This section contains a brief description of the tty manager routines that are essential for understanding the internals of the LynxOS-178 interrupt-based tty driver. The implementation details for these routines are omitted for the sake of brevity and clearness in this section. Refer to the contents of the file `ttymgr.c` for further details.

The tmgr_install() Function

The prototype of the `tmgr_install()` function is as follows:

```
int
tmgr_install(register struct ttystatics *s, register struct
    old_sgttyb *sg, int type, void (*transmit_enable)(),
    char *arg).
```

After clearing out the tty statics structure, the `tmgr_install()` routine initializes the values of the former structure, depending on the values of the other arguments this function is called with.

The tmgr_write() Function

The prototype of the `tmgr_write()` routine is as follows:

```
int tmgr_write(register struct ttystatics *s, struct
       file *f, char *buff, int count)
```

This routine handles various tasks pertaining to writing to the serial port. The sophisticated algorithm implemented in this function manages, among other things, the write queue and takes into account certain unusual situations, which may occur at or during the write operations.

The tmgr_read() Function

The prototype of the `tmgr_read()` function is as follows:

```
int tmgr_read(register struct ttystatics *s, struct
    file *f, char *buff, int count)
```

The code in the Kernel tty manager function implements the tty read functionality. The function is called from the read entry point of the LynxOS interrupt-based tty driver `read` entry point. The read queue management is also being carried out by this function.

The tmgr_ex() Function

The prototype of the `tmgr_ex()` function is as follows:

```
int tmgr_ex(register struct ttystatics *s, int com, int c)
```

The function is an exception dispatcher in the Kernel tty manager. This routine manages various exceptional serial line conditions, such as modem hang-up, parity error, lost carrier, and the like. The exception code is transmitted to this function via the `com` variable. The third argument of this function, an integer value `c`, is used to check which symbol was received when the error was encountered. Depending on the value of the `com`variable (the exception code), the tty manager exception dispatcher decides how to proceed with the particular serial line error.

Device Information Definition

```
/* ttyinfo.h */

struct tty_uinfo {
    long location;
    long vector;
    struct old_sgttyb sg;
};
```

The device information definition structure has one field associated with the interrupt-based driver. The `sg` structure is used by the tty device driver to define the modes and option settings for the serial port.

Device Information Declaration

```
/* ttyinfo.c */

#include "ttyinfo.h"

#define COM(name, ispeed, ospeed, local_mode_flags)      \
struct tty_uinfo name = {
    {
        ispeed, ospeed,          /* input and output speed */
         'H' - '@',             /* erase char */
         -1,                    /* 2nd erase char */
         'U' - '@',             /* kill char */
         ECHO | CRMOD,          /* mode */
         'C' - '@',             /* interrupt character */
         '\\' - '@',            /* quit char */
         'Q' - '@',             /* start char */
         'S' - '@',             /* stop char */
         'D' - '@',             /* EOF */
         -1,                    /* brk */
         local_mode_flags,      /* local mode word */
         'Z' - '@',             /* process stop */
         'Y' - '@',             /* delayed stop */
         'R' - '@',             /* reprint line */
         'O' - '@',             /* flush output */
         'W' - '@',             /* word erase */
         'V' - '@'              /* literal next char */
    }
}
```

```
/* COM1 port information structure */
COM(com1, B115200, B115200, \
    (LCRTBS | LCRTERA | LCRTKIL | LCTLECH | LNOMDM));

COM(com2, B115200, B115200, \
    (LCRTBS | LCRTERA | LCRTKIL | LCTLECH | LNOMDM));
```

The code snippet above fills the `sg` structure with the default values for COM1 and COM2 serial ports.

tty Manager Statics Structure

```
/* ttymgr.h -- the tty manager statics structure */

struct ttystatics {
    int flags;
    struct old_sgttyb sg;
    struct termios tm;
    long xflags1;
    struct queue rxqueue;
    struct queue txqueue;
    ksem_t hsem;
    char t_line;            /* indication for SLIP discipline */
};
```

This structure is a statics structure for the Kernel tty manager. It serves a purpose similar to that of the driver statics structure. In other words, it holds data that is shared between the function of the tty manager and those of the tty driver itself.

NOTE: The name of the tty driver's statics structure in the LynxOS-178 source code is the tty_u structure (refer to "Statics Structure"), which is defined in `ttydrvr.c`. This is different from the `ttystatics` structure used by the tty manager and defined in `ttymgr.h`.

In order to save space and reduce the information overload in this chapter, only some fields that are specific to the code examples given in this chapter are shown. Refer to the `struct ttystatics` definition in the header file `ttymgr.h` for complete information about the other fields in the statics structure of the LynxOS-178 tty driver.

The `rxqueue` and `txqueue` are receive and transmit queues, respectively. The `close_sem` is a semaphore that is used in the serial line closing code for handling the system timeouts. The `trans` field is a pointer to the function, which enables transmission, while `transarg` is the argument to the transmitter enabling function, with which the latter is called at various places in the Kernel tty manager.

The hsem variable is a Kernel semaphore used in the LynxOS-178 interrupt-based tty driver to implement critical sections at various places of the tty driver code in a manner explained in “Using Kernel Semaphores for Mutual Exclusion” in Chapter 4, “Synchronization”.

The xflags1 field holds the information about various facets of the UART channel state: whether the parity check is enabled, whether the serial line ignores a break, and so on. Refer to the file ttymgr.h for a complete list of values this field can assume.

The meaning of the sg field in the tty manager statics structure is the same as that of the field with the same name in the tty driver info structure. Refer to “Device Information Definition” on page 79 and “Device Information Declaration” on page 79 for more information.

The flags field is another integer whose bits denote various properties of the serial line. It is this variable that knows whether, for example, the line has transmission disabled or if it is at all active. Refer to the file ttymgr.h for a complete list of values this field can assume.

The termios field in the tty manager statics structure is the standard UNIX-like general terminal interface that is provided to control asynchronous communications ports. The fields of this structure describe the input, output, and control modes of the communication port, among other things.

Statics Structure

```
/* ttydrvr.c -- the tty driver statics structure */

#include <ttymgr.h>    /* for struct ttystatics, etc. */

struct tty_u {
    long vector;
    __port_addr_t uartp;
    int dummy_sem1;
    int dummy_sem2;
    int cts;           /* Clear To Send */
    int int_link_id;   /*
                           Interrupts can only be shared on
                           PPC sitka boards
                       */
    int init;
    int dcd_check;

    struct ttystatics channel;
};
```

The tty_u structure plays the role of the driver statics structure in the LynxOS-178 tty driver. In the tty_u structure, in addition to the ttystatics structure, some other tty channel-specific data is placed, such as the port address; a couple of semaphores used in the driver auxiliary routines; the Clear-to-Send state used in hardware handshake code; the `dcd_check` field, which is nonzero whenever the driver is doing a hardware handshake; the `int_link_id` field needed for the interrupt sharing code; and the boolean `init` field that is set to true if the hardware has been already initialized by the device driver.

The LynxOS-178 Kernel allows users to share one interrupt vector with more than one interrupt handler. The `ioint_link()` Kernel routine is used for this purpose. This routine should be called with the integer value returned from the corresponding call to the `iointset()` Kernel function. For more information, refer to the LynxOS-178 (RSC) Partitioning and RSC Interface Analysis Document (available for purchase as part of the DO-178 Artifacts Package. It is this latter return value that is stored in the `int_link_id` field of the tty driver statics structure for further use from the interrupt service routine (refer to "Interrupt Handler" on page 87).

Note that in case of extending the tty driver or when doing some modifications to the latter, it is the `tty_u` structure that is a convenient place to put the data specific to a driver instance. Refer to Chapter 8, "Kernel Threads and Priority Tracking" for an extended version of the UART channel structure.

install Entry Point

```
struct tty_u *
ttyinstall(struct tty_uinfo *info)
{
    struct tty_u        *p;
    struct ttystatics   *s;
    struct old_sgttyb   *sg;
    unsigned long       bus_freq;
    __port_addr_t       uartp;

    dev_info_t *uart = bsp_get_dev_by_type(uarts_found ++, UART_DEV_TYPE);
    if (uart == NULL)
        return (void *)SYSERR;
    /* There is only offset in info */
    uartp = uart->vaddr;

    dev_info_t *platform = bsp_get_dev_by_type(0, PLATFORM_DEV_TYPE);
    if (platform == NULL)
        return (void *)SYSERR;

    ...

    /* Get Core Complex Bus (CCB) clock */
    bus_freq = platform->u.p.bus_freq;

    /* Set baudrates */
    ba[B9600]   = UART_DIVISOR(bus_freq, 9600);
    ba[B19200]  = UART_DIVISOR(bus_freq, 19200);
```

```
    ba[B38400] = UART_DIVISOR(bus_freq, 38400);
    ba[B57600] = UART_DIVISOR(bus_freq, 57600);
    ba[B115200] = UART_DIVISOR(bus_freq, 115200);
    ba[B230400] = UART_DIVISOR(bus_freq, 230400);
    ba[EXTA]    = UART_DIVISOR(bus_freq, 115200);
    ba[EXTB]    = UART_DIVISOR(bus_freq, 230400);

    p = (struct tty_u *)sysbrk((long)sizeof(struct tty_u));

    if (p == NULL) {
        debug(("ttyinstall: sysbrk() failed\n"));
        return (char *)SYSERR;
    }

    bzero((void *)p, sizeof(struct tty_u));

    p->cts    = 0;
    p->uartp  = uart->vaddr;
    p->vector = uart->irq;

    sg = (struct old_sgttyb *)&info->sg;
    s  = &p->channel;

    /*
        The trans_en() routine looks at uartp and channel.tm.c_cflag,
        so it needs the full tty_u structure passed to it.
    */
    tmgr_install(s, sg, 0, trans_en, (char *)p);
    s->hsem = 1;

    ttyseth(s, (struct sgttyb *)sg, uartp, 1, 0);

    /* The device is not open yet, so disconnect the modem control lines */
    __port_andb(uartp + U_MCR, ~(MCR_DTR | MCR_RTS | MCR_OT2));

    debug(("ttyinstall: call iointset(%d) with p 0x%08x\n", p->vector,
p));
    p->int_link_id = iointset(p->vector, (int (*)())duart_int, (char *)p);
    debug(("ttyinstall: iointset() returns %d\n", p->int_link_id));

    if (p->int_link_id == SYSERR) {
        sysfree((void *)p, sizeof(struct tty_u));
        return (char *)SYSERR;
    }

    p->init = 0;

#if defined(__TTY_DEVCON__)
    /* Register entry points with devcon driver in case it is console */
    debug(("ttyinstall: call devcon_register()\n"));
    devcon_register((void *)p, &tty_funcs, tty_skdb_hook);
#endif

        /* Enable Modem Status Interrupt */
    debug(("ttyinstall: enable modem status interrupt\n"));
    __port_orb(uartp + U_IER, IER_MS);

    ...

    return p;
}
```

The `ttyinstall()` routine searches for a UART device, using the `bsp_get_dev_by_type()` function with UART_DEV_TYPE as the second argument.

Once the presence of the UART is confirmed, the `ttyinstall()` routine gets the platform bus frequency using the `bsp_get_dev_by_type()` function with PLATFORM_DEV_TYPE as the second argument and sets the divisor latch register. Then the `tty_u` data structure is allocated and initialized by the board-specific parameters supplied by means of the device information structure. The `tmgr_install()` routine from the tty manager Kernel subsystem is then invoked to initialize the `ttystatics` structure as well as some Kernel structures pertaining to the tty subsystem.

The `hsem` semaphore is initialized to the value of 1. Note that the semaphore initialization must be made before calling the `ttyseth()` function, which is described later in this chapter, in order to make certain that the call to `swait()` in `ttyseth()` succeeds.

The `iointset()` function is called to set the interrupt handler `duart_int()`. The `duart_int()` is defined by the tty driver and described in “Interrupt Handler” on page 95.

The return value of the `iointset()` function is saved into the `int_link_id` field of the channel structure `p` and used later by the interrupt sharing mechanism.

The modem control lines are disconnected and the modem status interrupts are enabled in the `ttyinstall()` routine. This puts the serial line into the state in which it is expected to stay after opening the serial line.

uninstall Entry Point

```
int ttyuninstall(struct tty_u *p) {
    __port_addr_t uartp;

    uartp = p->uartp;

    __outb(uartp + U_IER, 0);
    iointclr(p->vector);
    sysfree(p, (long)sizeof(struct tty_u));

    return OK;
}
```

The `uninstall` routine writes zero to the serial line interrupt enable register `U_IER`, thus disabling hardware interrupts from the serial line, resets the interrupt handler for the serial line interrupt vector and frees the memory allocated for the `tty_u` structure at driver installation.

open Entry Point

```
int ttyopen(struct tty_u *p, int dev, struct file *f)
{
    int out;
    struct ttystatics *s;
    __port_addr_t uartp;
    int data;
    int i;

    s = &p->channel;
    if ((out = tmgr_open(s, f)) == SYSERR) {
        return SYSERR;
    }

    if (s->sg.sg_ospeed != B0) {
        /*
            The output baud rate is not B0,
            so it's OK to assert modem control lines.
         */
        if (hardware_handshake(p, f, s) == SYSERR) {
            if (out) {
                ttyclose(p, f);
            }
            return SYSERR;
        }
    }

    if (out && 0 == p->init) {    /* first open */
        p->init = 1;
        uartp = p->uartp;
        __port_orb(uartp + U_MCR, MCR_OT2 | MCR_RTS);
        __outb(uartp + U_LCR, LCR_WL8 | LCR_STB_1);
        data = __inb(uartp + U_IIR);
        data = __inb(uartp + U_MSR);

        /* flush hardware buffer before enabling interrupts */

        for (i = 0; i < MAX_COUNT; i++) {
            if (!((data =_inb(uartp + U_LSR)) & IER_RDA)) {
                break;    /* no more fake data */
            }
            data = __inb(uartp + U_RBR);
        }

        __outb(uartp + U_IER, IER_RDA | IER_TRE | IER_RLS | IER_MS);
    }
        return OK;
}
```

The Kernel tty manager routine `tmgr_open()` is used to logically open the serial line. Refer to “tty Manager” on page 77 for the description of the `tmgr_open()` function. Further, in case of no errors have been caught in the call to the tty manager `tmgr_open()` function and provided that the serial line baud rate is nonzero, a hardware handshake is attempted. If the handshake produces an error, the tty is closed and an error value is returned from the tty driver open entry point.

As has been already mentioned in “Statics Structure” on page 81, the `init` field is used to track the information about whether the device has been initialized. By this point, no errors have been caught from either the tty manager or the hardware handshake process. The positive return value obtained from the `tmgr_open()` function means success. If, however, it is simultaneously found that the UART port has not yet been properly initialized, the open entry point performs a series of initialization steps. First, using the modem control register, the Request to Send as well as the auxiliary user-designated output (OUT2) signals are forced to be set to logic 0. The line control register is then logically or'ed with the constants `LCR_WL8` and `LCR_STB_1` that set the word length to 8 and the number of stop bits to 1. Then the interrupt identification register as well as the modem state register get flushed by reading from them. A similar operation is done with the receiver buffer register: while the line state register indicates that receiver data is available, the characters are read from the receiver buffer, the latter getting thereby emptied from any random data that might be there at device opening. Finally, the `open` entry point enables all hardware interrupts by setting all four bits of the interrupt enable register to logic 1.

close Entry Point

```
void
ttyclose(struct tty_u *p, struct file *f)
{
        __port_addr_t uartp;


        tmgr_close(&p->channel, f);
        uartp = p->uartp;
        xmit_quies(uartp,p);
        /*
          According to the POSIX Test Suite (DC:settable:054),
          if HUPCL and CLOCAL are both clear, then the last close
          shall not cause the modem control lines to be disconnected.
        */
        if ((p->channel.xflags1 &
              (X1_HUP_CLOSE | X1_MODEM_OK)) != X1_MODEM_OK) {
        /* HUPCL is set or CLOCAL is set */
               __port_andb(uartp + U_MCR, ~(MCR_DTR | MCR_RTS | MCR_OT2));
        }
        __outb(uartp + U_IER, 0);
       /* disable serial interrupts */
        p->channel.flags &= ~ACTIVE;
        p->init = 0;
}
```

Like most entry points in the LynxOS-178 tty driver, the `ttyclose()` function calls the tty closing procedure `tmgr_close()` from the Kernel tty manager. Refer to "tty Manager" on page 77 for a brief description of the `tmgr_close()` function. The `xmit_quies()` routine, which is called immediately afterwards, basically waits for the serial line transmitter to finish its operations and become quiet. Finally, the `ttyclose()` function cleans up the serial line state and disables the serial interrupts.

Interrupt Handler

```
static void duart_int(
            struct tty_u *p) /* ptr to duart channel structure */
             int32_t plevel,
             int32_t hv
             )
{
    __port_addr_t uartp;
    struct ttystatics *s;     /* ptr to static struct for a chan */
    int status;
    int data;
    unsigned char c;          /* make unsigned to aid debugging */
    int got_break = 0;        /* added for inbreak check */
    int parity_error = 0;     /* added for parity check */

    uartp = p->uartp;
    s = &p->channel;

    for (;;) {
        if ((status = __inb(uartp + U_IIR)) & IIR_PEN) {
                                            /* no intr pending */
            break;
        }
        switch (status & IIR_IID) {
            case IIR_RLS:                   /* receiver error */
                if ((data = __inb(uartp + U_LSR)) & LSR_BI) {
                    /* Break interrupt */
                    got_break = 1;
                    tmgr_ex(s, EX_BREAK, 0);
                }
                if (data & (LSR_PE | LSR_FE)) {
                    /* Parity error or framing error */
                    if (s->xflags1 & X1_ENABLE_INPUT_PC) {
                        parity_error = 1;
                    }
                }
                if ((data & LSR_BI) || !(data & (LSR_PE | LSR_FE))) {
                    /* Flush the receive buffer register */
                    data = __inb(uartp + U_RBR);
                }
                break;
            case IIR_RDA:       /* receiver data ready */
                data = __inb(uartp + U_RBR);

                if (skdb_entry_from_irq &&__tty_is_skdb_port(uartp)
                    && data == skdb_key_serial) {
                    tty_skdb(p);
                    continue;
                }

                /*
                    The ttydrvr receives a 0 on transition
                    from break on to break off.
                 */
                if (!data && (s->t_line != SLIPDISC)) {
```

```
                    if (got_break == 1) {
                        got_break = 0;
                        break;    /* don't send character on */
                    }
                }

                if (!parity_error) {
                    if (tmgr_rx(s, data)) {
                        /* enable transmit interrupt */
                        __port_orb(uartp + U_IER, IER_TRE);
                    }
                }
                else { /* parity error */
                    tmgr_ex(s, EX_PARITY, data);
                    parity_error = 0;
                    /* enable transmit interrupt */
                    __port_orb(uartp + U_IER, IER_TRE);
                }
                break;
            case IIR_TRE: /* transmitter empty */
                if (p->dcd_check && !p->cts) {
                    /* do not send because "No CTS" */
                    break;
                }
                if (tmgr_tx(s, (char *)&c)) { /* call manager */
                    /* disable transmit interrupt */
                    __port_andb(uartp + U_IER, ~IER_TRE);
                }
                else {
                    __outb(uartp + U_THR, c);
                }
                break;
            case IIR_MS:
                /* flow control hardware handshake */
                data = __inb(uartp + U_MSR);

                if ((data & MSR_DCTS) && (p->dcd_check
                    || (s->tm.c_cflag & (V_CRTSCTS | V_CRTSXOFF)))) {
                    if (!(data & MSR_CTS)) { /* DCE is not ready */
                        p->cts = 0;
                    }
                    else {
                        p->cts = 1;
                        /* enable transmit interrupt */
                        __port_orb(uartp + U_IER, IER_TRE);
                    }
                }
                break;
            }
        }

        /* shared interrupts */
        ioint_link(p->int_link_id),
        plevel, hv);
}
```

The above example is a simplified version of the `duart_int()` interrupt handler routine for the LynxOS-178 tty driver. The status word is obtained from the UART port, and, if no interrupt is pending, no action is taken by the interrupt handler. In case of a receiver error, the serial port line status register is queried for the data contained therein. If the data received indicates a break interrupt, the exception handling routine `tmgr_ex()` from the Kernel tty manager is invoked and the `got_break` variable, which is used elsewhere in the interrupt handler code, is set to 1. Refer to "tty Manager" on page 77 for the description of the `tmgr_ex()` function. If a break interrupt is encountered, the receive buffer register is forcibly flushed by the tty driver.

In the case of a receiver data ready condition with no parity error detected, the transmitter interrupts are enabled by writing the appropriate constant into the UART interrupt enable register after the `tmgr_rx()` function returns the positive truth value indicating that the queue management has succeeded. The parity error, in its turn, triggers the call to the `tmgr_ex()` function, which is an exception dispatcher implemented in the Kernel tty manager (refer to "tty Manager" on page 77). The transmitter empty interrupt condition leads to a call to the `tmgr_tx()` and, depending on the return value of the latter, either the transmit interrupts are masked out or the character to be sent is put to the UART transmit buffer.

