Writing Device Drivers for LynxOS

LynxOS Release 4.0

DOC-0452-00

Contents

PREFACE xiii

For More Information xiii

Typographical Conventions xiv

Special Notes xv

Technical Support xv

LynuxWorks U.S. Headquarters xv
LynuxWorks Europe xv
World Wide Web xv

CHAPTER 1 DEVICE DRIVER BASICS 1

What is a Device Driver? 1

Types of Device Drivers 3
Device Drivers and Devices 3

LynxOS Device Driver Components 4

Entry Point Functions 6
Data Structures 8
Handling Interrupts 10
Other Components 12

LynxOS Kernel Support Functions 13

Device Driver Development and Installation 13

Referencing Device Drivers 14

Major and Minor Device Designations 15
Referencing Driver and Device IDs Under LynxOS 17
Application Access to Devices and Drivers 18

CHAPTER 2 ENTRY POINT FUNCTIONS 21

Entry Point Functions 21

Required Functions 22
Declaring the Entry Point Functions 22
install() 23
uninstall() 24
open() 25
close() 26
read() 27
write() 28
ioctl() 28
select() 30
strategy() 32

CHAPTER 3 MEMORY MANAGEMENT 37
LynxOS Virtual Memory Model 37
DMA Transfers 38
LynxOS Address Types 38
Allocating Memory 39
Memory Locking 41
mem_lock() 41
mem_unlock() 42
Address Translation 43
Virtual Address Conversion 45
Validating Addresses 46
Accessing User Space from Interrupt Handlers and Kernel Threads 47
Accessing Hardware 49
Using permap() 50
Device Access on x86 Systems 51
Device Access on PowerPC Systems 55

CHAPTER 4 SYNCHRONIZATION 61
Introduction 61
What is Synchronization? 63
Kernel Semaphores 67
Other Kernel Semaphore Functions 68
Using Kernel Semaphores for Mutual Exclusion 68
Priority Inheritance Semaphores 69
Event Synchronization 69
Handling Signals 69

Using sreset() with Event Synchronization Semaphores 70
Resource Pool Management 72
Combining Synchronization Mechanisms 73
Manipulating a Free List 73
Signal Handling and Real-Time Response 76

CHAPTER 5 INTERRUPT AND TIMEOUT HANDLING 79
Introduction 79
Timeout Interrupts 80
Interrupts and Real-Time Response 80
LynxOS Interrupt Handlers 80
iointset() 81
iointclr() 81
Sharing IRQs 82
Interrupt Vector Values 82
Interrupt Levels 83
Implementing an Interrupt Handler 86
Use of Queues 86
Interrupt Handler Considerations 88
Example Code 88
x86 IRQ Device Defaults 95
Timeout Handlers 97

CHAPTER 6 KERNEL THREADS AND PRIORITY TRACKING 101
Device Drivers in LynxOS 101
Interrupt Latency 101
Interrupt Dispatch Time 102
Driver Response Time 102
Task Response Time 102
Task Completion Time 102
Real-Time Response 103
Kernel Threads 103
Creating Kernel Threads 104
Structure of a Kernel Thread 105
Exclusive Access 105
Multiple Access 107
Priority Tracking 109

User and Kernel Priorities 111
Exclusive Access 112
Multiple Access 114
Non-Atomic Requests 115
Controlling Interrupts 116

CHAPTER 7 NETWORK DEVICE DRIVERS 119
Kernel Data Structures 119
struct ether_header 120
struct arpcom 120
struct sockaddr 120
struct sockaddr_in 120
struct in_addr 121
struct ifnet 121
struct mbuf 122
Statics Structure 127
Packet Queues 127
Driver Entry Points 128
install Entry Point 128
Initializing the Ethernet Address 129
Initializing the ifnet Structure 129
Packet Output 130
ether_output Function 130
Kernel Thread Processing 132
Packet Input 132
Determining Packet Type 133
Copying Data to mbufs 133
Enqueueing Packet 134
Statistics Counters 134
ioctl Entry Point 134
SIOCSIFADDR 134
SIOCSIFFLAGS 135
watchdog Entry Point 136
reset Entry Point 136
Kernel Thread 136
Priority Tracking 137
Driver Configuration File 137
IP Multicasting Support 138

ether_multi Structure 138

CHAPTER 8 **INSTALLATION AND DEBUGGING 141**

Static Versus Dynamic Installation 141

Static Installation 141

Dynamic Installation 142

Static Installation Procedure 142

Driver Source Code 143

Device and Driver Configuration File 143

Configuration File: CONFIG.TBL 144

Rebuilding the Kernel 145

Dynamic Installation Procedure 145

Driver Source Code 145

Driver Installation 147

Device Information Definition and Declaration 150

Device Installation 151

Node Creation 151

Device and Driver Uninstallation 151

Common Error Messages During Dynamic Installation 152

Debugging 152

Communicating with the Device Driver 153

Simple Kernel Debugger (SKDB) 154

Handling Bus Errors 154

Probing for Devices 155

Additional Notes 155

CHAPTER 9 **DEVICE RESOURCE MANAGER (DRM).. 157**

DRM Concepts 158

Device Tree 158

DRM Components 159

DRM Nodes 160

DRM Node States 162

DRM Initialization 164

DRM Service Routines 164

Interface Specification 166

Using DRM Facilities from Device Drivers 168

Device Identification 168

Device Interrupt Management 169

Device Address Space Management 170
Device I/O 170
DRM Tree Traversal 171
Device Insertion/Removal 171
Using DRM Facilities from Applications 172
Hot Swap Management Applications 175
Example Driver 177
Sample Application 182

CHAPTER 10 WRITING FLASH MEMORY TECHNOLOGY DRIVERS (MTDS) 189
Cache Management 190
Interface Overview 190
Registering with flash_mgr 190
MTD Registration Data 191
Deregistering from flash_mgr 201
Writing Callback Routines 201
Operation Code 201
Flash Virtual Base 202
Operation Parameter 202
MTD-Specific Data 203
Return Code 203
Synchronization 203

CHAPTER 11 WRITING PC CARD CLIENT DRIVERS 205
Card Services Overview 206
Card Services Initialization 206
Logical Sockets 206
Card Services Groups 206
Card Services Calling Conventions 209
Header Files 209
Synopsis 209
Client Structure 210
Detecting the Presence of Card Services 210
Client Registration 210
PC Card Identification 212
PC Card Configuration 213
Client Deregistration 215
Sample Client Drivers 215

Card Services Reference 217
    AccessConfigReg 218
    DeregisterClient 219
    ErrorName 219
    GetCardServicesInfo 220
    GetConfigurationInfo 221
    GetFirstTuple 224
    GetNextTuple 226
    ParseTuple 227
    RegisterClient 237
    ReleaseConfiguration 238
    ReleaseIO 239
    ReleaseIRQ 240
    RequestConfiguration 240
    RequestIO 242
    RequestIRQ 243
    ServiceName 244

PC Card Support 245

Installing and Removing PC Card Support 246

PC Card Support Architecture 246
    Socket Services 247
    Card Services 247
    Client Device Drivers 247
    PC Card Enabler 248
    PC Card Utilities 248

Using a PC Card 248
    Supported Cards 249
    Hot Swapping 249

Adding Support for a New PC Card 250
    Adding Support to PC Card Enabler 250
    Create New Device Driver 251

Adding Support for a New PCMCIA Adapter 251
    Supported PCMCIA Adapters 252

Troubleshooting 252

Writing PC Card Socket Services 257

Socket Services Overview 258
    Socket Services Groups 258

Socket Services Structure 259
    Header Files 259

Registration 260
Event Notification 260
Socket Services Callback 260
Socket Services Reference 261
SS_GetInfo 261
SS_InquireSocket 262
SS_SetSocket 262
SS_GetSocket 263
SS_InquireWindow 263
SS_SetWindow 264
SS_GetWindow 265

APPENDIX A PORTING LINUX DRIVERS TO LYNXOS 267
GPL Issues 267
Driver Installation 268
Using a Device 268
Major and Minor Numbers 268
Accessing a Device 269
Driver Entry Points 269
System Call Processing 270
Preemption 271
Signal Handling 272
Interrupts 272
How Linux Handles Interrupts 273
How LynxOS Handles Interrupts 273
Registering Interrupts 273
Blocking and Non-Blocking I/O 274
Bottom-Halves and Kernel Threads 274
Kernel Support 275
Kernel Timer Support 275
Semaphore Support 275
LynxOS Semaphore Support 276
Address Translation 276
Address Translation for Linux 276
Address Translation for LynxOS 277
Driver Problem Reporting 277
Communications with Applications 277
Scheduling Differences 278

Linux Scheduling ... 278
LynxOS Scheduling ... 279
Differences in Setting up a Driver ... 280
Setup ... 280
Installation ... 280
Device Access: open() and close() ... 281
Device Access: read() and write(): ... 281
Device Access: Control ... 282

APPENDIX B PORTING UNIX DRIVERS TO LYNXOS ... 285
Kernel and Driver Design Goals ... 285
Porting Strategy ... 286
Driver Structure ... 287
Overall Structure ... 287
Global Variables ... 287
Major and Minor Device Numbers ... 287
Driver Interface with Kernel ... 288
Driver Service Calls ... 288
Initialization Routine ... 294
Probing for Devices ... 295
Interrupt Handling ... 296
U Structure ... 296
u_base, u_count, u_offset ... 296
u_fmode ... 297
u_error ... 297
u_segflg ... 297
u_procp ... 297
u_tsav, u_nofault ... 297
Reentrance and Synchronization ... 298
Critical Code Regions ... 298
Event Synchronization ... 298
Driver Interface with User Applications ... 299
Entry Points ... 299
Major and Minor Device Numbers ... 300
open/close ... 301
read/write ... 301
ioctl ... 302
select ... 303

Accessing User Space 303
Returning Errors to User Application 304
LynxOS Kernel Threads 305
Dynamic Installation 305
POSIX Programming Model 305
Asynchronous I/O 305
Synchronous I/O Multiplexing and Polling 305

APPENDIX C SAMPLE DEVICE DRIVER 307
Header Files 307
ptrinfo.h 307
prtioclt.h 307
Driver Code 307

INDEX 317

Preface

This guide contains information used to support the development of LynxOS device drivers. It covers:

- Device driver structure and organization,
- LynxOS system calls that support device drivers, and
- Device driver installation and referencing under LynxOS.