If the modem status interrupt is received, the tty driver interrupt handler resets the interrupt by reading the UART modem status register. After that, the driver provides for the special case when the following conditions are met:

- [The CTS input to the chip has changed state since the last time it was read by the CPU]

  AND

- [[The interrupt occurred during hardware handshake, which is indicated by nonzero value of the `dcd_check` field of the tty driver statics structure]

  OR

- [The `termios` control flag indicates that the hardware flow control is taking place]]

In this case, if the modem status register indicates that the clear-to-send is not set in the hardware, the `cts` field in the tty driver statics structure is zeroed out. Otherwise, the latter field is set to logic 1 and the transmit interrupts are enabled by writing an appropriate value into the UART interrupt enable register.

Having done everything needed to handle the interrupt received, the `duart_int()` function calls the `ioint_link()` function with the value that was stored in the tty driver statics structure at driver installation.

The function in the Kernel tty manager offer the mechanisms to handle the character buffer and queue management for the serial device. That is why the interrupt service routine itself has no buffer or queue handling code whatsoever.

write Entry Point

```
int ttywrite(struct tty_u *p, struct file *f, char *buff, int count)
{
      return tmgr_write(&p->channel, f, buff, count);
}
```

In this example, the *write to the serial port* functionality is entirely handled by the Kernel tty manager. Refer to “tty Manager” on page 77 for a brief description of the `tmgr_write()` function.

read Entry Point

```
int ttyread(struct tty_u *p, struct file *f, char *buff, int count)
{
      return tmgr_read(&p->channel, f, buff, count);
}
```

As in the write entry point of the LynxOS-178 tty driver, the `ttyread` function calls the `tmgr_read()` routine from the Kernel tty manager. It is the latter function that implements the read functionality for the tty driver. Refer to “tty Manager” on page 77 for a description of the `tmgr_read()` function.

ioctl Entry Point

The implementation of the `ioctl`entry point in the LynxOS-178 interrupt-based tty driver is not in any way specific to the class of the interrupt-driven device drivers. It is not, therefore, considered educational to analyze the source code of this entry point in this chapter.

The ttyseth Procedure

```
struct sgttyb {
    char sg_ispeed;          /* input speed */
    char sg_ospeed;          /* output speed */
    char sg_erase;           /* erase char (default "#") */
    char sg_2erase;          /* 2nd erase character */
    char sg_kill;            /* kill char (default "@") */
    int sg_flags;            /* various modes and delays */
};
```

```
static void
ttyseth(struct ttystatics *s, struct sgttyb *sg,
        __port_addr_t uartp, int firsttime, int tflags)
{
    swait(&s->hsem, SEM_SIGIGNORE);
    if (firsttime || (sg->sg_flags & (EVENP|ODDP)) != (tflags &
(EVENP|ODDP))) {
        if (sg->sg_flags & EVENP) {
          if (sg->sg_flags & ODDP) {  /* both even and odd 7 bit no parity
*/
                __outb(uartp + U_LCR, LCR_WL7 | LCR_STB_1);
            }
            else {
                __outb(uartp + U_LCR, LCR_WL7 | LCR_PEN | LCR_EPS |
LCR_STB_1);


            }
        }
        else if (sg->sg_flags & ODDP) {
            __outb(uartp + U_LCR, LCR_WL7 | LCR_PEN | LCR_STB_1);
        }
        else {
            __outb(uartp + U_LCR, LCR_WL8 | LCR_STB_1);
        }
    }

    if (s->sg.sg_ispeed != sg->sg_ispeed || s->sg.sg_ospeed != sg-
>sg_ospeed) {
        if (!set_baud(uartp, sg->sg_ispeed, sg->sg_ospeed)) {
            if (s->sg.sg_ospeed == B0 && sg->sg_ospeed != B0) {
                /*
                    Changing output speed from B0 to non-B0, so it is
                    necessary to reconnect the modem control lines.
                */
                __port_orb(uartp + U_MCR, (MCR_DTR | MCR_RTS | MCR_OT2));
            }
            s->sg.sg_ispeed = sg->sg_ispeed;
            s->sg.sg_ospeed = sg->sg_ospeed;
        }
    }
    ssignal(&s->hsem);
}
```

`ttyseth()` is a helper routine called from both the install and `ioctl` entry points of the LynxOS-178 VMPC tty driver.

The `ttyseth()` function receives the pointer to the statics structure as its first argument. Various bits and pieces of information about the serial line parameters are stored in the fields of the `sgttyb` structure, which is given as the second argument to this function. The UART port address, the first time call indicator flag, and the `tflags` bitmask make up the rest of the arguments to this helper routine.

Inside a critical section isolated by the `swait()`/`ssignal()` pair (refer to “Using Kernel Semaphores for Mutual Exclusion” on page 69 in Chapter 4), modes and option settings for the serial port are set by the `ttyseth()` procedure. For example, if the function is called for the first time (which happens when the `install` entry point calls it), the serial line word length, the number of stop bits, and the parity enable status are set according to the values received by this function from its caller.

CHAPTER 8 Kernel Threads and Priority Tracking

To off-load processing from interrupt-based sections of a device driver,

LynxOS-178 offers a feature known as *Kernel Threads,* also referred to as system threads. Kernel Threads are defined as independently schedulable entities which reside in the Kernel's virtual address space. They closely resemble processes but do not have the memory overhead associated with processes.

Although Kernel threads have independent stack and register areas, the Kernel threads share both text and data segments with the Kernel. Each Kernel thread has a priority associated with it which is used by the operating system to schedule it. Kernel threads can be used to improve the interrupt and task response times considerably. Thus, they are often used in device drivers.

Priority tracking is the method used to dynamically determine the Kernel thread's priority. The Kernel thread assumes the same priority as the highest-priority application which it is currently servicing.

Device Drivers in LynxOS-178

Device drivers form an important part of any operating system, but even more so in a real-time operating system such as LynxOS-178. The impact of the device driver performance on overall system performance is considerable. Since it is imperative for the operating system to provide deterministic response time to real-world events, device drivers must be designed with determinism in mind.

Some of the important components of real-time response are described in the following sections.

Interrupt Latency

Interrupt Latency is the time taken for the system to acknowledge a hardware interrupt. This time is measured from when the hardware raises an interrupt to when the system starts executing the first instruction of the interrupt routine (in the case of LynxOS-178 this routine is the interrupt dispatcher). This time is dependent on the interrupt hardware design of the system and the longest time interrupts are disabled in the Kernel or device drivers.

Interrupt Dispatch Time

Interrupt dispatch time is the time taken for the system to recognize the interrupt and begin executing the first instruction of the interrupt handler. Included in this time is the latency of the LynxOS-178 interrupt dispatcher (usually negligible).

Driver Response Time

The D*river Response Time* is the sum of the interrupt latency and the interrupt dispatch time. This is also known as the interrupt response time.

Task Response Time

The *Task Response Time* is the time taken by the operating system to begin running the first instruction of an application task after an interrupt has been received that makes the application ready to run. This figure is the total of:

- The driver response time (including the delays imposed by additional interrupts)
- The longest preemption time
- The context switch time
- The scheduling time
- The system call return time

Only the driver response time and the preemption time are under the control of the device driver writer. The other times depend on the implementation of LynxOS-178 on the platform for which the driver is being written.

Task Completion Time

The *Task Completion Time* is the time taken for a task to complete execution, including the time to process all interrupts which may occur during the execution of the application task.

NOTE: The device driver writer should be aware of all delays that the interrupts could potentially cause to an application. This is important when considering the overall responsiveness of the "application-plus-kernel" combination in the worst-possible timing scenario.

Real-Time Response

To improve the real-time response of any operating system, the most important parameters are the driver response time, task response time, and the task completion time. The time taken by the driver in the system can have a direct effect on the system's overall real-time response. A single breach of this convention can cause a high performance real-time system to miss a real-time deadline.

Kernel threads were introduced into LynxOS-178 in order to keep drivers from interfering with the real-time response of the overall system. LynxOS-178 Kernel threads were designed specifically to increase driver functionality while decreasing driver response time, task response time, and task completion time.

In a normal system, interrupts have a higher priority than any task. A task, regardless of its priority, is interrupted if an interrupt is pending (unless the interrupts have been disabled). The result could mean that a low priority interrupt could interrupt a task which is executing with real-time constraints.

A classic example of this would be a task collecting data for a real-time data acquisition system and being interrupted by a low priority printer interrupt. The task would not continue execution until the interrupt service routine had finished.

With Kernel threads, delays of this sort are significantly reduced. Instead of the interrupt service routine doing all the servicing of the interrupt, a Kernel thread is used to perform the function previously performed by the interrupt routine. The interrupt service routine is now reduced to merely signaling a semaphore which the Kernel thread is waiting on.

Since the Kernel thread is running at the application's priority (actually it is running at half a priority level higher as described below), it is scheduled according to process priority and not hardware priority. This ensures that the interrupt service time is kept to a minimum and the task response time is kept short. A further result of this is that the task completion time is also reduced.

The use of Kernel threads and priority tracking in LynxOS-178 drivers are the cornerstone to guaranteeing deterministic real-time performance.

Kernel Threads

Kernel Threads execute in the virtual memory context of the null process, which is process 0 of the partition they are created for. However, Kernel threads do not have any user code associated with them, so context switch times for Kernel threads are quicker than for user threads. Like all other tasks in the system, Kernel threads have a scheduling priority which the driver can change dynamically to implement priority tracking. They are scheduled with the `SCHED_FIFO`algorithm.

Creating Kernel Threads

A Kernel thread is created in the install or open entry point. The advantage of starting it in the open is that, if the device is not opened, the driver doesn't use up Kernel resources unnecessarily. A kernel thread with stack size greater than `KSTACKSIZE` bytes must be only created once. In this case, as the thread is only created once, the open routine must check whether this is the first call to open. One thread is created for each interrupting device, which normally corresponds to a major device. The following code fragment illustrates how a thread might be started from the install entry point:

```
int threadfunc ();
int stacksize, priority;
char *threadname;

s->st_id = ststart (threadfunc, stacksize, priority, threadname, 1, s);
if (s->st_id == SYSERR)
{
    sysfree (s, sizeof (struct statics));
    pseterr (EAGAIN);
    return (SYSERR);
}
```

The third parameter, `VMID vm_id`, specifies the VM the thread is going to run in. It is important to mention that in LynxOS-178 partitions are created after the invocation of the drivers’ install entry points, and at the time the install entry point is executed, the total number of partitions that is going to be created is yet undetermined

The thread function specifies a C function which will be executed by the thread. The structure of the thread code is discussed in the next section.

The second argument specifies the thread's stack size. This stack does not grow dynamically, so enough space must be allocated to hold all the thread's local variables.

As Kernel threads are preemptive tasks, they have a scheduling priority, just like other user threads in the system, which determines the order of execution between tasks. The priorities of Kernel threads are discussed more fully in "Priority Tracking" on page 101. It is usual to create the thread with a priority of 1.

The thread name is an arbitrary character string which is printed in the *name* column by the `ps T` command. It will be truncated to `PNMLEN` characters (including NULL terminator). `PNMLEN` is currently 32 (see the `proc.h` file).

The last two parameters allow arguments to be passed to the thread. In most cases, it is sufficient to pass the address of the statics structure, which normally contains all other information the thread might need for communication and synchronization with the rest of the driver.

According to the Device Driver Interface Standard [4] document, each Kernel thread is only supposed to perform actions on the resources belonging to the VM it is running in. No access to the resources belonging to other VMs is allowed.

Structure of a Kernel Thread

The structure of a Kernel Thread and the way in which it communicates with the rest of the driver depends largely on the way in which a particular device is used. For the purposes of illustration, two different driver designs will be discussed.

1. *Exclusive Access*. Only one user task is allowed to use the device at a time. The exclusive access is often enforced in the open entry point.
2. *Multiple Access*. Multiple user tasks are permitted to have the device open and make requests.

Exclusive Access

If we consider synchronous transfers only, then this type of driver will typically have the following structure:

1. The top-half entry point (read/write) starts the data transfer on the device, then blocks waiting for I/O completion.
2. The interrupt handler signals the Kernel thread when the I/O completes.
3. The Kernel thread consists of an infinite `for` loop which does the following:
   - Wait for work to do
   - Process interrupt
   - Wake up user task

The statics structure will contain a number of variables for communication between the thread and the other entry points. These would include synchronization semaphores, error status, transfer length, and so on.

Top-Half Entry Point

The read/write entry point code will not be any different from a driver that does not use Kernel threads. It starts an operation on the device, then blocks on an event synchronization semaphore.

```
int drv_read (s, f, buff, count)
struct statics *s;
struct file *f;
char  *buff;
int count;
{
    start_IO (s, buff, count, READ);
                         /* start I/O on device */
    swait (&s->io_sem, SEM_SIGABORT);
                         /* wait for I/O completion */
    if (s->error) {      /* check error status */
        pseterr (EIO);
        return (SYSERR);
    }
    return (s->count);   /* return # bytes transferred */
}
```

Interrupt Handler

Apart from any operations that may be necessary to acknowledge the hardware interrupt, the interrupt handler's only responsibility is to signal the Kernel thread, informing it that there is some work to do:

```
intr_handler (s) struct statics *s;
{
    ssignal (&s->intr_sem);  /* wake up kernel thread */
}
```

Kernel Thread

The Kernel thread waits on an event synchronization semaphore. When an interrupt occurs, the thread is woken up by the interrupt handler. It processes the interrupt, checking the device status for errors and the like and wakes up the user task that is waiting for I/O completion. For best system real-time performance, the Kernel thread should reenable interrupts from the device.

```
kthread (s)
struct statics *s;
{
     for (;;) {
      swait (&s->intr_sem, SEM_SIGIGNORE);
                        /* wait for work to do*/
...
      /* process interrupt, check for errors etc. */
...
      if (error_found) s->error = 1;
                        /* tell user task there was an error */
      ssignal (&s->io_sem);  /* wake up user task */
     }
}
```

Multiple Access

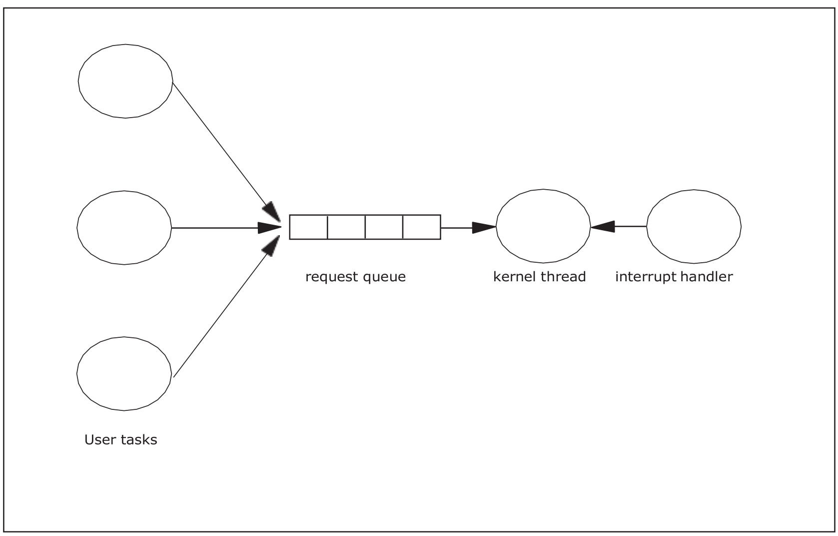
In this type of design, any number of user tasks can open a device and make requests to the driver. But as most devices can perform only one operation at a time, requests from multiple tasks must be held in a queue. In a system *without* Kernel threads, the structure of such a driver is:

1. The top-half routine starts the operation immediately if the device is idle; otherwise, it enqueues the request. It then blocks, waiting for the request to be completed.
2. The interrupt handler processes interrupts, does all I/O completion, wakes up the user task and then starts the next operation on the device immediately if there are queued requests.

The problem with this strategy is that it can lead to an overly long interrupt routine owing to the large amount of work done in the handler. Since interrupt handlers are not pre-emptive, this can have an adverse effect on system response times. When multiple requests are queued up, the next operation is started immediately after the previous one has finished. The result of this is that a heavily used device can generate a series of interrupts in rapid succession until the request queue is emptied. Even if the requests were made by low priority tasks, the processing of these interrupts and requests will take priority over high priority tasks because it is done within the interrupt handler.

The use of Kernel threads resolves these problems by off-loading the interrupt handler. A Kernel thread is responsible for dequeuing and starting requests, handling I/O completion and waking up the user tasks. Figure 8-1 illustrates the overall design.

A data structure containing variables for such things as event synchronization and error status is used to describe each request. The pending request queue and list of free request headers will be part of the statics structure. The interrupt handler code is the same as in the case of the exclusive use design



Top-Half Entry Point

```
drv_read (s, f, buff, count)
struct statics *s;
struct file *f;
char  *buff;
int count;
{
    struct req_hdr *req;

    ...
    enqueue (s, req);        /* enqueue request */
    swait (&req->io_sem, SEM_SIGABORT);
                                /* wait for I/O completion */
    ...
}
```

Kernel Thread

```
kthread (s)
struct statics *s
{
    struct req_hdr *curr_req;

    for (;;) {
        curr_req = dequeue (s); /* wait for a request */
        start_IO (s, curr_req); /* start I/O operation */
        /* wait for I/O completion */
        swait (&s->intr_sem, SEM_SIGIGNORE);
        ...
        /* process interrupt, check for errors etc. */
        ...
        if (error_found)
            /* tell user task there was an error */
            curr_req->error = 1;
        /* wake up user task */
        ssignal (&curr_req->io_sem);
    }
}
```

Priority Tracking

The previous examples did not discuss the priority of the Kernel thread. It was assumed to be set statically when the thread is created. There is a fundamental problem with using a static thread priority in that, whatever priority is chosen, there are always some conceivable situations where the order of task execution does not meet real-time requirements. The same is true of systems that implement separate scheduling classes for system and user level tasks.

Figure 8-2 shows two possible scenarios in a system using a static thread priority. In both scenarios, Task A is using a device that generates work for the Kernel thread. Other tasks, with different priorities, exist in the system. These are represented by Task B.

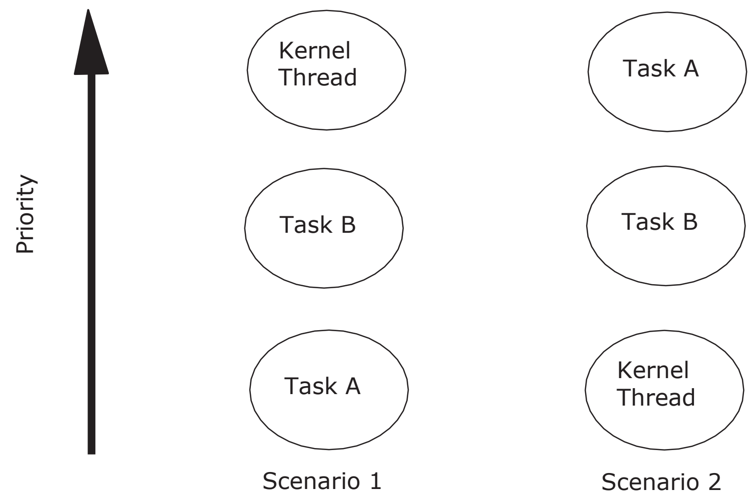


Figure 8-2: Scheduling with Static Thread Priorities

In the first scenario, Task B has a priority higher than Task A but lower than the Kernel thread. The Kernel thread will be scheduled before Task B even though it is processing requests on behalf of a lower priority task. This is essentially the same situation that occurs when interrupt processing is done in the interrupt handler. In Scenario 2, the situation is reversed. The Kernel thread is preempted by Task B resulting in Task A being delayed.

The only solution that can meet the requirements of a deterministic real-time system with bounded response times is to use a technique that allows the Kernel thread priority to dynamically follow the priorities of the tasks that are using a device.

User and Kernel Priorities

User applications can use 256 priority levels from 0–255. However, internally the Kernel uses 512 priority levels, 0–511. The user priority is converted to the internal representation simply by multiplying it by two, as illustrated in Figure 8-3.