For More Information

For more information on the features of LynxOS, refer to the following printed and online documentation.

- *Release Notes*

  This printed document contains late-breaking information about the current release.

- *LynxOS Installation Guide*

  This manual supports the initial installation and configuration of LynxOS and the X Windows System.

- *LynxOS User's Guide*

  This document contains information about basic system administration and kernel level specifics of LynxOS. It contains a "Quick Starting" chapter and covers a range of topics, including tuning system performance and creating kernel images for embedded applications.

- Online information

  Information about commands and utilities is provided online in text format through the **`man`** command. For example, a user wanting information about the GNU compiler would use the following syntax, where **`gcc`** is the argument for information about the GNU compiler:

```
man gcc
```

More recent versions of the documentation listed here may also be found online.

Typographical Conventions

The typefaces used in this manual, summarized below, emphasize important concepts. All references to file names 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 User's Guide.*

Environment variables, file names, 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 file name or a value that must be entered by the user	`cat` *`filename`*
	`mv` *`file1 file2`*

Blocks of text that appear on the display screen after entering instructions or commands	see below
Keyboard options, button names, and menu sequences	**Enter**, **Ctrl-C**

```
Loading file /tftpboot/shell.kdi
into 0x4000
.....................
File loaded. Size is 1314816
Copyright 2000 LynuxWorks, Inc.
All rights reserved.

LynxOS (ppc) created Mon Jul 17
17:50:22 GMT 2000
user name:
```

Special Notes

The following notations highlight any key points and cautionary notes that may appear in this manual.

NOTE: These callouts note important or useful points in the text.

CAUTION! Used for situations that present minor hazards that may interfere with or threaten equipment/performance.

Technical Support

LynuxWorks Technical Support is available Monday through Friday (holidays excluded) between 8:00 AM and 5:00 PM Pacific Time (U.S. Headquarters) or between 9:00 AM and 6:00 PM Central European Time (Europe).

The LynuxWorks World Wide Web home page provides additional information about our products and LynuxWorks news groups.

LynuxWorks U.S. Headquarters

Internet: `support@lnxw.com`
Phone: (408) 979-3940
Fax: (408) 979-3945

LynuxWorks Europe

Internet: `tech_europe@lnxw.com`
Phone: (+33) 1 30 85 06 00
Fax: (+33) 1 30 85 06 06

World Wide Web

`http://www.lynuxworks.com`

CHAPTER 1 *Device Driver Basics*

This chapter covers the basic concepts of LynxOS device drivers. The topics covered are:

- Device driver overview,
- Components of a LynxOS device driver,
- Kernel support functions overview, and
- Summary of the development and installation process under LynxOS.

This chapter covers topics applicable to the development of device drivers under LynxOS and is not intended to be an introduction to device driver development. The reader is assumed to be familiar with the concepts of driver development within a kernel environment.

What is a Device Driver?

The device driver is a software interface between the OS and hardware that hides the implementation specifics of the hardware from the OS. It provides the mechanism for the kernel to communicate with a particular type of device. Typically, these communication requests are to transfer data to and from the device or to control the device in some manner. The device driver provides the predefined and consistent interface for the kernel to make these requests. The following figure diagrams the LynxOS device driver model.

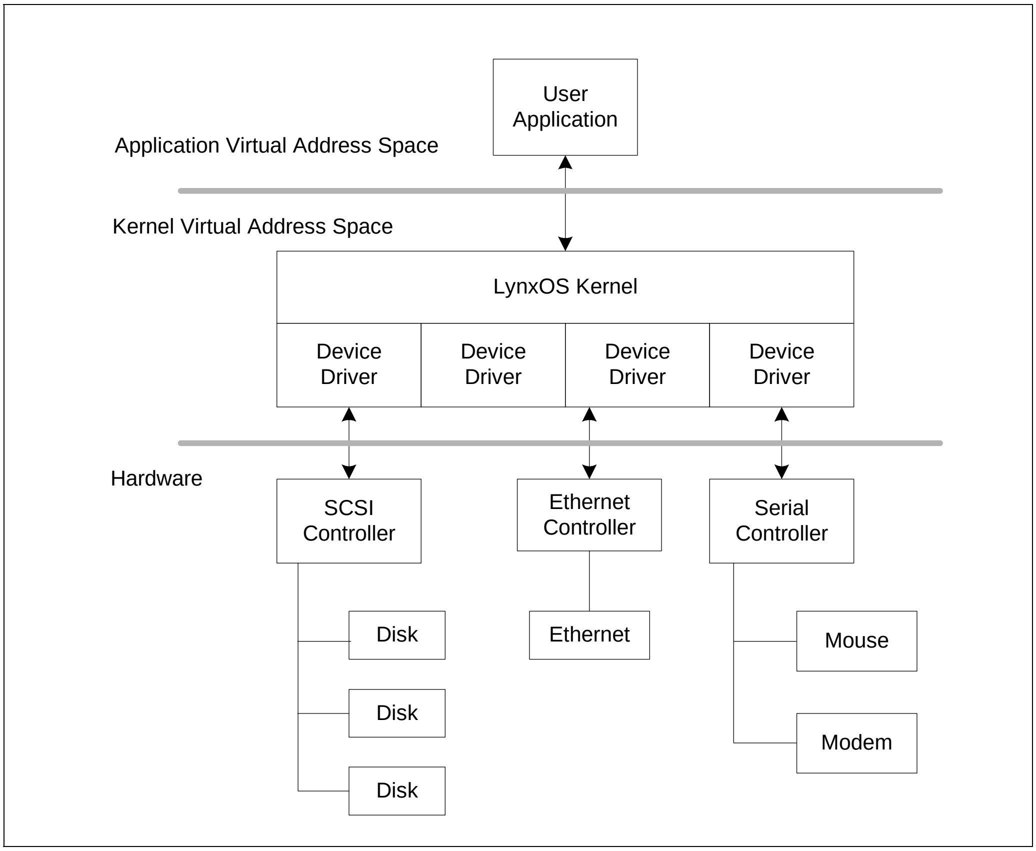


Figure 1-1: LynxOS Device Driver Interface Model

The device driver is linked into the kernel and interfaces directly with the controller card of a particular piece of hardware (drives, printers and modems, for example). An application can request access to devices using LynxOS I/O-related system calls such as `open()`, `read()`, or `write()`.

The kernel invokes the appropriate routines within the device driver code to handle the I/O requests of an application. In addition, a device driver can also be invoked in response to file system operations, interrupts, timeouts or bus errors or a change in the process using the device.

Types of Device Drivers

Device drivers are classified as either *block* or *character* types. The primary difference between the two types is that block type device drivers use a fixed size data format when transferring data and character type drivers do not.

Block type device drivers are well suited for storage devices like disk and tape drives and off-board RAM or shared memory. Block device driver characteristics include:

- Data transfers in multiples of a fixed sized buffer (512 bytes for example),
- Kernel-buffered data transfer requests,
- Concept of position on a device, and
- File system support.

Character type drivers typically support data transfer devices such as serial and parallel ports, clocks and timers, network adapters, and A/D or D/A convertors. Character type device driver characteristics include:

- Operate on arbitrarily sized data structures,
- No kernel buffering
- May or may not have concept of position on device.

Most block type devices are supported by both a block and a character type device driver. The character type driver is a counterpart to the block driver code to support the character (also known as raw) data transfer capability of the block device (CD-ROM drive, for example).

Device Drivers and Devices

The terms *device driver* and *driver* are used synonymously throughout this document. *Device* and *hardware* are used interchangeably and refer to a physical device except within the context of installation. Within this context, *device installation* refers to the process of loading a device driver and creating a device node under LynxOS for access to the device's controller. This process is covered in Chapter 8, "Installation and Debugging,". *Hardware installation* is used to refer to the act of installing a physical device.

LynxOS Device Driver Components

A LynxOS device driver consists of code and supporting data structures. The device driver also has various installation attributes that categorize its existence within the LynxOS environment.

The device driver code is a program module that is incorporated into the kernel. It does not have a `main()` routine. A basic device driver is composed of a set of *entry point* functions, an interrupt service routine, and kernel threads. (Although not an absolute requirement by the device driver interface specification, the interrupt service routine and kernel threads are essential to sustaining the hard real-time requirements of LynxOS.) Additionally, a device driver may include a timeout handler, a bus error handler, and one or more shared resources such as semaphores, buffers, and queues (refer to “Other Components” on page 12 for more information).

The supporting data structures for a device driver consists of:

- Device information
- Statics
- dldd

These data structures are implemented as C *struct* data types. The device information and statics data structures are memory buffers used by the device driver routines. The dldd data structure is only used with dynamically installed device drivers and provide the mechanism for registering the entry point function names with the kernel.

The installation attributes of a device driver become known when the driver code and device are installed into LynxOS. These attributes characterize how the device driver is incorporated into the kernel (statically or dynamically) and how the hardware it supports is accessed by the kernel (driver ID, device ID, and device nodes). The driver ID, device ID, and device node name are specific to the actual driver installation. The device driver has no knowledge of these attributes. Driver IDs, device IDs, and device node names are covered later in the chapter in “Referencing Device Drivers” on page 14.

A diagram of the LynxOS device driver is shown in the following figure.