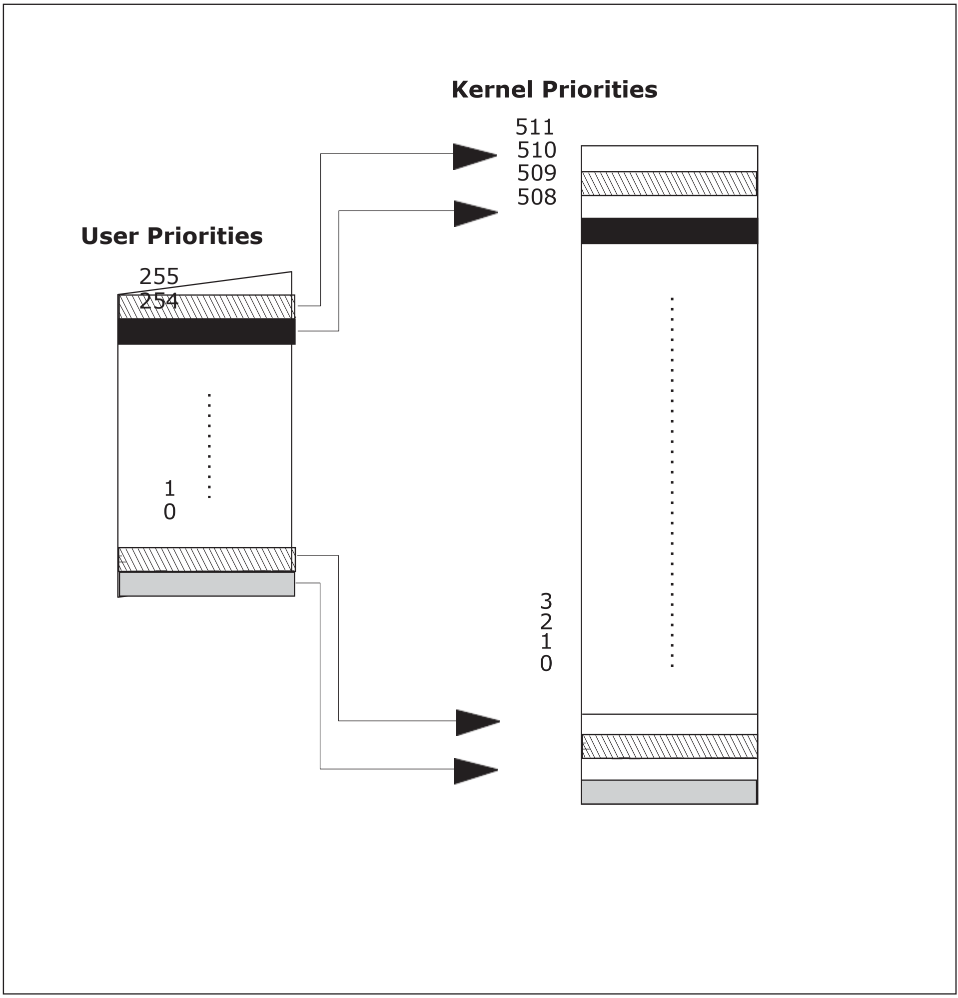


Figure 8-3: User and Kernel Priorities

As can be seen, a user task will always have an even priority at the Kernel level. This results in “empty”, odd priority slots between the user priorities. These slots play an important role in priority tracking.

The examples below will again discuss the exclusive and multiple access driver designs for illustrating priority tracking techniques.

Exclusive Access

Whenever a request is made to the driver, the top-half entry point must set the Kernel thread priority to the priority of the user task.

```
drv_read (s, f, buff, count)
struct statics *s;
struct file *f;
char  *buff;
int count;
{
    uprio = _getpriority ();
    /* get priority of current task */
    stsetprio (s->kt_id, (uprio << 1) + 1);
    /* set k.t. priority */
    start_IO (s, buff, count, READ);
    /* start I/O on device */
    swait (&s->io_sem, SEM_SIGABORT);
    /* wait for I/O completion */
    if (s->error) {
     /* check error status */
        pseterr (EIO);
        return (SYSERR);
    }
    return (s->count);
    /* return # bytes transferred */
}
```

The expression `(uprio << 1) + 1` converts the user priority to a Kernel level priority. The thread priority is in fact set to the odd numbered Kernel priority just above the priority of the user task. This ensures that the Kernel thread executes before any tasks at the same or lower priority as the user task making the request but after any user tasks of higher priority, as shown in Figure 8-4.

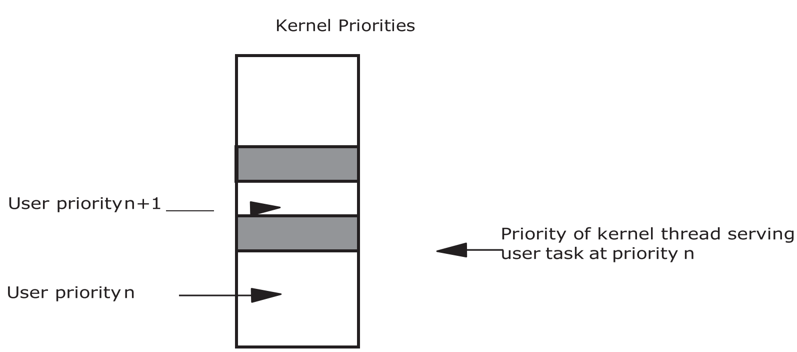


Figure 8-4: Kernel Thread Priorities

When the request has been completed the thread resets its priority to its initial value.

```
kthread (s)
struct statics *s;
{
    for (;;) {
        swait (&s->intr_sem, SEM_SIGIGNORE);
         /* wait for work to do */
         ...
        /* process interrupt, check for errors etc. */
         ...
        if (error_found)
            s->error = 1;
             /*tell user task there was an error*/
        ssignal (&s->io_sem);
         /* wake up user task */
        stsetprio (s->kt_id, 1);
        /* reset kernel thread priority */
    }
}
```

Multiple Access

As was seen in the previous discussion of this type of design, the driver maintains a queue of pending requests from a number of user tasks. These tasks will probably have different priorities. So, the driver must ensure that the Kernel thread is always running at the priority of the highest priority user task which has a request pending. If the requests are queued in priority order this will ensure that the thread is always processing the highest priority request. The thread priority must be checked and adjusted at two places: whenever a new request is made and whenever a request is completed.

How can the driver keep track of the priorities of all the user tasks that have outstanding requests? In order to do this the driver must use a special data structure, `struct priotrack`, defined in `st.h`. Basically, the structure is a set of counters, one counter for each priority level. The value of each counter represents the number of outstanding requests at that priority. The values of the counters are incremented and decremented using the routines `priot_add` and `priot_remove`. The routine `priot_max` returns the highest priority in the set. The use of these routines is illustrated in the following code examples.

Top-Half Entry Point

The top-half entry point must first use `priot_add` to add the new request to the set of tracked requests. The code then decides whether the Kernel thread's priority must be adjusted. This will be necessary if the priority of the task making the new request is higher than the thread's current priority. A variable in the statics structure is used to track the Kernel thread's current priority. The request header must also contain a field specifying the priority of the task making each request. This is used by the Kernel thread.

```
drv_read (s, f, buff, count)
struct statics *s;
struct file *f;
char  *buff;
int count;
{
    ...
    uprio = _getpriority (); /* get user task priority */
    req->prio = uprio;        /* save for later use */
    enqueue (s, req);         /* enqueue request */
    /*
     * Do priority tracking. Add priority of new request
     * to set. If priority of new request is higher than
     * current thread priority, adjust thread priority.
     */
    swait(&s->prio_sem, SEM_SIGIGNORE);
    /* synchronize with kernel thread */
    priot_add (&s->priotrack, uprio, 1);
    if (uprio > s->kt_prio) {
        stsetprio (s->kt_id, (uprio << 1) + 1);
        s->kt_prio = uprio;
    }
    ssignal(&s->prio_sem);
    swait (&req->io_sem, SEM_SIGABORT);
    /* wait for I/O completion */
    ...
}
```

Kernel Thread

When the Kernel thread has finished processing a request, the priority of the completed request is removed from the set using `priot_remove`. The thread must then determine whether to change its priority or not, depending on the priorities of the remaining pending requests. The thread uses `priot_max` to determine the highest priority pending request.

```
kthread (s)
struct statics *s;
{
    ...
    for (;;) {
        ...
        curr_req = dequeue (s); /* wait for a request */
        start_IO (s, curr_req); /* start I/O operation */
        swait (&s->intr_sem, SEM_SIGIGNORE);
        /* wait for I/O completion */
            ...
        /* process interrupt, check for errors etc. */
            ...
        /*
         * Do priority tracking. Remove priority of
         * completed request from set. Determine high
         * priority of remaining requests. If this is
```

```
                 * lower than current priority, adjust thread
                 * priority.
                 */
                swait(&s->prio_sem, SEM_SIGIGNORE);
                /* synchronize with top-half */
                priot_remove (&s->priotrack, curr_req->prio);
                maxprio = priot_max (&s->priotrack);
                if (maxprio < s->kt_prio) {
                    stsetprio (s->kt_id, (maxprio << 1) + 1);
                    s->kt_prio = maxprio;
                }
                ssignal(&s->prio_sem);
                ...
            }
        }
```

Nonatomic Requests

The previous examples implicitly assumed that requests made to the driver are handled atomically, that is to say, the device can handle an arbitrary sized data transfer. This is not always the case. Many devices have a limit on the size of transfer that can be made, in which case the driver may have to divide the user data into smaller blocks. A good example is a driver for a serial device. A user task may request a transfer of many bytes, but the device can transfer only one byte at a time. The driver must split the request into multiple single byte requests.

From the point of view of priority tracking, a single task requesting an *n* byte transfer is equivalent to *n* tasks requesting single byte transfers. Since each byte is handled as a separate transfer by the driver (each byte generates an interrupt), the priority tracking counters must count the number of bytes rather than the number of requests.

The functions `priot_add`*n* and `priot_remove`*n* can be used to add and remove multiple requests to the set of tracked priorities. What is defined as a request depends on the way the driver is implemented. It will not always correspond on a one-to-one basis with a request at the application level. Taking again the example of a driver for a serial device, a single request at the application level consists of a call to the driver to transfer a buffer of length *n* bytes. However, the driver will split the buffer into *n* single byte transfers, each byte representing a request at the driver level. The top-half entry point would add *n* requests to the set of tracked priorities using `priot_add`*n*. As each byte is transferred, the Kernel thread would remove each request priority using `priot_remove`.

The priority of the Kernel thread would only be updated when all bytes have been transferred. It is very important that the priority tracking is based on requests as defined at the driver level, not the application level, in order for the priority tracking to work correctly.

Controlling Interrupts

One of the problems that was discussed concerning drivers that perform all interrupt processing in the interrupt handler was the fact that in certain circumstances a device can generate a series of interrupts in rapid succession. For many devices, the use of the Kernel thread and priority tracking techniques illustrated above resolves the problem.

One example concerns a disk driver. Figure 8-5 represents a situation that can occur in a system without Kernel threads. A lower priority task makes multiple requests to the driver. Before these requests have completed, a higher priority task starts execution. But this higher priority task is continually interrupted by the interrupt handler for the disk. Because of the amount of processing that can be done within the interrupt handler and because the number of requests queued up for the disk could have been very large, the response time of the system and execution time for the higher priority task is essentially unbounded.

Figure 8-6 shows the same scenario using Kernel threads. The important thing to note is that the higher priority task can be interrupted only once by the disk. The Kernel thread is responsible for starting the next operation on the disk, but because the Kernel thread's priority is based on task B's priority, it will not run until the higher priority task has completed. In addition, the length of time during which task A is interrupted by the interrupt handler is a small constant time as the majority of the interrupt processing has been moved to the Kernel thread.

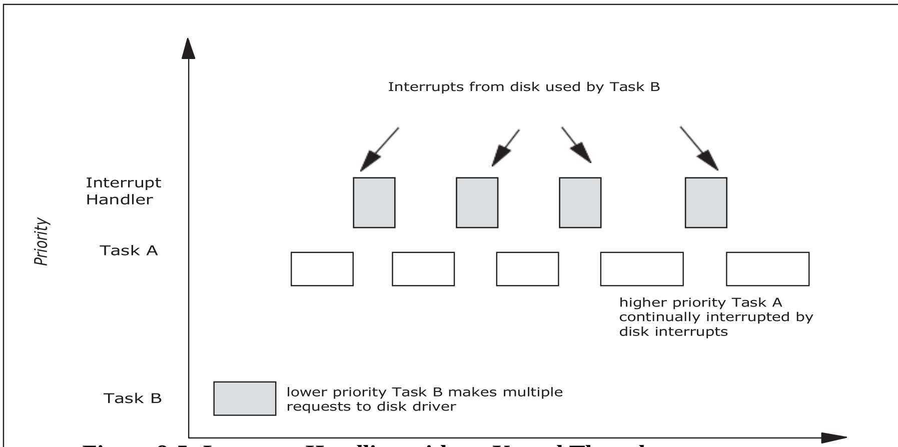


Figure 8-5: Interrupt Handling without Kernel Threads

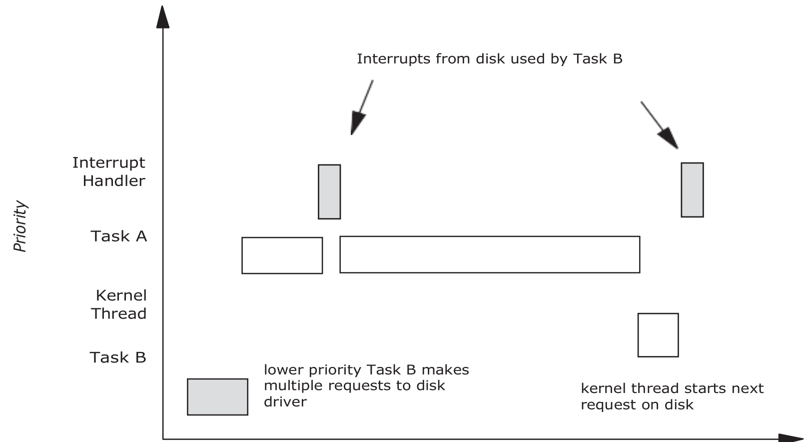


Figure 8-6: Interrupt Handling with Kernel Threads

This scheme takes care of devices where requests are generated by lower priority user tasks. But what about devices where data is being sent from a remote system? The local operating system cannot control when or how many packets are received over an Ethernet connection, for example. Or a user typing at a keyboard could generate multiple interrupts.

The solution to these situations is again based on the use of Kernel threads. For such devices, the interrupt handler must disable further interrupts from the device. Interrupts are then reenabled by the corresponding kernel thread. So again, a device can only generate a single interrupt until the thread has been scheduled to run.

Any higher priority tasks will execute to completion before the device-related thread and can be interrupted by a maximum of one interrupt from each device. The use of this technique requires that the device has the ability to store some small amount of incoming data locally during the time that its interrupt is disabled. This is not usually a problem for most devices.

Example

In this section an example of a thread-based VMPC tty driver is considered. Essentially, this driver is just another version of the one studied in detail in Chapter 7, “Interrupt Handling.” Taking the above into account, the internals of the

threaded tty driver are presented only where they differ from those of the interrupt-driven one.

In its threaded incarnation, the LynxOS-178 tty driver implements a lightweight interrupt handler whose sole purpose is to block the hardware interrupts once an interrupt is received and then signal a separate thread created at the driver installation. It is this latter thread that does the actual interrupt processing and unblocks the interrupts back once the processing is finished. Unlike the interrupt-driven tty driver, the threaded driver utilizes the UART line status register for determining the cause of hardware interrupt instead of the interrupt identification register because the latter does not allow checking for an interrupt once the interrupt has been masked out.

Statics Structure

The device driver statics structure, also referred to as the UART channel structure in this chapter, which describes the parameters of a tty channel, contains four fields that are specific to the threaded driver:

```
struct tty_u {

   /* The fields common to the interrupt-driven and
      thread-based driver implementation */

   int thr_id;
   ksem_t thr_sem;
   int intr_blocked;
   unsigned char ier;

   /* The rest of the fields common to the interrupt-driven
      and thread-based driver implementation */
};
```

In the `thr_id` field, the channel-wise thread ID is stored. It is via `thr_sem` that the interrupt handler function signals the driver thread to inform the latter about a hardware interrupt received. The boolean `inter_blocked` field is set to `true` if and only if the hardware interrupts are blocked. The `ier` field is used to cache the state of the UART interrupt enable register.

tty Driver Auxiliary Routines and Macros

```
static inline intr_block(struct tty_u *p, int intr_blocked)
{
      p->intr_blocked = intr_blocked;

      if (!p->intr_blocked) {
            __outb(p->uartp + U_IER, p->ier);
      } else {
```

```
            __outb(p->uartp + U_IER, 0);
      }
}

static inline __ier_outb(struct tty_u *p, unsigned char ier)
{
      p->ier = ier;

      if (!p->intr_blocked) {
            __outb(p->uartp + U_IER, p->ier);
      }
}

#define      __ier_orb(p, mask)      __ier_outb(p, (p)->ier | mask)

#define      ier_andb(p, mask)       ier_outb(p, (p)->ier & mask)
```

The `tty` driver auxiliary routines along with the macro definitions given above provide a convenient interface for handling the hardware device blocked state. For instance, when the `intr_block()` function is called, the truth value given as its second argument is stored in the `intr_blocked` field of the statics structure (see the description of this structure earlier in this chapter), and the hardware interrupts are either disabled altogether by writing zero to the UART interrupt enable register, or the value stored in the `ier` field of the channel structure is written back to the hardware device.

The `_ier_outb()` function stores the value of its second argument to the `ier` field of the UART channel structure pointed to by the first argument to this function, and, in case the interrupts are not blocked, writes the `ier` value to the UART interrupt enable register.

The macros_`ier_orb()` and_`ier_andb()` serve the purpose of masking or unmasking a particular interrupt, the meaning of both macros being apparent from their definitions.

The install Entry Point

```
#include <partschedu.h>            /* import: curr_vm */

struct tty_u * ttyinstall(struct tty_uinfo *info)
{
      struct tty_u *p;
      struct ttystatics *s;
      struct old_sgttyb *sg;
      __port_addr_t uartp;

      p = (struct tty_u *)SYSERR;
      if (uart_check(info->location)) {
            p = (struct tty_u *)sysbrk((long)sizeof(struct tty_u));
            p->cts = 0;
            p->uartp = uartp =_tty_uart_location(info->location);
            p->vector = info->vector;
            p->thr_sem = 0;
            p->intr_blocked = 0;
```

```
            p->ier = 0;

            sg = (struct old_sgttyb *)&info->sg;
            s = &p->channel;

            /*
               The trans_en() routine looks at uartp and
               channel.tm.c_cflag, so it needs the full
               tty_u structure passed to it.
             */
            tmgr_install(s, sg, 0, trans_en, (char *)p);
            ksem_init(&s->hsem, 1);
            ttyseth(s, (struct sgttyb *)sg, uartp, 1, 0);
            /* The device is not open yet, so disconnect the modem
               control lines. */
            __port_andb(uartp + U_MCR, ~(MCR_DTR | MCR_RTS | MCR_OT2));

            /* Start the interrupt handling thread. */
            p->thr_id = vmos_ststart((int(*)())duart_thread,
                        4096, curr_vm,
                        _getpriority()*2+1, "tty", 1, p);

            /* Mask the interrupt and register the handler. */
            __ier_outb(p, 0);
            p->int_link_id = iointset(p->vector,
                                     (int (*)())duart_int, (char *)p);
            p->init = 0;

            /* Enable the modem status interrupts. */
            __ier_orb(p, IER_MS);
      }
      return p;
}
```

Having initialized the fields of the UART channel structure, the `install` entry point starts the Kernel thread using the `ststart()` function. The priority of the Kernel thread is determined by the fourth argument of the `ststart()` routine stipulates. Finally, the `install` entry point sets the interrupt handler in just the same way as the interrupt-driven tty driver demonstrated in the previous chapter.

The Thread Function

```
static void duart_thread(struct tty_u *p)
{
   __port_addr_t uartp;
   unsigned char iir, msr, lsr, data, c;
   struct ttystatics *s;

   uartp = p->uartp;
   s = &p->channel;

   for (;;) {
      swait(&p->thr_sem, SEM_SIGIGNORE);

      lsr = __inb(uartp + U_LSR);

      /* 1. Process the receiver interrupt */
```

```
if (lsr & LSR_DR) {
   data = __inb(uartp + U_RBR);

   if (lsr & LSR_BI) {
      /* Break interrupt */
      tmgr_ex(s, EX_BREAK, 0);
   }

   /*
      The ttydrvr receives a 0 on transition
      from break on to break off.
    */
   if (!((lsr & LSR_BI) && data && (s->t_line != SLIPDISC))) {
      if ((lsr & (LSR_PE | LSR_FE)) &&
         (s->xflags1 & X1_ENABLE_INPUT_PC)) {
         /* Parity error or framing error */
         tmgr_ex(s, EX_PARITY, data);
         /* enable transmit interrupt */
         __ier_orb(p, IER_TRE);
      } else {
         if (tmgr_rx(s, data)) {
            /* enable transmit interrupt */
            __ier_orb(p, IER_TRE);
         }
      }
   }
}

/* 2. Check the modem signals status */
msr = __inb(uartp + U_MSR);
/*
  This case should only occur during an open when
  the file is configured as blocking, OR when we're
  doing hardware flow-control. Keep in mind that
  CRTSCTS means both input and output hardware
  flow-control (Linux compatible). The dcd_check
  is for the open routine to block appropriately.
*/
if ((msr & MSR_DCTS) && (p->dcd_check
   || (s->tm.c_cflag & (V_CRTSCTS | V_CRTSXOFF)))) {
   if (!(msr & MSR_CTS)) { /* DCE is not ready */
      p->cts = 0;
   } else {
      p->cts = 1;
      /* enable transmit interrupt */
      __ier_orb(p, IER_TRE);
   }
}

/* 3. Process the transmitter interrupt */
if (lsr & LSR_TRE) {                   /* transmitter empty */
   if (p->dcd_check && !p->cts) {
      /* do not send because "No CTS" */
      break;
   }
   if (tmgr_tx(s, (char *)&c)) {       /* call manager */
      /* disable transmit interrupt */
      __ier_andb(p, ~IER_TRE);
   } else {
      __outb(uartp + U_THR, c);
   }
}
```

```
        /* Restore the interrupt mask */
        intr_block(p, 0);
    }
}
```

Unlike the interrupt-driven tty driver, whose interrupt handler is designed to process hardware interrupts inside itself, the threaded driver does the same processing in a separate thread function. In the endless loop, the thread function waits on the thread semaphore `p->thr_sem` that is used to signal a hardware interrupt (see below for a description of the interrupt routine with more details). The UART line status register is used to read the information about the hardware interrupt, which is then used to take the actions appropriate for the particular interrupt type. These actions are very similar to those analyzed in the previous chapter so that the details about the workings of the interrupt handling code are left outside to keep the text concise. After processing the interrupt, the thread function unblocks the hardware interrupts and goes back waiting on the interrupt semaphore.