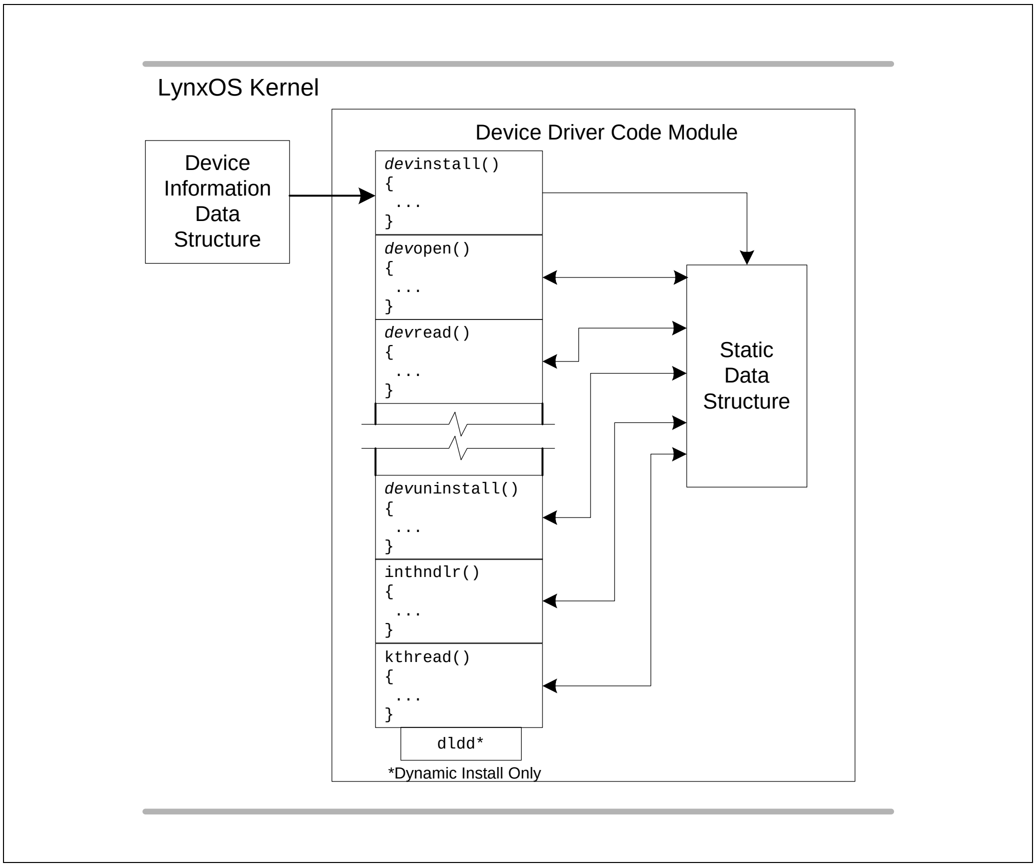


Figure 1-2: lynxos Device Driver Components

Note that the device information structure is external to the driver code. This data structure is instantiated independently of the device driver itself. The kernel passes the address of the device information structure to the driver when the device is installed. This data structure is described in the section “Device Information Data Structure” on page 8 and the process of instantiating this structure is described in Chapter 8, “Installation and Debugging,”

Entry Point Functions

The entry point functions are the main access points into the device driver. The entry point function interface is predefined and the developer must supply the code that interacts with the hardware. This section provides a brief overview of the entry point functions. They are described in detail in Chapter 2, “Entry Point Functions,” The table below summarizes the entry point functions.

Table 1-1: LynxOS Entry Point Functions Summary

Entry Point	Description
`install()`	Initializes the hardware and allocates resources required by the device driver.
`uninstall()`	Deallocates resources allocated by the device driver (shared memory and used interrupt vectors, for example).
`open()`	Initializes minor devices.
`close()`	Called only when the last open file descriptor pointing to a minor device is closed.
`read()`	Reads data from the device.
`write()`	Sends data to the device.
`ioctl()`	Executes a device-specific command.
`select()`	Supports I/O polling or multiplexing.
`strategy()`	Schedules a read or write operation on a block device.
`mmap()`	Maps data to memory.

The `install()` and `uninstall()` functions provide a mechanism for initializing and removing a device driver. Well known I/O-related system calls map to the following entry point functions: `open()`, `close()`, `read()`, `write()`, `ioctl()`, and `select()` (see next figure). The `strategy()` entry point is used to provide block-oriented I/O scheduling of reads or writes to a block device. The `mmap()` entry point is used to support data mapped to memory.