The Interrupt Routine

```
static void duart_int(struct tty_u *p)
{
    if ((__inb(p->uartp + U_IIR) & IIR_PEN) == 0) {
        /*
          There is an interrupt pending. Mask the
          interrupt on the device and signal the thread.
        */
        intr_block(p, 1);
        ssignal(&p->thr_sem);
    }
}
```

The interrupt handler in the threaded driver is designed to be as short and fast as possible in order to minimize the interrupt response time. If an interrupt is pending, the interrupts on the tty channel are blocked altogether and the thread semaphore is signaled to wake up the thread function. As explained in “The Thread Function” on page 112, the interrupts are unblocked by the thread function after the latter has done all the steps necessary to process the interrupt condition.

CHAPTER 9 Installation and Debugging

This chapter discusses the two methods of device driver installation in LynxOS-178: Static and Dynamic installations.

Static versus Dynamic Installation

The two methods of driver installation are as follows:

- Static Installation
- Dynamic Installation

A brief comparison of the two methods is given to ensure that the systems programmer understands the intricacies involved in each type of installation. The next sections will help in choosing the type of installation procedure to suit specific requirements.

Static Installation

With this method, the driver object code is incorporated into the image of the Kernel. The driver object code is linked with the Kernel routines and is installed during system start-up. A driver installed in this fashion can be removed (that is, made ineffective), but its text and data segments remain within the body of the Kernel.

The advantages of static installation are as follows:

- With static installation, devices are instantly available on system start-up, simplifying system administration. The initial console and root file system devices must use static installation.
- The installation procedure can be avoided each time the system reboots.

- Static linking allows the driver symbols to be visible from within the Kernel debugger.

NOTE: While neither installation method affects a device driver's functionality, Lynx Software Technologies recommends using a dynamic installation during the development of a new driver. This is because the driver can be installed statically once it is working without problems

Dynamic Installation

This method allows the installation of a driver after the operating system is booted. The driver object code is attached to the end of the Kernel image and the operating system dynamically adds this driver to its internal structure. A driver installed in this fashion can also be removed dynamically.

Static Installation Procedure

The code organization for static installation is done in the following manner.

Table 9-1: Code Organization for Static Installation

Directory	**File**	**Description**
`/`	`/sys/bsp.<bsp_name>/a.out`	LynxOS-178 Kernel, where `<bsp_name>` is the name of the BSP
`/sys/lib`	`libdrivers.a`	Drivers object code library
	`libdevices.a`	Device information declarations
`/sys/dheaders`	`<dev>info.h`	Device information definition for device <*dev*>
`/sys/devices`	`<dev>info.c`	Device configuration file for device <*dev*>
	`Makefile`	Instructions for making `libdevices.a`
`/sys/drivers/`*drvr*	driver source	The source code for the driver *drvr* to be installed

Table 9-1: Code Organization for Static Installation (Continued)

Directory	File	Description
`/sys/bsp.<bsp_name>`	`config.tbl`	Master device & driver configuration file, where `<bsp_name>` is the name of the BSP
	`a.out`	LynxOS-178 kernel binary
	`Makefile`	Instructions for making `/sys/bsp.<bsp_name>/\ a.out`
`/sys/cfg`	`driv.cfg`	Configuration file for `driv` driver and its devices.

The following steps describe how to do a static installation:

1. Create a device information definition and declaration. Place the device information definition file `devinfo.h` in the directory `/sys/dheaders` along with the existing header files for other drivers in the system.

2. Make sure that the device information declaration file `devinfo.c` is in the `/sys/devices` directory and has the following lines in the file in addition to the declaration.

```
....
#include "../dheaders/devinfo.h"
....
```

   This ensures the presence of the device information definition.

3. Compile the `<dev>info.c` file and update the `/sys/lib/libdevices.a` library file to include `<dev>info.o`. This may also be automated by adding `<dev>info.c` to the `Makefile`.

```
DEVICE_FILES=atcinfo.x dtinfo.x flopinfo.x devinfo.x
```

   To update `/sys/lib/libdevices.a`, enter:

```
make install
```

Driver Source Code

Assuming the new driver is `driver`, the following steps must be followed for driver code installation:

1. Make a new directory driver under `/sys/drivers` and place the code of the device driver there.
2. Create a `Makefile` to compile the device driver.
3. Update the library file `/sys/lib/libdrivers.a` with the driver object file using the command:

```
make install
```

Device and Driver Configuration File

The device and driver configuration file should be created with the appropriate entry points, major device declarations, and minor device declarations. The system configuration file is `config.tbl` in the `sys/bsp.`<*bsp_name*> directory.

The `config.tbl` file is used with the `config` utility to produce driver and device configuration tables for LynxOS-178. Drivers, major devices, and minor devices are listed in this configuration file. Each time the system is rebuilt, `config` reads `config.tbl` and produces a new set of tables and a corresponding `nodetab` file.

Configuration File: config.tbl

The parsing of the configuration files in LynxOS-178 follows these rules:

- The commands are designated by single letters as the first character on a line
- The field delimiter is `:`
- Spaces between the delimiter are not ignored. They are treated literally
- Blank lines are ignored

The special characters in the configuration file are

`#`	Used to indicate a comment in the configuration file. The rest of the line is ignored when this is found as the first character on any line.
`\`	Used as the continuation character to continue a line even within a comment.

: If the colon is the first character in the line, it is ignored.

`I:filename` Used to indicate that the contents of the file `filename` should replace the declaration.

The format of a device driver entry with its major and minor device declarations should look like this:

```
# Character device
C:driver name:driveropen:driverclose: \
    :driverread:driverwrite: \
    :driverselect:driverioctl: \
    :driverinstall:driveruninstall
D:some driver:devinfo::
N:minor_device1:minor_number
N:minor_device2:minor_number

# Block device
B:driver name:driveropen:driverclose: \
    :driverstrategy:: \
    :driverselect:driverioctl: \
    :driverinstall:driveruninstall
D:some driver:devinfo::
N:minor_device1:minor_number:permissions(optional)
N:minor_device2:minor_number:permissions(optional)
```

The entry points should appear in the same order as they are shown here. If a particular entry point is not implemented, the field is left out, but the delimiter should still be in place.

If above declarations are in a file `driver.cfg`, the entry:

```
I:driver.cfg
```

should be inserted into the `config.tbl` file.

Rebuilding the Kernel

To rebuild the LynxOS-178 kernel, type the following commands:

```
cd $ENV_PREFIX/sys/bsp.<bsp_name>
make install
```

where *<bsp_name>* is the name of the target BSP.

In order for the applications programs to use a device, a node must be created in the file system. For static device drivers, this is done automatically by the `mkimage` utility when creating a Kernel Downloadable Image, provided that the path to the `nodetab` file created by `config` is specified in the `mkimage` spec file.

Dynamic Installation Procedure

Dynamic installation requires a single driver object file, and a pointer to the entry points must be declared. The location of the driver source code is irrelevant in dynamic installation. In LynxOS-178, the installation of dynamically-loaded device drivers is usually done by trusted system software after system startup.

Driver Source Code

To install a device driver dynamically the entry points must be declared in a structure defined in `dldd.h`. The variable should be named `entry_points` and for a block composite driver `block_entry_points` is also required.

The format of the `dldd` structure is illustrated below:

```
#include <dldd.h>
struct dldd entry_points = { open, close, read,
                        write, select, ioctl, install,
                          uninstall, streamtab}
```

For block composite drivers, the block driver entry points are specified as:

```
struct dldd block_entry_points =
                  { b_open, b_close, b_strategy,
                  ionull, ionull, b_ioctl, b_install,
                    b_uninstall, streamtab}
```

The header file `dldd.h` must be included in the driver source code, and the declaration must contain the entry points in the same order as they appear above. If a particular entry point is not present in a driver, the field in the `dldd` structure should refer to the external function `ionull`, which is a Kernel function that simply returns `OK`. The last field in the `dldd` structure, `streamtab`, is only used for STREAMS drivers and should be set to `NULL` for other drivers.

NOTE: On the PowerPC platform, the `dldd` structures should not be declared static.

The following example shows the null device driver that will be installed dynamically.

```
/* --------------  NULLDRVR.C -------------------*/

#include <conf.h>
#include <kernel.h>
#include <file.h>
#include <dldd.h>

extern int ionull ();

int nullread(void *s, struct file *f, char *buff, int count)
{
    return 0;
}
int nullwrite(void *s, struct file *f, char *buff, int count)
{
    return (count);
}
int nullioctl()
{
    pseterr (EINVAL);
    return (SYSERR);
}
int nullselect()
{
    return (SYSERR);
}
int nullinstall()
{
    return (0);
}
int nulluninstall()
{
    return (OK);
}
struct dldd entry_points = {
                ionull,
                 ionull,
                nullread,
                nullwrite,
                nullselect,
                nullioctl,
                nullinstall,
                nulluninstall,
                (char *) 0
};
```

Note that calls to a driver entry point replaced by `ionull` will still succeed. If a driver does not support certain functionality, then it must include an entry point that explicitly returns `SYSERR`, as in the case of the `ioctl` and `select` entry points in the above example. This will cause calls to these entry points from an application task to fail with an error.

NOTE: To dynamically install the null driver on the PowerPC platform, omit the keyword `static` from the `struct dldd` declaration.

Driver Installation

In this release of LynxOS-178, follow these recommendations for compiling and installing dynamic device drivers on a specific LynxOS-178 platform.

LynxOS-178 supports dynamic driver compilation with the GNU C compiler.

1. Compile the driver with GNU C.

```
gcc  -c -o driver.o driver.c
     -I/sys/include/kernel
     -I/sys/include/family/<arch> -D__LYNXOS
```

   where *<arch>* is the architecture (for example, `x86`).

2. Now create a dynamically loadable object module by entering the following (all on one line):

```
ld -o driver.obj driver.o
```

Once a dynamic driver object module has been created, this object module can now be dynamically installed. LynxOS-178 provides a `drinstall` utility that allows driver installation from the command line. This utility is for the Development Environment only.

For character device drivers, enter:

```
drinstall -c driver.obj
```

For block device drivers, enter:

```
drinstall -b driver.obj
```

If successful, `drinstall` utility returns the unique `driver-id` that is defined internally by the LynxOS-178 kernel. For the block composite driver, the `driver-id` returned will be a logical `OR` of the character `driver-id` in the lower 16 bits and the block `driver-id` in the upper 16 bits.

It is also possible to use a program to install a driver by using the system call `dr_install()`.

For a character device driver, use:

```
dr_install("./driver.obj", CHARDRIVER);
```

For a block device driver, use:

```
dr_install("./driver.obj", BLOCKDRIVER);
```

Device Information Definition and Declaration

The device information definition is created the same way as in the static installation. To create a device information declaration, one can use `objcopy` utility to instantiate the device information definition.

Assuming the device information definition appears as:

```
/* myinfo.h */
struct my_device_info {
    int address;
    int interrupt_vector;
};
/* myinfo.c */
struct my_device_info devinfo = { 0xd000, 4 };
```

Compile `myinfo.c` into an object file `myinfo.o`:

```
gcc -c myinfo.c
```

The device information file can then be created using:

```
objcopy -O binary myinfo.o my.info
```

Device Installation

The installation of the device should be done after the installation of the driver. The two ways of installing devices are either through the `devinstall` utility program or `cdv_install` and `bdv_install` systemcalls.

```
devinstall -c -d driver_id mydevice_info
devinstall -b -e raw_driver_id -d
block_driver_id mydevice_info
```

The `driver_id` is the identification number returned by the `drinstall` command or `dr_install()` system call. This installs the appropriate device with the corresponding driver and assigns a major device number to it (in this case, we will assume this is `major_no`).

Node Creation

Unlike the static installation, there is no feature to automatically generate the nodes under dynamic installation. This could be done manually using the `mknod` command or system call. (See the *LynxOS-178 User's Guide*.) For LynxOS-178, this is usually done by trusted system software.

Since the root file system in LynxOS-178 is always read-only, by convention, a RAM disk is typically created and mounted under `/dev/ddev`, and the dynamic device node is created in the `/dev/ddev` directory. The creation of the nodes allows application programs to access the driver by opening and closing the file that has been associated with the driver through the `mknod` command.

`mknod /dev/ddev/`*`device`* **`c`** *`major_no minor_no`*

The *`major_no`* is the number assigned to the device after a `devinstall` command. This can be obtained by using the `devices` command. The *`minor_no`* is the minor device number which can be specified by the user in the range of 0–255. The **`c`** indicating a character device could also be a **`b`** to indicate a block device.

Device and Driver Uninstallation (Development Environment Only)

Dynamically loaded device drivers can be uninstalled when they are no longer needed in the system. This can help in removing unwanted code in physical memory when it is no longer relevant. Removal is performed with the `drinstall` command. However, the device attached to the driver has to be uninstalled before uninstalling the driver. Removing the device is accomplished with the `devinstall` command.

For character devices:

```
devinstall -u -c device_id
```

For block devices:

```
devinstall -u -b device_id
```

After the device is uninstalled the driver can be uninstalled using the command:

```
drinstall -u driver_id
```

Common Error Messages During Dynamic Installation

The following list describes some common error messages that may be encountered during dynamic installation of a device driver. In this case, the LynxOS-178 Kernel assists in debugging efforts by printing help messages to the system console.

- Bad Exec Format

  This is usually seen when a `drinstall` command is executed. If this happens, it indicates that a symbol in the device driver has not been resolved with the Kernel symbols. Make sure that there are no symbols that cannot be resolved by the kernel and that the structure `dldd` has been declared inside the driver. This error can also be caused by the driver binary being stripped, or if "resident=true" is specified in the spec file and the "executable" permission is set for the driver binary.

- Device Busy

  This error message is seen when attempting to uninstall the driver before uninstalling the device. The correct order is to uninstall the device before uninstalling the driver.

NOTE: A driver cannot be dynamically installed on a Kernel that has been stripped.

Debugging

The development of device drivers can become complicated unless proper debugging facilities are available. Since device drivers are not attached to a particular control terminal, ordinary `printf()` statements will not work. LynxOS-178 provides a device driver service routine to debug device drivers effectively. The routine is `kkprintf()`. This routine has exactly the same format as the `printf()` call. The `KKPF_PORT` preprocessor macro defined in the `/sys/bsp.`<*bsp_name*>`/uparam.h` header file controls where the output of the `kkprintf()` routine is sent. Note that the LynxOS-178 Kernel needs to be rebuilt each time the value of this macro is changed. Please refer to the values of the `SKDB_C*` family of macros defined in the same header file for the options that can be used for the definition of the `KKPF_PORT` macro.

NOTE: Configuring the `kkprintf()` port may not be supported by a particular BSP, in which case the `kkprintf()` output always goes to the port hard-coded in the BSP. Usually, it is the primary serial port or the console.

The device driver support routine `cprintf()` will print to the console. Unlike `kkprintf()`, `cprintf()` cannot be used in an interrupt routine or where interrupts or preemption is disabled.

The `kkprintf()` routine can also be used in interrupt routines as it polls the UART for it to be ready for the next character.

The Simple Kernel Debugger (SKDB) can also be used to debug device drivers. It allows breakpoints to be set and can display stack trace and register information.

However, since the symbol table of a dynamic device driver is not added to the symbol table of the Kernel, SKDB may not be useful for debugging dynamic device drivers.

The debugging routines and SKDB are available only in the Development Environment.

Hints for Device Driver Debugging

- Insert some `kkprintf()` statements in the `install()` routine of the new driver. After the `devinstall` command is executed, the `install()` entry point is invoked. This is a good way of identifying that the `install()` entry point is being invoked at all and if the device information declaration passed is being received properly.
- Insert `kkprintf()` statements in the `uninstall()` entry point. After every deinstallation the `uninstall()` entry point is invoked. The `kkprintf()` statements should be seen after deinstallation.
- Initially, it is advisable to put a `kkprintf()` statement at the beginning of every entry point to make sure it is being invoked properly. Once all the errors have been found, they can be removed.
- Using `kkprintf()` and `cprintf()` statements for debugging can affect the timing characteristics of the driver and may mask timing related problems. A way to reduce this debugging overhead involves having the driver write status information to an internal chunk of memory. When a failure occurs, use SKDB to investigate this area in memory.
- Use a serial connection to a second machine running kermit to capture debugging output from SKDB.

Additional Notes

- Statically installed device drivers in LynxOS-178 can also be uninstalled dynamically. However, the memory reclamation of the `TEXT` section is not done in this case.
- Symbols from dynamically-loaded device drivers cannot be used for symbol resolution in other dynamically loaded device drivers. If there are two dynamically-loaded device drivers using function `f()`, the code for function `f()` has to be present in both the drivers' source code. This is

because if one of the drivers is loaded initially, function `f()` does not get resolved with the second device driver even though it is in memory. Thus, only statically-loaded drivers' symbols are resolved with dynamic drivers

- Garbage collection is not provided in LynxOS-178. Thus, any memory that is dynamically allocated must be freed before uninstalling the device driver.

CHAPTER 10 Device Resource Manager

The Device Resource Manager (DRM) is a LynxOS-178 module that functions as an intermediary between the operating system, device drivers, and physical devices and buses. The DRM provides a standard set of service routines that device drivers can use to access devices or buses without having to know device or bus-specific configuration options. DRM services include device identification, interrupt resource management, device I/O to drivers, and device address space management. The DRM also supports dynamic insertion and deletion of devices.

The functionality of DRM in LynxOS-178 is reduced compared to LynxOS. In particular, the user-level interface to DRM is removed.

This chapter introduces DRM concepts and explains DRM components. Sample code is provided for DRM interfaces and services. The PCI bus layer is described in detail with a sample driver and application. This chapter provides information on the following topics:

- DRM concepts
- DRM service routines
- Using DRM facilities from device drivers
- Advanced topics
- PCI bus layer
- Example driver

DRM Concepts

Device Tree

The device tree is a hierarchical representation of the physical device layout of the hardware. DRM builds a device tree during Kernel initialization. The device tree is made up of nodes representing the I/O controllers, host bridges, bus controllers, and bridges. The root node of this device tree represents the system controller

(CPU). There are two types of nodes in the device tree: DRM bus nodes and DRM device nodes.

DRM bus nodes represent physical buses available on the system, while DRM device nodes represent physical devices attached to the bus.

The DRM nodes are linked together to form parent, child, and sibling relationships. A typical device tree is shown in Figure 10-1. To support Hot Swap environments, DRM nodes are inserted and removed from the device tree, mimicking Hot Swap insertion and extraction of system devices.

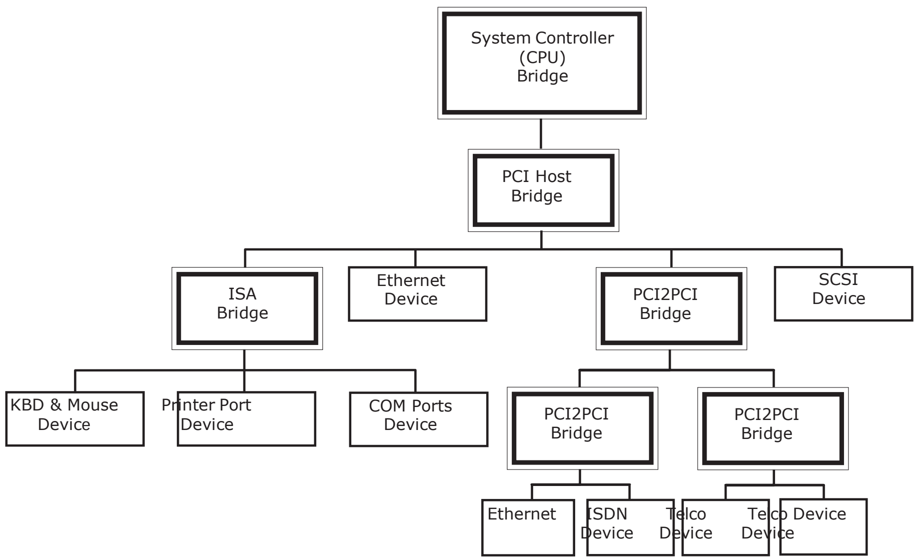


Figure 10-1: Device Tree

DRM Components

A module view of DRM and related components is shown in Figure 10-2. A brief description of each module is given below the figure.

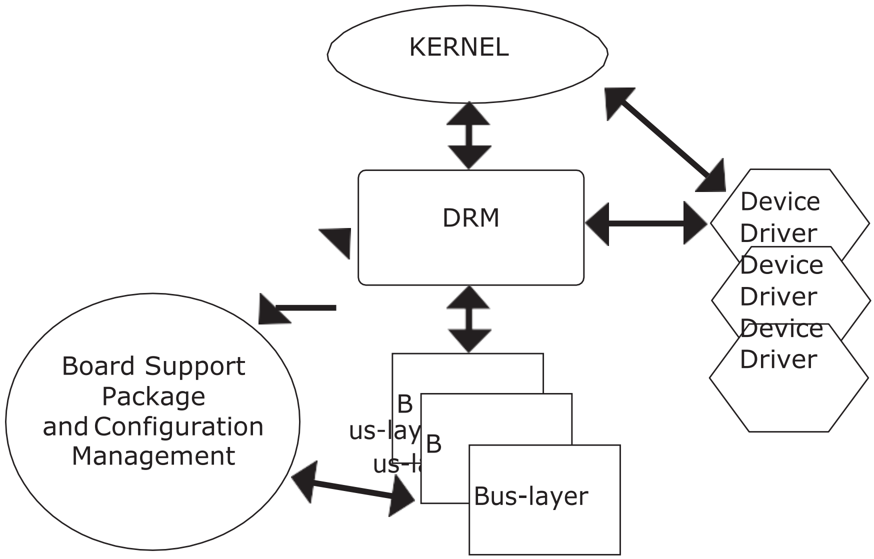


Figure 10-2: Module View

- DRM— DRM provides device drivers with a generalized device management interface.
- Kernel— The LynxOS-178 kernel provides service to applications and device drivers. DRM uses many of the Kernel service routines.
- Bus Layer— These modules perform bus-specific operations. DRM uses the service routines of the bus layer to provide service to the device drivers.
- Device Driver— These modules provide a generic application programming interface to specific devices.
- BSP— The Board Support Package (BSP) provides a programming interface to the specific hardware architecture hosting LynxOS-178. This module also provides device configuration information to other modules.

Bus Layer

DRM uses bus layer modules to support devices connected to many different kinds of buses. There are numerous bus architectures, many of which are standardized.

Typical bus architectures seen in systems are the ISA, PCI, and VME standards, however, proprietary bus architectures also exist. DRM needs a specific bus layer module to support a specific kind of bus architecture. The device drivers use DRM service routines to interface to the bus layers. The bus layers interface with the BSP to get board-specific information.

The bus layers provide the following service routines to DRM:

- Find bus nodes and device nodes.
- Initialize bus and device nodes.
- Allocate resources for bus and device nodes.
- Free resources from bus and device nodes.
- Map and unmap a device resource.
- Perform device I/O.
- Insert a bus or device node.
- Remove a bus or device node.

LynxOS-178 supports only one bus layer which is used for managing PCI and Compact PCI devices. Some of the DRM functions described later in this chapter require the bus layer ID. The correct symbol to use is `PCI_BUSLAYER`.

DRM Nodes

A DRM node is a software representation of the physical device. Each node contains fields that provide identification, device state, interrupt routing, bus-specific properties, and links to traverse the device tree. DRM service routines are used to access the DRM node fields. These routines provide device drivers access to DRM facilities via a standard interface. This eliminates the need to know implementation details of the specific software structure. Some of the important fields of the DRM node are shown in the next table.

NOTE: Subsequent coding examples in this chapter make reference to a data structure of type `drm_node_s`. This structure is a data item used internally by the LynxOS-178 Kernel as the software representation of a DRM node and is not intended to be accessed at the driver or user level. LynxOS-178 does not export a definition of this structure. The coding examples use opaque pointers which are passed around and are not meant to be dereferenced.

Table 10-1: DRM Node Fields

Field Name	Description
`vendor_id`	This field is used for device vendor identification.
`device_id`	This field identifies the DRM node.
`vendor_id2`	This field is the second vendor identifier of the device.
`device_id2`	This field contains information on the second device identifier.
`pbuslayer_id`	This field identifies the primary bus layer of the bus/device node.
`sbuslayer_id`	This field identifies the secondary bus layer of a bus node.
`node_type`	This field indicates the node type: bus node or device node, and indicates if it is statically configured or dynamically configured.
`drm_state`	This field describes the life cycle state of the DRM node. DRM nodes include: IDLE, SELECTED, READY, or ACTIVE.
`parent`	This field links this node to its parent node. The root node has this field set to NULL to indicate that it has no parent.
`child`	This field links to the child node of this bus node. Only bus nodes have children.
`sibling`	This field links to the sibling node of this DRM node. The last sibling of a bus has this field set to NULL.
`intr_flg`	This field indicates if the device raises an interrupt to request service.
`intr_cntlr`	If the device uses interrupt service, this field indicates the controller to which the device is connected.
`intr_irq`	This indicates the interrupt request line of the controller to which this device is connected.
`drm_tstamp`	This field indicates when this node was created.
`prop`	This field links to bus-specific properties of the device.

DRM Node States

The status of a DRM node is indicated by its state. Initially, a DRM node is set to IDLE when it is created. Devices that are removed from the DRM tree, or undetected devices are considered UNKNOWN. The UNKNOWN state is not used by DRM, but it is used to denote a device that is unrecognized to DRM. The following diagram details the stages of DRM node states.

DRM Node State	Description
UNKNOWN	Devices that exist in the system, but are not known to DRM are considered UNKNOWN. DRM does not use this state, but it is useful for users to think of all devices unrecognized by DRM to be in this state. Deleting a DRM node makes the device UNKNOWN to DRM. Once the DRM has initialized a device node, it is considered IDLE.
IDLE	A device is discovered/inserted into the DRM tree and initialized to a minimal extent. The `buslayer_id`, `device_id`, and `vendor_id` node fields identify the node. Some of its bus-specific properties are initialized. System resources are not allocated to the device. Devices for which drivers are not available, or devices that are not needed, are left in this state. IDLE nodes are created by the `drm_locate()`, or the `drm_insertnode()` service routines. IDLE nodes are deleted by the `drm_delete_node()` service routine.
SELECTED	Devices needed by the system are set to the SELECTED state and resources are allocated to them. In Hot Swap and high availability, environments , configuration-specific policies control the selection of nodes. IDLE DRM nodes are selected by the `drm_select_node()` service routine. SELECTED nodes are set to the IDLE state by the `drm_unselect_node()` service routine.
READY	When resources are allocated to the SELECTED node, it is set to READY. A node in READY state is fully initialized, but not in use. System resources and bus resources are assigned to this node. In case the resource allocation for the DRM node fails, the node remains in the SELECTED state. A SELECTED node is set to the READY state by the service routine `drm_alloc_resource()`. The READY nodes are put into a SELECTED state by the `drm_free_resource()` service routine.
ACTIVE	A DRM node in use by a device driver is in the ACTIVE state. Either the `drm_get_handle()` or `drm_claim_handle()` service routines make a DRM node ACTIVE. A driver releases a device by using the `drm_free_handle()` service routine. This causes the DRM node to go into a READY state.

Figure 10-3: DRM Node States

DRM Initialization

The DRM module is initialized during LynxOS-178 Kernel initialization. DRM builds a device tree of all visible devices and brings them up to a READY state, if possible. This is to enable all statically linked drivers to claim the DRM nodes and bring up the basic system service routines. Some DRM nodes may be left in the SELECTED state after Kernel initialization is complete. Typically, this can be the result of unavailable resources.

DRM Service Routines

DRM service routines are used by device drivers to identify, set up, and manage device resources. Typically, they are used in the `install()` and `uninstall()` entry points of the device driver. Device drivers locate the device they need to service and obtain an identifying handle. This handle is used in subsequent DRM calls to reference the device. The table below gives a brief description of each service routine and typical usage. See the DRM man pages for more details. Additionally, see “Example Driver” on page 156.

Table 10-2: Summary of DRM Services

Service	Description	Usage
`drm_get_handle`	Searches for a DRM node with a specific vendor and device identification and claims it for use	All Drivers
`drm_free_handle`	Releases a DRM node and makes it READY	All Drivers
`drm_register_isr`	Sets up an interrupt service routine for Legacy interrupts	All Drivers
`drm_unregister_isr`	Clears an interrupt service routine for Legacy interrupts	All Drivers
`drm_msi_alloc_msg`	Allocate messages to a MSI/MSI-X capable device	All Drivers
`drm_msi_free_msg`	Free messages allocated to a MSI/MSI-X capable device	All Drivers
`drm_msi_register_isr`	Sets up an interrupt service routine for MSI/MSI-X interrupts	All Drivers
`drm_msi_unregister_isr`	Clears an interrupt service routine for MSI/MSI-X interrupts	All Drivers
`drm_map_resource`	Creates an address translation for a device resource	All Drivers

`drm_unmap_resource`	Removes an address translation for a device resource	All Drivers
`drm_device_read`	Performs read on a device resource	All Drivers
`drm_device_write`	Performs write on a device resource	All Drivers

Table 10-2: Summary of DRM Services (Continued)

Service	Description	Usage
drm_locate	Locates and builds the DRM device tree; It probes for devices and bridges recursively and builds the DRM subtree	General Device Management
drm_insertnode	Inserts a DRM node with specific properties. Only a single node is added to the DRM tree by this service routine.	General Device Management
drm_delete_subtree	Removes a DRM subtree. Only nodes in the IDLE state are removed.	General Device Management
drm_prune_subtree	Removes a DRM subtree. Nodes in the READY state are brought to the IDLE state and then deleted.	General Device Management
drm_select_node	Selects a node for resource allocation	General Device Management
drm_select_subtree	Selects a DRM subtree for resource allocation. All the nodes in the subtree are SELECTED.	General Device Management
drm_unselect_node	Ignores a DRM node for resource allocation	General Device Management
drm_unselect_subtree	Ignores an entire DRM subtree for resource allocation	General Device Management
drm_alloc_resource	Allocates a resource to a DRM node or subtree	General Device Management
drm_free_resource	Frees a resource from a DRM node or subtree	General Device Management
drm_claim_handle	Claims a DRM node, given its handle. The DRM node is now ACTIVE.	General Device Management
drm_getroot	Gets the handle to the root DRM node	General Device Management
drm_getchild	Gets the handle to the child DRM node	General Device Management
drm_getsibling	Gets the handle to the sibling DRM node	General Device Management
drm_getparent	Gets the handle to the parent DRM node	General Device Management

Table 10-2: Summary of DRM Services (Continued)

Service	Description	Usage
drm_getnode	Gets the DRM node contents	General Device Management
drm_setnode	Sets the DRM node contents	General Device Management

Interface Specification

Device drivers call DRM service routines like any standard Kernel service routine. The following table provides a synopsis of the service routines and their interface specification. Refer to LynxOS-178 man pages for a complete description.

Table 10-3: DRM Service Routine Interface Specification

Name	Synopsis
drm_locate()	int drm_locate(drm_node_handle node_h)
drm_insertnode()	int drm_insertnode(drm_node_handle parent_h void *prop_h, drm_node_handle *new_h)
drm_delete_subtree()	int drm_delete_subtree(drm_node_handle node_h)
drm_prune_subtree()	int drm_prune_subtree(drm_node_handle node_h)
drm_select_subtree()	int drm_select_subtree(drm_node_handle node_h)
drm_unselect_subtree()	int drm_unselect_subtree(drm_node_handle node_h)
drm_select_node()	int drm_select_node(drm_node_handle node_h)
drm_unselect_node()	int drm_unselect_node(drm_node_handle node_h)
drm_alloc_resource()	int drm_alloc_resource(drm_node_handle node_h)
drm_free_resource()	int drm_free_resource(drm_node_handle node_h)
drm_get_handle()	int drm_get_handle(drm_id_t bus_type, drm_id_t vend_id, drm_id_t dev_id, drm_node_handle *node_h)
drm_claim_handle()	int drm_claim_handle(drm_node_handle node_h)
drm_free_handle()	int drm_free_handle(drm_node_handle node_h)
drm_register_isr()	int drm_register_isr(drm_node_handle node_h, void (*isr)(void *), void *arg)
drm_unregister_isr()	int drm_unregister_isr(drm_node_handle node_h)
drm_msi_alloc_msg()	int drm_msi_alloc_msg(drm_node_handle node_h, int num_req_msg, int *num_assign_msg)
drm_msi_free_msg()	int drm_msi_free_msg(drm_node_handle node_h)
drm_msi_register_isr()	int drm_msi_register_isr(drm_node_handle node_h, void (*isr)(void *), void *arg, int msg_offset)

`drm_msi_unregister_isr()`	`int drm_msi_unregister_isr(drm_node_handle node_h, int msg_offset)`
`drm_map_resource()`	`int drm_map_resource(drm_node_handle node_h, int resource_id, unsigned int *vaddr)`

Using DRM Facilities from Device Drivers

Device Identification

In the `install()` device driver entry point a driver attempts to connect to the device it intends to use. To locate its device, the driver needs to use the `drm_get_handle()` service routine. `drm_get_handle()` returns a pointer to the DRM node handle via its `handle` argument. The driver specifies the device it is interested in by using `drm_get_handle()` in the following manner:

```
int install() {

    ret  = drm_get_handle(buslayer_id,
                vendor_id, device_id, &handle)
    if(ret)
    {
    /* device not found .. abort installation */
    }
}
```

It is possible to supply a wild card to `drm_get_handle()` using `vendor_id = -1` and `device_id = -1` as parameters. This claims and returns the first READY device in an unspecified search order. The driver examines the properties of the device to perform a selection. The driver needs to subsequently release the unused devices.

It is also possible to navigate the device tree using traversal functions and obtain handles for the nodes. Device selection is performed by other modules, drivers or system management applications. If device selection has been done by some other means, the driver claims the device by using the `drm_claim_handle()` service routine, taking the node handle as a parameter.

```
int install() {

    /* handle obtained externally */
    ret = drm_claim_handle(handle);
    if(ret)
    {
    /* Cannot claim device  -- abort device install */
    }
    ..
}
```

The drm_free_handle() service routine is used to release the handle. The release of the device is typically done in the uninstall() routine of the driver. The drm_free_handle() takes the node handle to be freed as a parameter.

```
int uninstall() {

    ret = drm_free_handle(handle);
    if(ret)
    {
    /* Error freeing handle, perhaps handle is bogus? */
    }
}
```

In Hot Swap environments, system management service routines select, make devices ready, and provide node handles for drivers to claim and use. The system management service routines facilitates the selection and dynamically loading of needed drivers and provides them with node handles for use.

Device Interrupt Management

PCI devices can use one of the following interrupt delivery methods:

- Legacy Interrupts using an interrupt line (pin),
- Message Signaled Interrupts (MSI and MSI-X types).

DRM maintains all interrupt routing data for a device node.

In case of Legacy interrupts, the driver uses the drm_register_isr() service routine to register an interrupt service routine and the drm_unregister_isr() service routine to clear a registration.

In case of MSI/MSI-X, the driver can register more than one service routine (up to 32 for MSI and up to 2048 for MSI-X). The driver can check if a device is MSI/MSI-X capable reading the PCI_RESID_PCICAP PCI resource. If a device is MSI/MSI-X capable, the driver can get the number of requested messages reading the PCI_RESID_MSG_REQ PCI resource. The driver uses the drm_msi_alloc_msg() service routine to allocate required number of messages and the drm_msi_free_msg() service routine to free allocated messages. The driver uses the drm_msi_register_isr() service routine to register each interrupt service routine and the drm_msi_unregister_isr() service routine to clear its registration.

Typically, these service routines are used in the install() and uninstall() entry points of the driver. To support sharing of interrupts in a hot swap/high availability environment, DRM internally dispatches all ISRs sharing an interrupt. There is no need to use the iointlink() service routine for interrupt sharing if an interrupt service routine is registered with the drm_register_isr() service routine or the drm_msi_register_isr() service routine.

The following code segments illustrate the use of these DRM service routines for Legacy interrupts:

```
int install() {

    int ret, link_id;
    ret = drm_get_handle(buslayer_id, vendor_id,
                device_id, &handle);
    link_id = drm_register_isr(handle, isr_func, args);
}

int uninstall() {

    ret = drm_unregister_isr(handle);
    if(Ret)
    {
    /* Cannot unregister isr? bogus handle? */
    }
    ret = drm_free_handle(handle);
    if(ret)
    {
    /* Cannot free handle, is handle bogus? */
    }
}
```

The interrupt management service routines return a status message when applied to a polled mode device.

Device Address Space Management

Many devices have internal resources that need to be mapped into the processor address space. The bus layers define such device-specific resources. For example, the configuration registers, the bus number, device number, and the function number of PCI devices are considered resources. The bus layer defines resource IDs to identify device-specific resources. Some of the device resources may need to be allocated. For example, the base address registers of a PCI device space need to be assigned a unique bus address space. DRM provides service routines to map and unmap a device resource into the processor address space. The function `drm_map_resource()` takes as parameters the device handle, resource ID and a pointer to store the returned virtual address. The `drm_unmap_resource()` takes as parameters a device handle and resource ID.

The following code fragment illustrates the use of these service routines:

```
int install() {
    ret = drm_get_handle(PCI_BUSLAYER,SYMBIOS_VID,NCR825_ID,
                &handle);
    if(ret)
    {
    /* Cannot find the scsi controller */
        }
    link_id = drm_register_isr(handle,scsi_isr,
                scsi_isr_args);
    ret = drm_map_resource(handle,PCI_BAR1,
                &scsi_vaddr);
    if(ret)
    {
    /* Bogus resource_id ? */
    /* resource not mappable  2*/
    /* invalid device handle ? */
    }
    scsi_control_regs =
                (struct scsi_control *)(scsi_vaddr);
    ret =drm_unmap_resource(handle,PCI_BAR1);
    if(ret)
    {
    /* Bogus handle */
    /* resource is not mappable */
    /* resource was not mapped */
    /* invalid resource_id */
    }
}
```

Device IO

DRM provides service routines to perform read and write to the bus layer-defined resources. The `drm_device_read()` service routine allows the driver to read a device-specific resource. The `drm_device_write()` service routine allows the driver to perform a write operation to a device-specific resource. The resource IDs are usually specified in a bus layer-specific header file. For example, the file `pci_resource.h` defines the `PCIBUSLAYER` resources. Both these service routines use the `handle`, `resource ID`, `offset`, `size`, and a buffer as parameters. The meaning of the `offset` and `size` parameter is defined by the bus layer. Drivers implement platform-independent methods of accessing device resources by using these service routines. The following code fragment illustrates the use of these service routines.

```
#include <pci_resource.h>
...

/* Enable PCI_IO_SPACE */
ret = drm_device_read(handle,PCI_RESID_CMDREG,0,0,
        &pci_cmd);
if(ret)
{
/* could not read device resource? validate parameters? */
}
pci_cmd |= PCI_IO_SPACE_ENABLE;
ret = drm_device_write(handle,PCI_RESID_CMDREG,0,0,
        &pci_cmd);
if(ret)
{
/* could not write device resource? validate parameters? */
}
```

This code is platform-independent. The service routines take care of endian conversion, serialization, and other platform-specific operations.

DRM Tree Traversal

DRM provides a set of functions to navigate the device tree. Most of these functions take a reference node as input and provide a target node as output. The functions are listed below:

`drm_getroot(&handle)` returns the root of the device tree in `handle`.

`drm_getparent(node,&handle)` returns the parent of node in `handle`.

`drm_getchild(node,&handle)` returns the child of node in `handle`.

`drm_getsibling(node,&handle)` returns the sibling of node in `handle`.

Device Insertion/Removal

DRM provides two service routines that add nodes to the DRM tree: `drm_locate()` recursively finds and creates DRM nodes given a parent node as reference; `drm_insertnode()` inserts one node. The `drm_insertnode()` service routine is used when sufficient data is known about the device being inserted. The `drm_locate()` service routine is used to build entire subtrees.

A typical application involves inserting the bridge device corresponding to a slot, using the `drm_insertnode()` service routine. For a given configuration, the geographic data associated with the slots is generally known. This data is used to insert the bridge device. The data that is needed to insert a node is bus layer-specific. For the `PCIBUSLAYER`, the PCI device number and function number are provided. The reference parent node determines the bus number of the node being inserted. Also, the bus layer determines the location of the inserted node in the DRM tree. Once the bridge is inserted, the `drm_locate()` service routine is used to recursively build the subtree below the bridge.

The `drm_locate()` and `drm_insertnode()` service routines initialize the DRM nodes to the IDLE state. The `drm_selectnode()` or `drm_select_subtree()` service routines are used to select the desired nodes and sets the nodes to the SELECTED state. The `drm_alloc_resource()` service routines are used to set the nodes to a READY state. DRM nodes in the READY state are available to be claimed by device drivers. After being claimed, the node is set to the ACTIVE state.