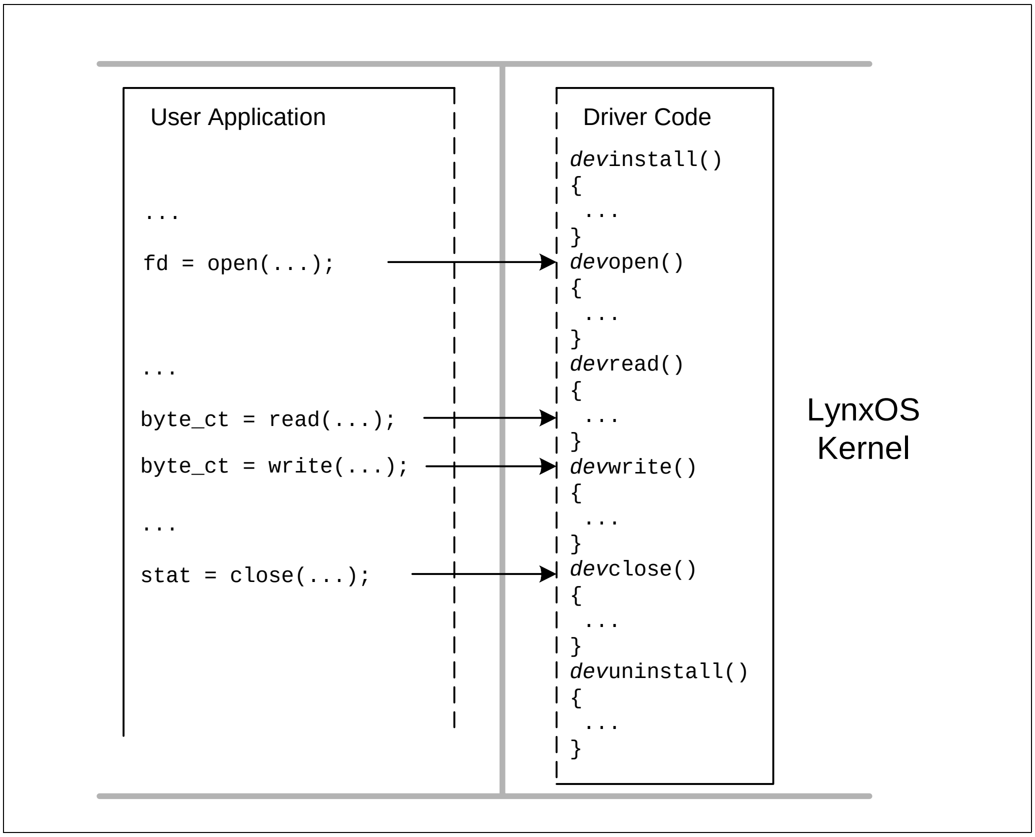


Figure 1-3: Mapping to Entry Point Functions

The set of entry points required for device drivers varies from device to device. The inherent characteristics, attributes, and purpose of the hardware determine which entry points are needed. However, for entry point routines not implemented, an empty routine, or one that simply returns the system defined constant, **`OK`**, should be provided.

Naming Convention

An identifier should be prepended to the standard entry point function names to uniquely identify them. The identifier chosen is entirely up to the developer, however, it is recommend that a convention be selected that identifies the intended device. For example, to name entry point functions for a device driver that supports the Xyz SCSI controller, `XyzSCSI_install()`, `XyxSCSI_open()`, `XyzSCSI_read()` and so on, can be used.

NOTE: In the coding examples that are used in this chapter and in the remaining chapters of this book, *`dev_`* is prepended to the entry point function names as a generic device driver identifier.

Data Structures

The data structures, *device information* and *statics*, are used in conjunction with the device driver code to provide a mechanism to pass device-specific information to the driver and to provide the driver routines with a shared memory area to place information about the state or status of the device and the driver. The *dldd* data structure is used to pass the entry point function names to the kernel for dynamically installed device drivers.

The device information and statics data structures are discussed in abstract terms because their structures are not predefined and their purposes vary. Concrete examples are provided to illustrate their usage. However, keep in mind that their field structure is determined by the particular device they are supporting and the developer is free to define them as appropriate.

Device Information Data Structure

The device information data structure is used to pass hardware-specific parameters to the device driver. These parameters (typically, configuration parameters) are essential to the proper functioning of the device driver code but would limit the driver's range of applicability if hardcoded. These parameters are typically unknown until the hardware is actually installed. They include items such as I/O address, IRQ level, and available resources.

Other uses of the device information structure include:

- Specifying specific operational characteristics of a controller card.
- Accessing special capabilities of the device.
- Accessing a specific port on a multifunction controller.
- Supporting multiple instances of a controller within the same system.

The device information data structure exists within the kernel address space. The kernel passes the address of this data structure to the `install()` entry point function. The device information data structure is deallocated when the device is uninstalled. An example of this data structure is shown below.

```
typedef struct adapter_info {
  unsigned char   scsi_id;       /* SCSI ID of host adapter */
  unsigned short  base_address;  /* adapter base I/O address */
  unsigned short  dma_chan;      /* DMA channel used */
  unsigned short  vector;        /* interrupt vector */
  unsigned short  level;         /* interrupt level */
  unsigned short  performance;
};
```

The developer is free to choose the structure name and make the field declarations appropriate to the hardware’s attributes. An empty data structure is also allowable. The data structure definition is placed into a header file and is incorporated into the device driver module using the compiler `#include` directive.

With the values contained in the device information structure, the `install()` entry point can access and initialize the hardware. The process of initializing and instantiating the device information data structure, installing devices, and the mechanism used to pass the initialized structure to the driver is discussed in Chapter 8, “Installation and Debugging,”

Statics Data Structure

The statics data structure is a memory buffer commonly shared by the functions of the device driver. It is dynamically allocated and initialized by the `install()` entry point function. Its address is returned to the kernel by the `install()` entry point. The kernel passes the address of the statics structure to other entry point functions.

The developer is free to choose the structure name and make the field declarations appropriate to the requirements of the device driver. An empty data structure is also allowable. The statics data structure definition can be placed into a header file and incorporated into the device driver module using the compiler `#include` directive or placed within the driver code module. An example statics structure is shown below.

```
struct if_3c5x9_statics {
  struct arpcom ds_ac; /* Ethernet common part */
  int flags ;            /* interface flags */
  int int_sem;           /* semaphore for interrupts */
  int io_int;            /* flag to indicate I/O interrupt */
  int xcvr_type;         /* xcvr type */
  int slot;              /* slot type: EISA or PCMCIA */
  int xmt_mon_cntr;      /* xmit hang monitor counter */
  int if_3c5x9_cip_cnt0; /* global cip counter */
  int show_overruns;
...
} ;
```

dldd Data Structure

The dldd data structure supports dynamic installation of device drivers. If used, the dldd structure is initialized within the driver code module. This data structure cannot exist in a device driver that is to be statically installed.