During extraction, device drivers release the DRM node using the `drm_free_handle()` service routine. This brings the DRM node back to a READY state. Resources associated with the nodes are released by using the `drm_free_resource()` service routine. This sets the nodes to the SELECTED state. The DRM nodes are then put in an IDLE state by using the `drm_unselect_subtree()` or `drm_unselect_node()` service routines. The IDLE nodes are removed by using the `drm_delete_subtree()` or `drm_delete_node()` service routines. This last operation puts the device back into an unknown state. The device is now extracted from the system. A convenience function, `drm_prune_subtree()`, removes DRM's knowledge of an entire subtree. This routine operates on subtrees that are in the READY state.

When DRM nodes are inserted, they are time-stamped to assist in locating recently inserted nodes.

Advanced Topics

Adding a New Bus Layer

This section describes the entry points and the Application Programming Interface (API) between the DRM module and the bus layer. The information in this section is useful for implementing a new bus layer or enhancing an existing bus layer.

The DRM invokes functions in the bus layer by means of a bus handle. The bus handle is a pointer to a structure containing function pointers that implement the bus layer functionality. The file `drm.h` defines the bus handle as:

```
struct  drm_bushandle_s {

    int (*init_buslayer)(); /* Initialize the buslayer */
    int (*busnode_init)();   /* Initialize a bus node */
    int (*alloc_res_bus)();  /* Allocate resource to
                                a bus node */
    int (*alloc_res_dev)();  /* Allocate resources
                                to a device node */
    int (*free_res_bus)();   /* Free resources
                                from a bus node */
    int (*free_res_dev)();   /* Free resources from a
                                device node */
    int (*find_child)();/* Find children of a bus node*/
    int (*map_resource)();   /* Map a device resource
                                into kernel virtual
                                address space */
    int (*unmap_resource)(); /* Unmap a device resource from
                                kernel address space */
    int (*read_device)();    /* Perform a read operation
                                on a device resource */
    int (*write_device)();   /* Perform a write operation
                                on the device resource */
    int (*translate_addr)(); /* address translation service */
    int (*del_busnode)();    /* Remove a bus node */
    int (*del_devnode)();    /* Remove a device node */
    int (*insertnode)();     /* Insert a new drm node */
    int (*ioctl)();          /* buslayer specific ioctl functions */
};
```

The DRM maintains an array of `drm_bushandles` registering all the known bus handles. The index into this array is the bus layer ID for the bus layer. A new bus layer is registered by adding a new entry to this array. A NULL entry indicates the end of the list. The file `drm_conf.c` has this configuration information.

init_buslayer ()

This entry point is invoked during the bus layer initialization. This function is called only once. The bus layer is supposed to initialize bus layer-specific data. For example, any built-in resource allocators are initialized, tables are allocated, devices are turned on or off, and so forth. `init_buslayer()` takes no parameters. This entry point is invoked when the `drm_init()` service routine is called.

busnode_init ()

This entry point is invoked to initialize a bus-node. Bus layer-specific data is allocated during this call. The hardware must be configured to allow further probes behind this node. For example, the `PCIBUSLAYER` needs to allocate the secondary bus number and the subordinate bus number of PCI-to-PCI Bridges during `busnode_init()`. `busnode_init()` takes a DRM node handle as a parameter.

```
int busnode_init(struct drm_node_s *handle);
```

The secondary bus layer ID determines which bus layer is invoked to call `busnode_init()`. This entry point is invoked during recursive probes and when a bus node is inserted.

alloc_res_bus ()

This entry point is called as a result of invoking the `drm_alloc_resource()` service routine on this bus node. The bus layer must allocate node-specific resources like IO windows and bus space.

`alloc_res_bus()` takes a DRM node handle as the parameter.

```
int alloc_bus_res(struct drm_node_s *handle);
```

The secondary bus layer ID determines which bus layer is invoked by this call to `alloc_res_node()`. This entry point is invoked every time the node transits from the SELECTED state to the READY state. Take care of any previous history of this node by clearing unused registers.

alloc_res_dev ()

This entry point is called as a result of invoking the `drm_alloc_resource()` service routine on this node. The bus layer must allocate device-specific resources, like bus address, physical address, interrupt controller ID, interrupt request line, and `interrupt_needed` flag. The device needs to be programmed with the allocated resources so that it is in the READY state. The DRM node handle is passed as a parameter to this call. For example, the base address registers are allocated and programmed in a PCI device as a result of this call:

```
int alloc_dev_res(struct drm_node_s *handle);
```

free_res_bus ()

This entry point is called as a result of invoking the `drm_free_resource()` service routine on this bus node. The bus layer must free resources allocated like IO windows, bus space, and so forth. This function should return an error if any of its children have resources still allocated. The DRM node handle for the bus node is passed in as an input parameter to this call:

```
int free_res_bus(struct drm_node_s *handle);
```

free_res_dev ()

This entry point is invoked as a result of invoking `drm_free_resource()` on this device node. The bus layer should free any resources allocated specific to the device such as bus address space, interrupt request line, and so forth. The DRM node handle is passed in as an input parameter to this call:

```
int free_res_dev(struct drm_node_s *node_h);
```

find_child ()

This entry point is used by DRM to probe and find a child node. The find entry point is invoked with the following arguments:

```
int find_child(drm_node_handle parent_h,
    drm_node_handle ref_h,
    drm_node_handle *child_h);
```

The DRM keeps a list of sibling nodes linked singly. `find_child()` needs to enumerate the bus and find a node between the `ref_h` and `ref_h->next`. If `ref_h` is NULL, `find_child()` needs to see if there are any new nodes at the head of the list. If `ref_h->next` is NULL, `find_child()` needs to find a new node at the end of the list. If no new nodes are found, `find_child()` returns an existing sibling node (if any). The head of the list is returned if `ref_h` was NULL; otherwise, `ref_h->next` is returned.

If a new node is discovered by the bus layer, it needs to use the `drm_alloc_node()` service routine to allocate a new DRM node. This new node is returned in the parameter `child_h`. DRM checks the state of the node to determine if it is a new node. All new nodes have their states set to DRM_VIRGIN. This is a transitory state that is used by the DRM to indicate a known but uninitialized node.

When the find_child() discovers a new node, it needs to minimally initialize the new node so that further processing on the node is performed. The DRM node fields that need to be initialized are: device_id, vendor_id, and node_type (DRM_BUS or DRM_DEVICE, DRM_AUTO or DRM_STATIC). The bus layer- specific prop field is populated with data that provides geographic information and other properties of the node. As prop is of type (void *), a pointer to a bus layer-specific data structure is maintained in this field. In the case of a DRM_BUS device, the secondary bus ID needs to be determined and set up in the sbuslayer_id field. The pbuslayer_id field is inherited from the parent's sbuslayer_id field.

DRM_AUTO represents a node that was created during runtime with a call to the drm_alloc_node() service routine. DRM_STATIC represents a node that was configured into the Kernel at build time. Only nodes that are of type DRM_AUTO are deleted. If needed, the bus layer maintains two lists, one that is static and one that automatically implements devices that are removed and inserted.

map_resource ()

This entry point is invoked when the driver calls the drm_map_resource() service routine. The bus layer assigns resource IDs to device resources. Some of these resources are mapped in the Kernel address space. This entry point creates an address translation of the device resource. The bus layer must take care of multilevel translations as necessary. For example, an address associated with a VME device behind a VME2PCI bridge needs several translations to convert to a Kernel virtual address. The VME address gets translated to a PCI bus address. The PCI bus address gets translated to a CPU physical address. The CPU physical address gets translated into a Kernel virtual address. The map_resource() functions takes the following arguments:

```
int map_resource(drm_node_handle node_h, int resource_id,
    unsigned int resource_attr, unsigned int *vaddr);
```

The virtual address needs to be returned to *vaddr when it is invoked successfully. This function generates a failure if the resource ID is invalid or not mappable, or if the resource is already mapped.

unmap_resource ()

This entry point is called when the driver invokes the `drm_unmap_resource()` service routine. The bus layer needs to clear address translations set up for the indicated resource. The node handle and resource ID are passed in as input arguments to the call:

```
int unmap_resource(drm_node_s *node, int resource_id);
```

read_device ()

This entry point is called when the driver invokes the `drm_device_write()` service routine. This call is used to perform device I/O to defined device resources. The resources are identified by the resource ID. The call takes the arguments as shown:

```
int read_device(drm_node_handle node_h, int resource_id,
    int offset, int size, void *buffer);
```

where `node` is the DRM node representing the device resource ID. It identifies the resource being read. The `offset` and `size` parameters are specific to the bus layers. If the resource is seekable, then `offset` and `size` are used as seek offset and read size. If a set of device registers are being read, the `offset` indicates the register number. If the data is accessed in different widths, the `size` parameter is used to indicate the width. The result of the read operation is placed in the buffer. The bus layer implements platform-specific synchronization instructions or I/O flush instructions as needed. Mapped resources are allowed I/O access via this interface. Bus layer resources associated with the device are allowed access via this interface. For example, the PCI bus number, the PCI device number, and the PCI function number are read using this interface.

write_device ()

This entry point is similar to the `read_device()` entry point and allows the device to be written by the driver. The DRM makes the call to this entry point in the following manner:

```
int write_device(drm_node_handle node_h, int resource_id,
    int offset, int size, void *buffer);
```

translate_addr ()

This entry point is called in response to a `drm_translate_addr()` service routine to translate an address. This function has not yet been finalized; in the bus-specific implementation, it just returns `DRM_ENOTSUP`.

finalize_buslayer ()

This entry point is called by the DRM after all the resources of the bus layers have been located. The resources are then initialized and host bridge ports are determined and configured here. `finalize_buslayer()` takes no parameters. This entry point is invoked when the `drm_init()` service routine is called.

flush_buslayer ()

This entry point is called when the driver invokes the `drm_flush_resource()` service routine. This call is used to tell the buslayer to complete all suspended/pending operations (i.e., flush queues, etc.) associated with the resource. The buslayer then complete all writes of data that may be temporarily stored in the resource output queues/FIFOs or destroy any data that may be remaining in the resource input queues/FIFOs. `flush_buslayer()` takes no parameters.

del_busnode ()

This entry point is called prior to a bus node being removed from the DRM tree. The bus layer must release any pending resources associated with this node. For example, any data storage for this node subject to `sysbrk`needs to be freed. The DRM invokes this entry point with the node handle as a parameter:

```
int del_busnode(drm_node_s *handle);
```

This entry point is called when the `drm_delete_node()` service routine is invoked. After this operation the bus node is made visible by a `drm_locate()` or a `drm_insertnode()` service routine.

del_devnode ()

This entry point is called prior to a device node being removed from the DRM tree. The bus layer must release all pending resources allocated to this node. Bus layer-

specific storage attached to the `node_h->prop` field needs to be freed. The `del_devnode()` functions takes the following arguments:

```
int pcidel_devnode(drm_node_handle node_h
```

insertnode ()

This entry point is called in response to a `drm_insertnode()` service routine. The bus layer must verify the existence of the device being inserted, check if it is a duplicate, and return a reference node so that the DRM inserts the node into the tree. The DRM calls this entry point in the following manner:

```
int insertnode(struct drm_node_s *parent, void *prop,
    struct drm_node_s **new_h,
    struct drm_node_s **ref_h);
```

The bus layer creates a new node based on the properties given and returns it in `*new_h`. The bus layer indicates where this node is to be inserted by initializing `*ref_h` with the node handle to insert `*new_h`. If `*ref_h` is NULL, the DRM inserts the node at the head of the list. Similar to `find_child()`, the node should be minimally initialized so that it participates in the DRM service routine calls. If the node being inserted is a `DRM_BUS` type node, the `sbuslayer_id` field should be initialized. DRM invokes `busnode_init` on the bus node inserted by this call. This call adds only one node to the DRM tree.

PCI Bus Layer

LynxOS-178 provides a PCI bus layer that interfaces with DRM. The PCI bus layer supports PCI-to-PCI bridges and CompactPCI Hot Swap environments. The PCI bus layer features a plug-in resource allocator technology to customize PCI resource allocation policies. LynxOS-178 supplies a default PCI resource allocator suitable for most applications.

Services

Device drivers use the standard `DRM API` to interface with the PCI bus layer. The PCI bus layer-specific definitions are in the `machine/pci_resource.h` include file.

NOTE: All PCI devices are disabled by default. The device driver enables the device by writing to the command register in the `install()` function of the driver. Failure to do this results in the device not responding to `IO/MEM/BUSMASTER` accesses and appear to be nonfunctional.

Resources

The PCI bus layer resources are defined in the header file `pci_resource.h`. These resources provide the following functionality:

- Access to the PCI registers
- Mapping and unmapping to the base address registers
- Access to the bus number, device number, and function number of the device

The resource `PCI_RESID_REGS` allows access to all the PCI registers. The `offset` parameter to the `drm_device_read()` and `drm_device_write()` service routines indicates the register to be used. The `size` parameter uses 1, 2, or 4, indicating a byte access, short access, or a word access. A `size` of zero implicitly performs a word access. The table below describes the PCI resources and the corresponding DRM service routines in which they are used.

Table 10-4: PCI Resources

PCI Resource	Description	DRM Service Routine
`PCI_RESID_REGS`	Read/write any PCI config space register. The `offset parameter` in the `drm_device_read/write()` service routines is the register number.	`drm_device_read()` `drm_device_write()`
`PCI_RESID_BUSNO`	The bus number on which the device resides	`drm_device_read()`
`PCI_RESID_DEVNO`	The geographic device number	`drm_device_read()`

Table 10-4: PCI Resources (Continued)

PCI Resource	**Description**	**DRM Service Routine**
PCI_RESID_FUNCNO	The function number with in the device	drm_device_read()
PCI_RESID_DEVID	The PCI device ID	drm_device_read()
PCI_RESID_VENDORID	The PCI vendor ID	drm_device_read()
PCI_RESID_CMDREG	The PCI command/status register	drm_device_read() drm_device_write()
PCI_RESID_REVID	The PCI revision ID	drm_device_read()
PCI_RESID_STAT	Alias for the PCI command status register	drm_device_read() drm_device_write()
PCI_RESID_CLASSCODE	The PCI class code register	drm_device_read()
PCI_RESID_SUBSYSID	The PCI subsystem ID	drm_device_read()
PCI_RESID_SUBSYSVID	The PCI subsystem vendor ID	drm_device_read()
PCI_RESID_PCICAP	Bitset for the supported capabilities The following constants and macro are defined: PCI_CAP_MSI - MSI Capability bit, PCI_CAP_MSIX - MSI-X Capability bit, PCI_MSI_CAPABLE(val) - true if any bit of MSI and MSI-X capabilities is set. If the device has both MSI and MSI-X capabilities, only the MSI-X capability is reported.	drm_device_read()
PCI_RESID_MSG_REQ	The number of messages requested. The device should have MSI/MSI-X capability.	drm_device_read()
PCI_RESID_MSG_ASSIGN	The number of messages allocated. The device should have MSI/MSI-X capability.	drm_device_read()
PCI_RESID_BAR0	Base address register 0	drm_map_resource() drm_unmap_resource()
PCI_RESID_BAR1	Base address register 1	drm_map_resource() drm_unmap_resource()
PCI_RESID_BAR2	Base address register 2	drm_map_resource() drm_unmap_resource()

PCI_RESID_BAR3	Base address register 3	drm_map_resource() drm_unmap_resource()
PCI_RESID_BAR4	Base address register 4	drm_map_resource() drm_unmap_resource()
PCI_RESID_BAR5	Base address register 5	drm_map_resource() drm_unmap_resource()

Plug-in Resource Allocator

The PCI bus layer uses a set of function pointers stored in a `struct pcibus_alloc_handle_s` to allow a customized resource allocator to be installed. This structure is defined in `drm/pci_conf.h`as follows:

```
struct pcibus_alloc_handle_s {
          int (*pcialloc_init)();
          int (*pcialloc_busno)();
```

```
          int (*pcialloc_bus_res)()
          int (*pcialloc_dev_res)();
#ifdef HOT_SWAP
          int (*pcifree_busno)();
          int (*pcifree_bus_res)();
          int (*pcifree_dev_res)();
#endif /* HOT_SWAP */
};
```

It is also useful to know the `pci_node` structure maintained by the PCI bus layer for every PCI device. This structure is as follows:

```
struct pci_node_s {
/* Common PCI node related data */
unsigned int hostbridge_no;/* the bus number for the host PCI bus
(i.e., PCIA, PCIB, etc) */
unsigned int bus_no; unsigned int dev_no; unsigned int func_no;
struct pci_resource_s resource[PCI_NUM_BAR];

/* BusNode related data */

   unsigned int sec_bus_no; /* Used by BusNodes */ unsigned int sub_bus_no;
   /* Used by BusNodes */ struct pci_resource_s
   *res_heads[PCI_NUM_RES_TYPES];
   /* Used by Busnodes */

   /* The following are for backward compatibility */
   #define res_io_headres_heads[PCI_IO_RES_TYPE]
   #define res_mem_headres_heads[PCI_MEM_RES_TYPE]
   #define res_mem64_head res_heads[PCI_MEM64_RES_TYPE]
   #define res_mempf_head res_heads[PCI_MEMPF_RES_TYPE]
   /* New resource allocation algorithm
   */
   int io_nodes_left; int mem_nodes_left;
};
```

The `pci_resource` structure referenced above is:

```
struct pci_resource_s {
   struct pci_resource_s *next;
   struct drm_node_s *node;           /* node this resource belongs to */
   unsigned int bar;                   * base register */
   unsigned int alignment;
   unsigned int size;
   pci_res_type_t type;
   unsigned int baddr;
     /* The physical address associated with the mapping */
   unsigned int paddr[MAX_RESOURCE_MAPPINGS];
     /* The virtual address associated with the mapping */
   unsigned int vaddr[MAX_RESOURCE_MAPPINGS];
     /* The port associated with the mapping */
   unsigned int port[MAX_RESOURCE_MAPPINGS];
     /* PCI resource attributes */
   unsigned int attr[MAX_RESOURCE_MAPPINGS];
};
```

pcialloc_init()

This entry point is invoked during bus layer initialization. Any global data structures that are used by the allocator should be created and set up in this routine. The number of host bridges is passed in as input arguments to the call:

```
int pcialloc_init(unsigned int num_host_bridges)
```

pcialloc_busno()

This entry point is invoked to allocate the secondary and subordinate bus numbers for a PCI-to-PCI bridge node. The array index `PCI_IO_RES_TYPE` is assigned to hold the PCI I/O resources. The index `PCI_MEM_RES_TYPE` is assigned to PCI memory resource. The index `PCI_MEM64_RES_TYPE` is assigned to PCI memory64, and the index `PCI_MEMPF_RES_TYPE` is assigned to PCI prefetch memory. These allocations are linked using the next field to the parent resource list. The `pci_node_s` data structure also maintains the resource list heads for the children of this bridge. This function needs to populate the resource array suitably and clear the list head pointers.

pcialloc_bus_res()

This entry is used to allocate address space resources for the passed in PCI-to-PCI bridge node. The resources assigned to this bridge are stored in the resource array. Array index 0 is assigned to hold the PCI I/O resources. Index-1 is assigned to PCI memory resource. Index 2 is assigned to PCI 64-bit memory, and Index 3 is assigned to PCI prefetch memory. These allocations are linked using the `next` field to the parent resource list. The `pci_node_s` data structure also maintains the resource list heads for the children of this bridge. This function needs to populate the resource array suitably and clear the list head pointers.

pcialloc_dev_res()

This entry point is used to allocate device-specific address space. This function gets input as parameters the DRM node handle and a base address register (BAR) index. The type, size, and alignment requirements are filled in by the PCI bus layer. The allocator needs to fill in the `baddr` (bus address), the `paddr` (CPU physical address), the `attr` (PCI resource attributes), and the `port` (port associated with the mapping) fields of the resource. Note that the `vaddr` (virtual address) is allocated when the resource is mapped in. Depending on the policy implemented by the allocator, the parent node's resource list-head and allocations are used to assign resource to this node. All the resources are chained to the parent's resource list-heads, if needed, by using the `next` field in the resource structure.

pcifree_busno()

This entry point is called in response to a `drm_free_resource()` service routine request. The allocator should reclaim the `busno` and give it back to the free pool. This function is passed the `DRM` node as input.

pcifree_bus_res()

This entry point is invoked in response to a `drm_free_resource()` service routine request on a PCI2PCI bridge node. The allocator should reclaim the bus address space and give it back to the free pool. This function should report an error if any of the children still have resources allocated.

pcifree_dev_res()

This entry point is used to reverse the allocation performed by the `pcialloc_dev_res()` entry point. The node handle and the BAR index are passed in as parameters.

Example Driver

```
/* This is a sample driver for a hypothetical PCI device. This PCI device has a
vendor_id of ABC_VENDORID and a device_id ABC_DEVICEID. This device has one base
address register implemented as a PCI Memory BAR and needs 4K of space. The
device registers are implemented in this space. The device needs a interrupt
service routine to handle events raised by the device. It may be possible that
there are multiple of these devices in the system. */


#include <pci_resource.h>

#define PCI_IO_ENABLE 0x1
#define PCI_MEM_ENABLE 0x2
#define PCI_BUSMASTER_ENABLE 0x4

struct device_registers {
   unsigned int register1;
   unsigned int register2;
   unsigned int register3;
   unsigned int register4;
};


struct device_static {
   struct drm_node_s *handle;
   struct device_register *regptr;
   int bus_number;
   int device_number;
   int func_number;
};

int abc_install(struct info_t *info)
{

   struct device_static *static_ptr;
   int rv = 0;
   unsigned int val;

   /* Allocate device static block */

   static_ptr = (struct device_static *)
            sysbrk(sizeof(struct device_static));


   if(!static_ptr)
   {
       /* memory allocation failed !! */
       goto error_0;
   }

   /* Find the device ABC_VENDORID, ABC_DEVICEID. Every call to abc_install()
by the OS, installs a ABC device. The number of times abc_install() is called
depends on how many static devices for ABC have been configured via the standard
LynxOS device configuration facilities. This entry point is also called during
a dynamic device install. */
```

```
/* A Hot Swap capable driver may replace the next call with drm_claim_handle()
and pass the handle given by the system management layer, instead of finding the
device by itself */

#if !defined(HOTSWAP)

    rv = drm_get_handle(PCI_BUSLAYER,
    ABC_VENDORID,ABC_DEVICEID,
    &(static_ptr->handle));

    if(rv)
    {
    /* drm_get_handle or drm_claim_handle failed to find a
         device. return failure to the OS saying install failed. */

         debug(("failed to find device(%x,%x)\n",
         ABC_VENDORID,ABC_DEVICEID));
         goto error_1;
    }

    /* Register an interrupt service routine for this
         device */

    rv = drm_register_isr(static_ptr->handle,
    abc_isr, NULL);

    if(rv == SYSERR)
    {

    /*If register isr fails release the handle and exit*/

         debug(("drm_register_isr failed %d\n",rv));
         goto error_2;
    }

    /* Map in the memory base address register (BAR) */

    rv = drm_map_resource(static_ptr->handle,
    PCI_RESID_BAR0,
    &(static_ptr->regptr));

    if(rv)
    {

    /*drm_map_resource failed , release the device and
    exit*/

         debug(("drm_map_resource failed with %d\n",rv));

         goto error_3;

    }

    /* Enable the device for memory access */

    rv = drm_device_read(static_ptr->handle,
    PCI_RESID_CMDREG,0,0,&val);


    if(rv)
    {
         debug(("drm_device_read failed with %d\n",rv));
```

```
    goto error_4;
}

val |= PCI_MEM_ENABLE ;

rv = drm_device_write(static_ptr->handle,
        PCI_RESID_CMDREG,0,0,&val);

if(rv)
{
    debug(("drm_device_write failed to update the
        command register, error = %d\n",rv);
    goto error_4;
}



/* Read the Geographic properties of the device, this
    is used by the driver to uniquely identify the
    device */

rv = drm_device_read(static_ptr->handle,
        PCI_RESID_BUSNO,0,0,
        &(static_ptr->bus_number));

if(rv)
{
    debug(("drm_device_read failed to read bus
        number %d\n",rv));
    goto erro_4;
}

rv = drm_device_read(static_ptr->handle,
        PCI_RESID_DEVNO,0,0,
        &(static_ptr->device_number));

if(rv)
{
    debug(("drm_device_read failed to read device
        number %d\n",rv));
    goto error_4;
}

rv = drm_device_read(static_ptr->handle,
        PCI_RESID_FUNCNO,0,0,
        &(static_ptr->func_number));

if(rv)
{

    debug(("drm_device_read failed to read function
        number %d\n",rv));
    goto error_4 ;
}


/* perform any device specific initializations here,
    the following statements are just illustrative */

/* recoset() is used to catch any bus errors */

if(!recoset())
```

```
    {

        static_ptr->regptr.register1 = 0;
        static_ptr->regptr.register2 = 9600;
        static_ptr->regptr.register3 = 1024;

        if(static_ptr->regptr.register4 == 0x4)
        {
            static_ptr->regptr.register3 = 4096;
        }
    } else {
        /* caught a bus error */
        goto error_4;
    }
    noreco(); /* ……………… and so on */

    /* Successful exit from the install routine, return
        the static pointer */

    return(static_ptr);

error_4:

    drm_unmap_resource(static_ptr->handle,
            PCI_RESID_BAR0);

error_3:

    drm_unregister_isr(static_ptr->handle);

error_2:

    drm_free_handle(static_ptr->handle);

error_1:

    sysfree(static_ptr, sizeof(struct device_static));

error_0:

    return(SYSERR);

} /* abc_install */

int abc_uninstall(struct device_static *static_ptr)
{

    unsigned int val;
    int rv = 0;

    /* perform any device specific shutdowns */

    static_ptr->regptr.register1 = 0xff ;

    /* and so on */


    /* Disable the device from responding to memory access */

    rv = drm_device_read(static_ptr->handle,
            PCI_RESID_CMDREG,0,0,&val);
    if(rv)
```

```
    {
        debug(("failed to read device %d\n",rv));
    }
    val &= ~(PCI_MEM_ENABLE);
    rv = drm_device_write(static_ptr->handle,
            PCI_RESID_CMDREG,0,0,&val);
    if(rv)
    {
        debug(("failed to write device %d\n",rv));
    }


    /* Unmap the memory resource */

    rv = drm_unmap_resource(static_ptr->handle,
            PCI_RESID_BAR0);
    if(rv)
    {
        debug(("failed to unmap resource %d\n",rv));
    }

    /* unregister the isr */

    rv = drm_unregister_isr(static_ptr->handle);
    if(rv)
    {
        debug(("failed to unregister isr %d\n",rv));
    }
    /* release the device handle */

    rv = drm_free_handle(static_ptr->handle);
    if(rv)
    {
        debug(("Failed to free the device handle %d\n",
            rv));
    }

    sysfree(static_ptr,sizeof(struct device_static));

    return(0);
}

/* The other entry points of the driver are device specific */

int abc_open(…) {
}

int abc_read(…) {
}

int abc_write(…) {
}

int abc_ioctl(…) {
}

int abc_close(…) {
}

int abc_isr(...) {
}
```

APPENDIX A Network Device Drivers

In LynxOS-178, two network stacks are supplied: LCS and BSD-derived. This appendix describes the BSD-derived network stack and network devices for this stack.

The LynxOS-178 BSD-derived TCP/IP module is provided to facilitate software development. It is not partition-aware and, therefore, can run only within VM0. TCP/IP networking using the BSD-derived network stack is available only in the Development Environment.

A network driver is defined as a link-level device driver that interfaces with the LynxOS-178 TCP/IP module. Unlike other drivers, a network driver does not interface directly with user applications. It interfaces instead with the LynxOS-178 TCP/IP module. This interface is defined by a set of driver entry points and data structures described in the following sections.

Kernel threads play an important role in LynxOS-178 networking software, not only within the drivers, but also as part of the TCP/IP module.

The example code fragments given below illustrate the points discussed for an Ethernet device. These examples can easily be adapted to other technologies.

Kernel Data Structures

A network driver must make use of a number of Kernel data structures. Each of these structures is briefly described here and their use is further illustrated throughout the rest of the chapter.

A driver must include the following header files which define these structures and various symbolic constants that are used in the rest of this chapter:

```
#include <types.h>
#include <io.h>
#include <ioctl.h>
#include <socket.h>
#include <bsd/in.h>
#include <bsd/if.h>
#include <bsd/if_ether.h>
#include <bsd/in_var.h
#include <bsd/bsd_mbuf.h>
#include <bsd/netisr.h>
```

struct ether_header

The Ethernet header must be prefixed to every outgoing packet. It specifies the destination and source Ethernet addresses and a packet type. The symbolic constants `ETHERTYPE_IP`, `ETHERTYPE_ARP`, and `ETHERTYPE_RARP` can be used for the packet type.

```
struct ether_header {
   u_char  ether_dhost[6];  /*  dest  Ethernet  addr  */
   u_char ether_shost[6]; /* source Ethernet addr */
   u_short ether_type;     /* Ethernet packet type */
}
```

struct arpcom

The `arpcom` structure is used for communication between the TCP/IP module and the network interface driver. It contains the `ifnet` structure (described below) and the interface's Ethernet and Internet addresses. This structure must be the first element in the statics structure.

```
struct arpcom {
       struct ifnet ac_if;/* network visible interface */
       u_char ac_enaddr[6];  /* Ethernet address */ struct
       in_addr  ac_ipaddr;  /*  Internet  address  */  struct
       ether_multi *ac_multiaddrs;
                      /* list of ether multicast addrs */
       int ac_multicnt;/* length of ac_multiaddrs list */
};
```

struct sockaddr

The `sockaddr` structure is a generic structure for specifying socket addresses, containing an address family field and up to 14 bytes of protocol-specific address data.

```
struct sockaddr {
    u_char sa_len;               /* total length */
    u_char  sa_family;           /* address family */
    char sa_data[14];            /* longer; addr value */
};
```

struct sockaddr_in

The `sockaddr_in` structure is a structure used for specifying socket addresses for the Internet protocol family.

```
struct sockaddr_in { u_char sin_len;
       u_char sin_family;                 /* always AF_INET */
       u_short sin_port;                     /* port number */
       struct in_addr sin_addr; /* host Internet addr */
```

```
        char sin_zero[8];
};
```

struct in_addr

The `in_addr` structure is a structure specifying a 32-bit host Internet address.

```
struct in_addr {
    u_long s_addr;
};
```

Struct ifnet

This is the principle data structure used to communicate between the driver and the TCP/IP module. It provides the TCP/IP software with a generic, hardware-independent interface to the network drivers. It specifies a number of entry points that the TCP/IP module can call in the driver, a flag variable indicating general characteristics and the current state of the interface, a queue for outgoing packets, and a number of statistics counters.

```
struct ifnet {
   char *if_name;                  /* name, e.g. "wd" or "oblan" */
   char *p;                                /* user defined field */
   struct ifnet *if_next;          /* all struct ifnets are chained */
   struct ifaddr *if_addrlist;     /* linked list of addresses */
   int if_pcount;                  /* number of promiscuous listeners */
   caddr_t if_bpf;                 /* packet filter structure */
   u_short if_index;               /* numeric abbreviation for if */
   short if_unit;                  /* sub-unit for lower level driver */
   short if_timer; /* time 'til if_-watchdog called */
   short if_flags;                  /* up/down, broadcast, etc. */

struct if_data {
/* generic interface information */
   u_char ifi_type;                 /* Ethernet, token ring etc */
   u_char ifi_addrlen;                     /* media header length */
   u_long ifi_mtu;                 /* maximum transmission unit */
   u_long ifi_metric;              /* routing metric (external) */
   u_long ifi_baudrate;                             /* line speed */

/* volatile statistics */
   u_long ifi_ipackets;            /* packets received on i/f */
   u_long ifi_ierrors;             /* input errors on i/f */
   u_long ifi_opackets;            /* packets sent on i/f */
   u_long ifi_oerrors;             /* output errors on i/f */
   u_long ifi_collisions;          /* collisions on csma i/f */
   u_long ifi_ibytes;              /* total number of bytes received */
   u_long ifi_obytes;              /* total number of octets sent */
   u_long ifi_imcasts;             /* packets received via multi-cast */
   u_long  ifi_omcasts;            /* packets sent via multi-cast */
   u_long  ifi_iqdrops;       /* dropped on input, this interface */
   u_long  ifi_noproto;       /* destined for unsupported protocol */
   struct  timeval ifi_lastchange;   /* last updated */
} if_data;
```

```
/* procedure handles */
   int (*if_init)();                         /* init routine */
   int (*if_output)();                       /* output routine */
   int (*if_start)();                /* initiate output routine */
   int (*if_done)();                 /* output complete routine */
   int (*if_ioctl)();                        /* ioctl routine */
   int (*if_reset)();                        /* bus reset routine */
   int (*if_watchdog)();                     /* timer routine */
   int (*if_setprio)();              /* prio tracking of kthread */

/* output queue */
   struct ifqueue {
      struct mbuf *ifq_head;
      struct mbuf *ifq_tail;
      int  ifq_len;
      int ifq_maxlen;
      int  ifq_drops;
   } if_snd;
struct raweth *if_raweth;
};
```

The symbolic constants `IFF_UP`, `IFF_RUNNING`, `IFF_BROADCAST`, and `IFF_NOTRAILERS` can be used to set bits in the `if_flags` field.

Looking at the `arpcom` structure, notice that the first member is an `ifnet` structure. A driver should declare a struct `arpcom` as part of the statics structure and use the `ifnet` structure within this. There is an important reason for this, which is explained in the discussion of the `ioctl` entry point.

struct mbuf

Data packets are passed between the TCP/IP module and a network interface driver using mbuf structures. This structure is designed to allow the efficient encapsulation and decapsulation of protocol packets without the need for copying data. A number of functions and macros are defined in `mbuf.h` for using mbuf structures.

```
/* header at beginning of each mbuf: */
   struct m_hdr {
   struct mbuf    *mh_next;     /* next buffer in chain */ struct
   mbuf *mh_nextpkt;                   /* next chain in queue */
   int mh_len;                /* amount of data in this mbuf */
   caddr_t mh_data;                      /* location of data */
   short mh_type;              /* type of data in this mbuf */
   short mh_flags;                    /* flags; see below */
};

/* record/packet header in first mbuf of chain;
* valid if M_PKTHDR set
*/

struct  pkthdr {
     int len;                 /* total packet length */
     struct  ifnet *rcvif;            /* rcv interface */
};
```

```
/* description of external storage mapped into mbuf,
* valid if M_EXT set
*/

struct m_ext {
     caddr_t ext_buf;                    /* start of buffer */
     void (*ext_free)();                 /* free routine */
     u_int   ext_size;       /* size of buffer, for ext_free */
};

struct mbuf {
   struct m_hdr m_hdr;
   union {
     struct {
            struct pkthdr MH_pkthdr; /* M_PKTHDR set */
     union {
            struct m_ext MH_ext;/* M_EXT set */
            char MH_databuf[MHLEN];
     } MH_dat;
   } MH;
   char  M_databuf[MLEN];/* !M_PKTHDR, !M_EXT */
   } M_dat;
};

#define m_next      m_hdr.mh_next
#define m_len       m_hdr.mh_len
#define m_data      m_hdr.mh_data
#define m_type      m_hdr.mh_type
#define m_flags     m_hdr.mh_flags
#define m_nextpkt   m_hdr.mh_nextpkt
#define m_act       m_nextpkt
#define m_pkthdr    M_dat.MH.MH_pkthdr
#define m_ext       M_dat.MH.MH_dat.MH_ext
#define m_pktdat    M_dat.MH.MH_dat.MH_databuf
#define m_dat       M_dat.M_databuf
```

Adding or Removing Data in an mbuf

The position and number of data currently in an `mbuf` are identified by a pointer and a length. By changing these values, data can be added or deleted at the beginning or end of the `mbuf`. A pointer to the start of the data in the `mbuf` can be obtained using the `mtod` macro. The pointer is cast as an arbitrary data type, specified as an argument to the macro:

```
char  *cp;
struct mbuf *mb;

cp = mtod (mb, char *);
/* get pointer to data in mbuf */
```

The macro `dtom` takes a pointer to data placed anywhere within the data portion of the `mbuf` and returns a pointer to the `mbuf` structure itself. For example, if we know that `cp` points within the data area of an `mbuf`, the sequence will be:

```
struct mbuf *mb;
char *cp;
```

```
mb = dtom(cp);
```

Data is added to the head of an mbuf by decrementing the m_data pointer, incrementing the m_len value, and copying the data using a function such as bcopy. Data is added to the tail of an mbuf in a similar manner by incrementing the m_len value. The ability to add data to the tail of an mbuf is useful for implementing trailer protocols, but LynxOS-178 does not currently support such protocols.

Data is removed from the head or tail of an mbuf by simply incrementing the m_data pointer and/or decrementing m_len.

Allocating mbufs

The above examples did not discuss what to do when sufficient space is not available in an mbuf to add data. In this case, a new mbuf can be allocated using the function m_get. The new mbuf is linked onto the existing mbuf chain using its m_next field. m_get can be replaced with MGET, which is a macro rather than a function call. MGET produces faster code, whereas m_get results in smaller code. The example to add data to the beginning of a packet now becomes:

```
struct mbuf *m;
caddr_t src, dst;

MGET(m, M_DONTWAIT, MT_HEADER);
if (m == NULL)
    return (ENOBUFS);
dst = mtod (m, caddr_t);
bcopy (src, dst, n);
```

The second argument to m_get/MGET specifies whether the function should block or return an error when no mbufs are available. A driver should use M_DONTWAIT, which will cause the mbuf pointer to be set to zero if no mbufs are free.

The third argument to m_get/MGET specifies how the mbuf will be used. This is for statistical purposes only and is used, for example, by the command netstat -m. The types used by a network driver are MT_HEADER for protocol headers and MT_DATA for data packets.

mbuf Clusters

When receiving packets from a network interface, a driver must allocate mbufs to store the data. If the data packets are large enough, a structure known as an mbuf cluster can be used. A cluster can hold more data than a regular mbuf, MCLBYTES

bytes as opposed to MLEN. As a rule of thumb, there is benefit to be gained from using a cluster if the packet is larger than MCLBYTES/2.

Freeing mbufs

Since there are a limited number of mbufs in the system, the driver must take care to free mbufs at the appropriate points, listed below:

Packet Output:

- Interface down
- Address family not supported
- No mbufs for Ethernet header
- if_snd queue full
- After packet has been transferred to interface

Packet Input:

- Not enough mbufs to receive packet
- Unknown Ethernet packet type
- Input queue full

The sections on packet input and output show appropriate code examples for each of the above situations. Failure to free them will eventually lead to the system running out of mbufs.

Table A-1: Summary of Commonly Used mbuf Macros

Macro	Description
MCLGET	Get a cluster and set the data pointer of the mbuf to point to the cluster.
MFREE	Free the mbuf. On return, mbuf's successor (pointed to by m->m_next) is stored in the second argument.
MGETHDR	Allocate an mbuf and initialize it as a packet header.
MH_ALIGN	Set the m_data pointer to an mbuf containing a packet header to place an object of the specified size at the end of mbuf, longword aligned.
M_PREPEND	Prepend specified bytes of data in front of the data in the mbuf.

Table A-1: Summary of Commonly Used mbuf Macros

Macro	Description
`dtom`	Convert the data pointer within `mbuf` to `mbuf` pointer.
`mtod`	Convert `mbuf` pointer to data pointer of specified type.

Table A-2: Summary of Commonly Used mbuf Functions

Function	Description
`m_adj`	Remove data from `mbuf` at start/end.
`m_cat`	Concatenate one `mbuf` chain to another.
`m_copy`	Version of `m_copym` that does not wait.
`m_copydata`	Copy data from `mbuf` chain to a buffer.
`m_copyback`	Copy data from buffer to an `mbuf` chain.
`m_copym`	Create a new `mbuf` chain from an existing `mbuf` chain.
`m_devget`	Create a new `mbuf` chain with a packet header from data in a buffer.
`m_free`	A function version of `MFREE` macro.
`m_freem`	Free all `mbufs` in a chain.
`m_get`	A function version of `MGET` macro.
`m_getclr`	Get an `mbuf` and clear the buffer.
`m_gethdr`	A function version of `MGETHDR` macro.
`m_pullup`	Pull up data so that the certain number of bytes of data are stored contiguously in the first `mbuf` in the chain.

Statics Structure

In keeping with the general design philosophy of LynxOS 7 drivers, network drivers should define a statics structure for all device-specific information. However, the TCP/IP software has no knowledge of this structure, which is specific to each network interface, and does not pass it as an argument to the driver entry points. The solution to this problem is for the `ifnet` structure to be contained within the statics structure. The user-defined field `p` in the `ifnet` structure is initialized to contain the address of the statics structure.

Given the address of the `ifnet` structure passed to the entry point from the TCP/IP software, the driver can obtain the address of the statics structure as follows:

```
struct ifnet *ifp;
struct statics *s = (struct statics *) ifp->p;
```

The `arpcom` structure *must* be the first element in the statics structure. In the code examples below, the `arpcom` structure is named `ac`.

Packet Queues

A number of queues are used for transferring data between the interface and the TCP/IP software. There is an output queue for each interface, contained in the `ifnet` structure. There are two input queues used by all network interfaces - one for IP packets and another for ARP/RARP packets. All queues are accessed using the macros `IF_ENQUEUE`, `IF_DEQUEUE`, `IF_QFULL`, and `IF_DROP`.

Driver Entry Points

A network driver contains the following entry points:

`install/uninstall` Called by the Kernel in the usual manner. The `install` routine must perform a number of tasks specific to network drivers.

`interrupt handler` The interrupt handler is declared and called in exactly the same manner as for other drivers. It will not be discussed further here.

`output` Called by TCP/IP software to transmit packets on the network interface.

`ioctl` Called by TCP/IP software to perform a number of commands on the network interface.