The names of the entry point functions are assigned to the fields of this structure and subsequently passed to the kernel when the device driver is installed. The dldd data structure is defined in `<dldd.h>`. The variable name associated with this data structure must be `entry_points`. For example:

```
#include <dldd.h>
static struct dldd entry_points =
{
  dev_open, dev_close, dev_read, dev_write,
  dev_select, dev_ioctl, dev_install,
  dev_uninstall, dev_mmap
};
```

NOTE: The *static* keyword is omitted on PowerPC platforms.

The *dev*`_mmap` field is used only by the `mem` and `zero` device drivers. All others can omit `mmap`.

For block-type drivers, *dev*`_read` is replaced with *dev*`_strategy`, and *dev*`_write` is replaced with `NULL`. The name of the data structure is `block_entry_points`. Any unused entry point function names can be replaced with `NULL`.

Handling Interrupts

Interrupts are external hardware signals delivered to the processor to indicate the occurrence of a specific event. Interrupts may signify:

- Completion of an operation
- Device has data available
- Device is ready for input or a command
- Device has changed status

Interrupt handlers are functions created by the developer to be part of the device driver code. They can be written to service the interrupt directly or can be used in conjunction with a kernel thread that can more effectively handle post interrupt processing in a scheduled and prioritized manner. Because interrupt handlers have the highest run priority, the minimization of the length of each is paramount.

Interrupt handling is covered in more detail in Chapter 5, “Interrupt and Timeout Handling,”

Interrupts and Real-Time Response

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 that is executing with real-time constraints.

Using 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.

Because the kernel thread is running at the application’s priority (actually, at half a priority level higher), 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. The use of kernel threads for servicing interrupts is covered in detail in Chapter 6, “Kernel Threads and Priority Tracking,”

Kernel Threads

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

Kernel threads and priority tracking are covered in detail in Chapter 6, “Kernel Threads and Priority Tracking,”

Other Components

Other components of a device driver include:

- Shared resources
- Timeout handler
- Error handler

These components may not be necessary in simple device drivers but may have a major role in the proper operation of more complex ones.

Shared Resources

Shared resources include memory objects such as semaphores, buffers, and queues. Semaphores are instrumental in the implementation of synchronization. They can be used as mutexes to protect critical code regions, as counters to manage shared resources, and as the gating object for event synchronization. (See Chapter 4, “Synchronization,”) Buffers and queues are data objects shared by the functions of the device drivers. These data objects must be instantiated using special system calls that allocate the appropriate type of kernel memory (Chapter 3, “Memory Management,”).

Timeout Handler

Timeouts are interrupts called by the clock interrupt handler. Timeouts can be used to generate interrupts at precise intervals of 10 millisecond granularity. LynxOS provides the system function `timeout()` to set up timeout handlers. Timeout handling is covered in Chapter 5, “Interrupt and Timeout Handling,”

Error Handler

An error handler is useful because it can change the default system behavior should a bus error occur. By default, LynxOS displays a message that a problem has occurred and attempts to halt the system. In most situations, system halts caused by bus errors can be avoided by implementing an error handler. More information on bus error handling can be found in “Handling Bus Errors” on page 154.

LynxOS Kernel Support Functions

The kernel support functions available to a device driver fall into the following categories:

- Memory management,
- Synchronization, and
- Exception handling.

Memory management functions are available for the allocation and deallocation of memory objects that the device driver uses and for validating memory pointers passed to the device driver routines. Memory management functions are covered in detail in Chapter 3, “Memory Management,”

Synchronization mechanisms which include mutual exclusion, disabling interrupts and preemption, and shared resource management are covered in Chapter 4, “Synchronization,”

Exception handling is implemented using interrupt service routines, timeout handlers and bus error handlers. These topics are covered in Chapter 5, “Interrupt and Timeout Handling,” and Chapter 8, “Installation and Debugging,”

Device Driver Development and Installation

The development process consists of defining and coding the required entry point routines and supporting functions and defining the necessary device information and statics structures and shared data resources. The table “Summary of Device Driver Components” summarized the components to be considered in the development of a device driver. An example device driver is provided in Appendix C. Other examples can be found in `/sys/devices`.