`watchdog` Called by the TCP/IP software after a user-specified timeout period.

`reset` Called by the Kernel during the reboot sequence.

`setprio` Called by the TCP/IP software to implement priority tracking.

By convention, the entry point names are prefixed with the driver name.

install Entry Point

In addition to the usual things done in the `install` routine (allocation and initialization of the statics structure, declaration of interrupt handler, and so on), the driver must also fill in the fields of the `ifnet` structure and make the interface known to the TCP/IP software. Note also that hardware initialization is normally done in the `ioctl` routine rather than in the `install` routine.

Finding the Interface Name

The `install` routine must initialize the `if_name` field in the `ifnet` structure. This is the name by which the interface is known to the TCP/IP software. It is used, for example, as an argument to the `ifconfig` and `netstat` utilities.

The interface name is a user-defined field specified in the driver's device configuration file (`drvr.cfg`) in the `/sys/cfg` directory. The usual technique used by the driver to find this field is to search the `ucdevsw` table for an entry with a matching device information structure address. `ucdevsw` is a Kernel table containing entries for all the character devices declared in the `config.tbl` file. The Kernel variable `nucdevsw` gives the size of this table.

```
extern int nucdevsw;
extern struct udevsw_entry ucdevsw[];
struct statics *s;
struct ifnet *ifp;

ifp->if_name = (char *) 0;
for (i = 0; i < nucdevsw; i++) {
    if (ucdevsw[i].info == (char *) info) {
        if (strlen (ucdevsw[i].name) > IFNAMSIZ) {
            sysfree (s, (long) sizeof (struct statics));
            return ((char *) SYSERR);
        }
        ifp->if_name = ucdevsw[i].name;
        break;
    }
}
if (ifp->if_name == (char*) 0) {
    sysfree (s, (long) sizeof (struct statics));
    return ((char *) SYSERR);
}
```

Note that this method only works for statically installed drivers. Dynamically installed drivers do not have an entry in the ucdevsw table.

Initializing the TCP/IP Software

A network driver's install routine must also test whether the TCP/IP software has been initialized. If it has not, it must call the bsd_sysinit function:

```
extern int bsdinit_tobedone;
if (bsdinit_tobedone)
   bsd_sysinit();
```

This must be done *before* calling the if_attach function.

Initializing the Ethernet Address

The ac_enaddr field in the arpcom structure is used to hold the interface's Ethernet address, which must be included in the Ethernet header added to all outgoing packets. The install routine should initialize this field by reading the Ethernet address from the hardware.

```
struct statics *s;

get_ether_addr (&s->ac.ac_enaddr);
```

In the above example, get_ether_addr would be a user-written function that reads the Ethernet address from the hardware.

Initializing the ifnet Structure

The various fields in the ifnet structure should be initialized to appropriate values. Unused fields should be set to zero or NULL. Once the structure has been initialized, the network interface is made known to the TCP/IP module by calling the if_attach function.

```
struct statics *s;
struct ifnet *ifp;

ifp->if_timer = 0 ;
ifp->p = (char *) s;/* address of statics structure */
ifp->if_unit = 0;
ifp->if_mtu = ETHERMTU;
ifp->if_flags = IFF_BROADCAST | IFF_NOTRAILERS;
ifp->if_init = NULL;
ifp->if_output = ether_output;
ifp->if_ioctl = drvr_ioctl;
ifp->if_reset = drvr_reset;
ifp->if_start = drvr_start;
ifp->if_setprio = drvr_setprio;
ifp->if_watchdog = drvr_watchdog;
```

```
if_attach (ifp);
```

Note that the if_output handle in the ifnet structure should point to the ether_output routine in the TCP/IP module. Previously it pointed to the driver-specific local routine. In BSD 4.4, most of the hardware-independent output code has been moved to the ether_output routine. After the ether_output routine has packaged the data for output, it calls a start routine specified by if_start, a member of the interface ifnet structure. For example:

```
ifp->if_start = lanstart;
```

Packet Output

The processing of outgoing packets is divided into two parts. The first part concerns the TCP/IP module which is responsible for queueing the packet on the interface's output queue. The actual transmission of packets to the hardware is handled by the driver start routine and the Kernel thread. The driver is responsible in all cases for freeing the mbufs holding the data once the packet has been transmitted or at any time an error is encountered.

ether_output Function

A number of tasks previously performed by the driver output routine are now done in TCP/IP module by the ether_output routine. Thus, the if_output field of the interface ifnet structure is initialized to the address of the ether_output routine in the driver install routine:

```
ifp->if_output = ether_output;
```

This causes the ether_output routine to be called indirectly when the TCP/IP module has a packet to transmit. After enqueuing the packet to transmit, ether_output calls a device-specific function indirectly through the if_start pointer in the ifnet structure. For example, if ifp points to an ifnet structure,

```
(*ifp->if_start)(ifp),
```

the if_start field is also initialized by the driver install routine. The driver start routine starts output on the interface if resources are available. Before removing packets from the output queue for transmission, the code will normally have to test whether the transmitter is idle and ready to accept a new packet. It will typically dequeue a packet (which is enqueued by `ether_output`) and transmit it.

```
        sdisable(s);
        if (ds->xmt_pending) {
    /* if already one transmission is in progress */
            srestore(s);
            return 1;
        }
        IF_DEQUEUE(&ifp->if_snd, m);
        if (m == 0) {
            srestore(s);
            return 0;
        }
        ds->xmt_pending = 1;
        srestore(s);

        ...
        ...
        /* Initiate transmission */
        return 0;
```

Note that manipulation of the output queue must be synchronized with `sdisable`/`srestore`. Once the packet has been transferred, the driver must free the `mbuf` chain using `m_freem`.

One important point to consider here is that the start routine can now be called by the TCP/IP module (via `ether_output`) and the driver Kernel thread upon receiving an interrupt. Thus, the start routine must protect code and data in the critical area. For example, it could check a “pending” flag, which is set before starting to transmit and cleared when a transmit done interrupt is received. If the transmit start routine is not reentrant, it could signal a semaphore in order to notify the driver’s Kernel thread that packets are now available on the output queue. The routine should then return 0 to indicate success.

```
ssignal(&s->thread_sem);
return (0);
```

Also note that the total data available in an `mbuf` can be obtained from the `mbuf` packet header. This is a new feature in BSD 4.4 and above. Previously, one had to traverse the `mbuf` chain to calculate it.

```
/* put mbuf data into TFD data area */
length = m->m_pkthdr.len;
m_copydata(m, 0, length, p);
m_freem(m);
```

Kernel Thread Processing

The Kernel thread must perform the following activities relating to packet output:

- Start transmission
- Maintain statistics counters

Starting Transmission

As explained above, the Kernel thread also calls the driver start routine to start transmission. The transmit start routine dequeues a packet from the interface send queue and transmits it.

Statistics Counters

The counters relating to packet output are the `if_opackets`, `if_oerrors`, and `if_collisions` fields in the `ifnet` structure. The `if_opackets` counter should be incremented for each packet that is successfully transmitted by the interface without error. If the interface indicates that an error occurred during transmission, the `if_oerrors` counter should be incremented. The driver should also interrogate the interface to determine how many collisions, if any, occurred during transmission of a packet. A collision is not necessarily an error condition. An interface will normally make a number of attempts to send a packet before raising an error condition.

Packet Input

When packets are received by a network interface, they must be copied from the device to `mbufs` and passed to the TCP/IP software. Because this can take a significant amount of time, the bulk of the processing of incoming packets should be done by the driver's Kernel thread so that it does not impact the system's real-time performance. The interrupt handler routine should do the minimum necessary to ensure that the interface continues to function correctly.

To maintain bounded system response times, the interrupt handler should also disable further interrupts from the interface. These will be reenabled by the driver's Kernel thread. The processing of input packets involves the following activities:

- Determine packet type.

- Copy data from interface into `mbufs`.
- Strip off Ethernet header.
- Enqueue packet on input queue.
- Reenable receiver interrupts.
- Maintain statistics counters.

Determining Packet Type

The packet type is specified in the Ethernet header and is used by the driver to determine where to send the received packet. In the following code fragment, the `ptr` variable is assumed to be a pointer to the start of the received Ethernet frame. The use of the `ntohs` function ensures the portability of code across different CPU architectures.

```
ether_type  =  ntohs  (((struct
        ether_header *)ptr)->ether_type);
```

Copying Data to mbufs

Most network devices have local RAM that is visible to the device driver. On packet reception, the driver must allocate sufficient `mbufs` to hold the received packet, copy the data to the `mbufs`, and then pass the `mbuf` chain to the TCP/IP software. The `ifnet` structure is added to the start of the packet so that the upper layers can easily identify the originating interface. The Ethernet header must be stripped from the received packet. This can be achieved simply by not copying it into the `mbuf`(s). If the entire packet cannot be copied, any allocated `mbufs` must be freed. The following code outlines how a packet is copied from the hardware to `mbufs` using the `m_devget` routine. `m_devget` is called with the address and the size of the buffer that contains the received packet. It creates a new `mbuf` chain and returns the pointer to the chain.

```
m = m_devget(buf, len, 0, ifp, 0);
```

`ifp` is the device interface pointer. The variable `buf` points to the received data. This is usually an address in the interface's local RAM.

By default, `m_devget` uses `bcopy`, which copies data one byte at a time. A driver can provide a different algorithm for more efficiency and pass its address to the `m_devget` routine.

Enqueueing Packet

The packet read routine finally calls a TCP/IP module called `ether_input` to enqueue the received packet on one of the TCP/IP software's input queues and for further processing.

```
struct ifnet *ifnet;
struct ether_header *et;
struct mbuf *m;

ether_input(ifp, et, m);
```

Statistics Counters

The counters relating to packet input are the `if_ipackets` and `if_ierrors` fields in the `ifnet` structure. The `if_ipackets` counter should be incremented for each packet that is successfully transferred from the interface and enqueued on the TCP/IP input queue. Receive errors are normally indicated by the interface in a status register. In this case, the `if_ierrors` counter should be incremented.

ioctl Entry Point

The `ioctl` entry point is called by the TCP/IP software with the following syntax:

```
drvr_ioctl (ifp, cmd, arg)
struct ifnet *ifp;
u_long cmd;                          /* ioctl command id */
caddr_t arg;                         /* command specific data */
```

The `ioctl` function must support the two commands `SIOCSIFADDR` and `SIOCSIFFLAGS`.

SIOCSIFADDR

This command is used to set the network interface's IP address. Currently, the only address family supported is Internet. Typically, this `ioctl` gets called by the `ifconfig` utility. The driver should set the `IFF_UP` bit in the `if_flags` and call the `drvr_init` function to initialize the interface. The argument passed to the `ioctl` routine is cast to a pointer to an `ifaddr` structure, which is then used to initialize the interface's Internet address in the `arpcom` structure. The driver should also call `arpwhohas` to broadcast its Internet address on the network. This will allow other nodes to add an entry for this interface in their ARP tables.

SIOCSIFFLAGS

This command is used to bring the interface up or down and is called, for example, by the command `ifconfig` *`name`* `up`. The TCP/IP software sets or resets the `IFF_UP` bit in the `if_flags` field before calling the driver's `ioctl` entry point to indicate the action to be taken. An interface that is down cannot transmit packets.

When the interface is being brought up, the driver should call the `drvr_init` function to initialize the interface. When the interface is being brought down, the interface should be reset by calling `drvr_reset`. In both cases, the statistics counters in the `ifnet` structure should be zeroed.

The driver normally defines a flag in the statics structure which it uses to keep track of the current state of the interface (`s->ds_flags` in the example code below).

```
struct statics *s;
struct ifaddr *ifa;

case SIOCSIFADDR:
    ifa = (struct ifaddr *) arg;
    ifp->if_flags |= IFF_UP;
    drvr_init (s);
    switch (ifa->ifa_addr->sa_family) {
        case AF_INET :
            ((struct arpcom*)ifp)->ac_ipaddr = IA_SIN
                   (ifa)->sin_addr;
            arpwhohas ((struct arpcom*)ifp, &IA_SIN
                    (ifa)->sin_addr);
             break;
        default :
             break;
    }
    break;

case SIOCSIFFLAGS:
    if ((ifp->if_flags & IFF_UP) == 0 && s->ds_flags &
            DSF_RUNNING) {
        drvr_reset (s);        /* interface going down */
         s->ds_flags &= ~DSF_RUNNING;
    } else if ((ifp->if_flags & IFF_UP) &&
            !(s->ds_flags & DSF_RUNNING)) {
        drvr_init(s);           /* interface coming up */
        }
        ifp->if_ipackets  =  0  ;
        ifp->if_opackets  =  0  ;
        ifp->if_ierrors  =  0  ;
        ifp->if_oerrors  =  0  ;
        ifp->if_collisions = 0 ;
        break;
```

Watchdog Entry Point

The watchdog entry point can be used to implement a function that periodically monitors the operation of the interface, checking for conditions such as a hung transmitter. The function can then take corrective action, if necessary. If the driver does not have a watchdog function, the corresponding field in the ifnet structure should be set to NULL before calling if_attach.

The watchdog function is used in conjunction with the if_timer field in the ifnet structure. This field specifies a timeout interval in seconds. At the expiration of this interval, the TCP/IP module calls the watchdog entry point in the driver, passing it the p field from the ifnet structure as an argument. The p field is normally used to contain the address of the statics structure.

Note that the timeout interval specified by if_timer is a one-shot function. The driver must reset it to a nonzero value to cause the watchdog function to be called again. Setting the if_timer value to 0 will disable the watchdog function.

reset Entry Point

This entry point is called by the Kernel during a reboot sequence, passing it the p field from the ifnet structure, which is normally the address of the statics structure. This function may also be called internally from the driver's ioctl entry point. The function should reset the hardware, putting it into an inactive state.

Kernel Thread

The Kernel thread receives events from two sources, the interrupt handler (indicating completion of a packet transmission or reception) and the driver output routine (indicating the availability of packets on the if_snd queue). A single event synchronization semaphore is used for both these purposes. The thread should handle interrupts first and then the packets on the output queue. The general structure of the thread will look something like this:

```
struct statics *s;

for (;;) {
    swait (&s->threadsem, SEM_SIGIGNORE);
```

```
    handle_interrupts (s);/* handle any interrupts */
    output_packets (s);
                  /* start transmitter if necessary */
}
```

The precise details of the thread code will be very much dependent on the hardware architecture. The function for processing interrupts will contain the packet input code discussed above. It will also maintain the various statistics counters. Also, receiver interrupts, if disabled by the interrupt handler, are reenabled at this point. The output function will perform the tasks discussed above in the section on packet output.

Priority Tracking

Whenever the set of user tasks using the TCP/IP software changes or the priority of one of these tasks changes, the `setprio` entry point in the driver is invoked to allow the driver to properly implement priority tracking on its Kernel thread. The entry point is passed two parameters, the address of the `ifnet` structure and the priority which the Kernel thread should be set to.

```
int drvrsetprio (struct ifnet *ifp, int prio)

{
int ktid;                       /* kernel thread id */

    ktid = ((struct statics *) (ifp->p))->kthread_id;
    stsetprio (ktid, prio);
}
```

Driver Configuration File

The driver configuration file `drvr.cfg`, in the `/sys/cfg` directory, needs to declare only the `install` (and `uninstall`) entry points. The other entry points are declared to the TCP/IP module dynamically using the `if_attach` function. A typical configuration file looks something like this:

```
C:wd3e: \
    ::::: \
    :::wd3einstall:wd3euninstall
D:wd:wd3e0_info::
N:wd:0:
```

IP Multicasting Support

ether_multi Structure

For each Ethernet interface, there is a list of Ethernet multicast address ranges to be received by the hardware. This list defines the multicast filtering to be implemented by the device. Each address range is stored in an `ether_multi` structure:

```
struct ether_multi {
  u_char enm_addrlo[6];             /* low/only addr of range */
  u_char enm_addrhi[6];             /* high/only addr of range */
  struct arpcom *enm_ac;            /* back pointer to arpcom */
  u_int enm_refcount;               /* num claims to addr/range */
  struct  ether_multi *enm_next;    /* ptr to next ether_multi */
};
```

The entire list of `ether_multi` is attached to the interface's `arpcom` structure.

1. If the interface supports IP multicasting, the install routine should set the `IFF_MULTICAST` flag. For example:

   ```
   ifp->if_flags = IFF_BROADCAST | IFF_MULTICAST;
   ```

   where `ifp` is pointer to the interface `ifnet` structure.

2. Two new `ioctl`s need to be added. These are `SIOCADDMULTI` to add the multicast address to the reception list and `SIOCDELMULTI` to delete the multicast address from the reception list:

   ```
   case SIOCADDMULTI:
   case SIOCDELMULTI:
       /* Update our multi-cast list */
       error = (cmd == SIOCADDMULTI) ?
       ether_addmulti((struct ifreq *)data,
                   &s->es_ac) :
       ether_delmulti((struct ifreq *)data,
                   &s->es_ac);

       if (error == ENETRESET) {
        /*
         * Multi-cast list has changed; set the
         * hardware  filter accordingly.
         */
            lanreset(s);
            error = 0;
       }
   ```

3. The driver reset routine must program the controller filter registers from the filter mask calculated from the multicast list associated with this

interface. This list is available in the `arpcom` structure, and there are macros available to access the list. For example:

```
struct ifnet *ifp = &s->s_if;
register struct ether_multi *enm;
register int i, len;
struct ether_multistep step;

/*
 * Set up multi-cast address filter by passing
 * all multi-cast addresses through a crc
 * generator, and then using the high order 6
 * bits as an index into the 64 bit logical
 * address filter. The high order two bits
 * select the word, while the rest of the bits
 * select the bit within the word.
 */

bzero(s->mcast_filter,
            sizeof(s->mcast_filter));
    ifp->if_flags &= ~IFF_ALLMULTI;
    ETHER_FIRST_MULTI(step, &s->es_ac, enm);

    while (enm != NULL) {
        if (bcmp((caddr_t)&enm->enm_addrlo,
            (caddr_t)&enm->enm_addrhi,
             sizeof(enm->enm_addrlo)) != 0) {
 /*
  * We must listen to a range of multi-cast
  * addresses. For now, just accept all
  * multi-casts, rather than trying to set only
  * those filter bits needed to match the
  * range.
  * (At this time, the only use of address
  * ranges is for IP multi-cast routing, for
  * which the range is big enough to require
  * all bits set.)
  */
            for (i=0; i<8; i++)
                     s->mcast_filter[i] = 0xff;
            ifp->if_flags |= IFF_ALLMULTI;
            break;
        }
        getcrc((unsigned char *)&enm->enm_addrlo,
              s->mcast_filter);
        ETHER_NEXT_MULTI(step, enm);
    }
```

4. If the driver input routine receives an Ethernet multicast packet, it should set the `M_MCAST` flag in the `mbuf` before passing that `mbuf` to `ether_input`:

```
char *buf;
struct ether_header *et;
u_short ether_type;
struct mbuf *m = (struct mbuf *)NULL;
int flags = 0;

/* set buf to point to start of received frame */
```

```
...
...
et = (struct ether_header *) buf;
ether_type = ntohs((u_short) et->ether_type);

if (et->ether_dhost[0] & 1)
    flags |= M_MCAST;

/* pull packet off interface */
...
...
m->m_flags |= flags;
ether_input(ifp, et, m);
```

Berkeley Packet Filter (BPF) Support

1. If the system has a `bpf` driver installed, call `bpfattach` to register the interface with `bpf`. `bpfattach` is called indirectly via `bpf_attach_p`, which is initialized if the `bpf` driver is installed.

```
if (nbpfilter)
    (*bpf_attach_p)(&ifp->if_bpf, ifp,
    DLT_EN10MB, sizeof(struct ether_header));
```

2. If `bpf` is listening on this interface, let it see the packet before it is committed to the write. This is done by tapping data in the driver output routine as follows:

```
if (ifp->if_bpf)
    (*bpf_tap_p)(ifp->if_bpf, buf, length);
```

   where `buf` is the output buffer and `length` is the size of the packet to be transmitted.

3. If there is a `bpf` filter listening on this interface, hand off the raw (received) packet to `enet`. This is done in the packet transmit routine as follows:

```
if (ifp->if_bpf) {
    (*bpf_tap_p)(ifp->if_bpf, sbuf, slen);

/* keep the packet if it is a broadcast or has
our physical Ethernet address (or if we
support multi-cast and it is one.
*/
    if ((flags & (M_BCAST | M_MCAST)) == 0 &&
       bcmp(et->ether_dhost, s->s_enaddr,
         sizeof(et->ether_dhost)) != 0)
       return;
    }
```

where sbuf is the receiver buffer and slen is the size of the received packet.

4. The SIOCSIFFLAGS ioctl should support setting/clearing of promiscuous mode.

```
/*
* If the state of the promiscuous bit changes,
* the interface must be reset.
*/
if (((ifp->if_flags ^ s->es_iflags) &
         IFF_PROMISC) && (ifp->if_flags &
         IFF_RUNNING)) {
    s->es_iflags = ifp->if_flags;
    laninit(s);
}
```

where s is the driver statics structure.