Installation consists of choosing an installation method, dynamic or static, then performing the necessary steps, based on the method chosen, to incorporate the device driver into the LynxOS kernel. Device driver installation is covered in Chapter 8, “Installation and Debugging,”

Table 1-2: Summary of Device Driver Components

Component	Description
Entry point functions	These are the core access points into the device driver. They are called by the kernel. Not all entry point functions need to be implemented.
Device information structure	This data structure is instantiated external to the device driver and its address is passed to the `install()` entry point function. It typically contains information that characterizes the major device such as I/O address, IRQ, and available resources.
Statics structure	This data structure is commonly shared by device driver entry point functions and supporting routines. Its use varies in complexity.
dldd structure	This data structure is required by device drivers that are installed dynamically.
Interrupt handler	Used to service interrupts.
Shared resources	Shared resources includes semaphores, buffers, queues, and any other data objects shared by device driver routines.
Kernel threads	Kernel threads are standard components of LynxOS and can be used in conjunction with interrupt and timeout handlers to handle the servicing of an interrupt more efficiently.
Timeout handler	A type of interrupt that is generated in conjunction with the clock interrupt handler
Error handler	Code that can prevent a system halt due to a bus error

Referencing Device Drivers

The LynxOS kernel maintains a set of tables to keep track of installed drivers and devices. Each device driver is assigned a driver ID and each installed device a device ID.

The kernel assigns driver IDs when the device code module is loaded. Each driver ID is unique. For block-type device drivers, the block driver and its character counterpart receive different driver IDs.

Device IDs are assigned when the device is installed. A device is installed when its device information block (See “Device Information Data Structure” on page 8.) is loaded into the OS.

Major and Minor Device Designations

Each installed device has a *major device* and *minor device* component associated with it and each of these components has its own ID. The major and minor device IDs are also referred to as major and minor numbers.

The major device component is essentially the instantiation of the device information block for the device. The minor device component is used in a number of ways and one or more minor devices can exist for each major device. It is also possible for a device driver to support multiple major devices.

Major devices generally refer to a single controller card. Minor devices commonly refer to a single channel (sub-device) on a controller card but may also refer to different modes of dealing with the major device.

The major device correlates to the ID assigned to the device when it is installed by the LynxOS `devinstall` command or by the `cdv_install()` or `bdv_install()` system calls. These routines associate a major device to a specific driver and loads the device information block for the device. The following figure shows the relationship between drivers and major and minor devices.

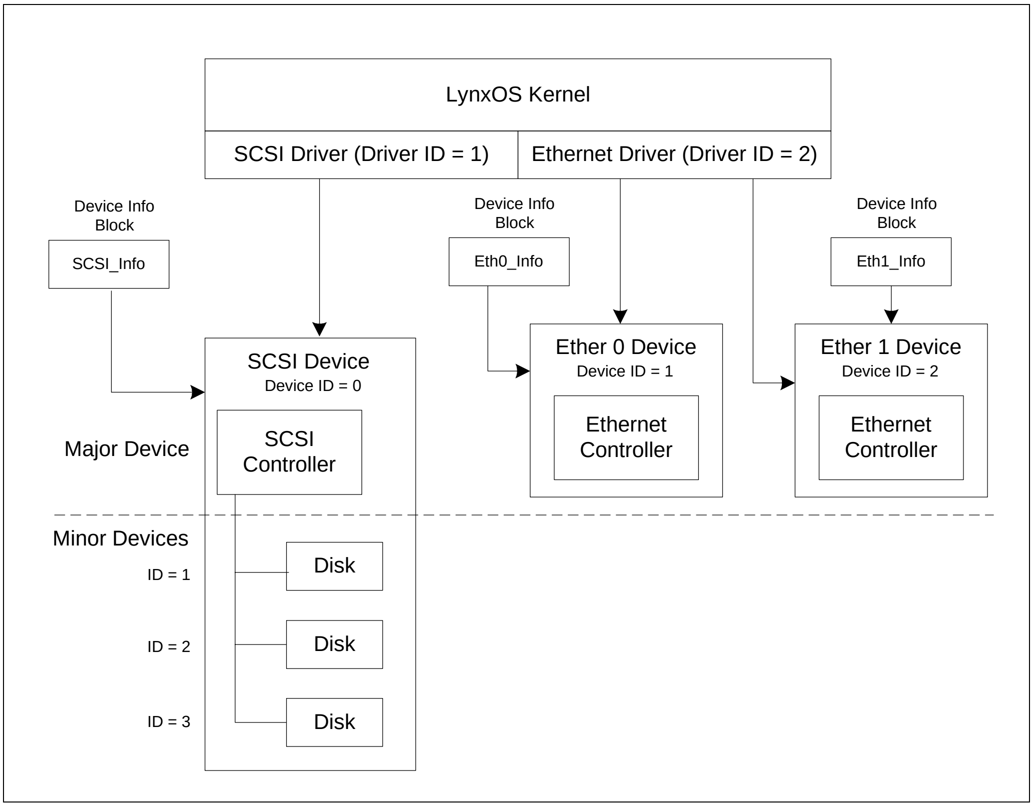


Figure 1-4: Major and Minor Devices

Some examples of minor device usage include:

- For SCSI controllers
  - SCSI target ID
  - Target type (tape or disk)
  - Partition
- For floppy disk controllers
  - Physical drive
  - Density

Minor devices are defined within the device driver code and are initialized by way of the `open()` entry point function. The meaning of the minor device is interpreted only by the device driver code. The minor device number is assigned by the developer. The minor device number is only necessary if the major device supports multiple minor devices. For hardware in which a minor device is not applicable, zero is used as the minor device number.

Referencing Driver and Device IDs Under LynxOS

Drivers and devices in LynxOS can be referenced by used their identification numbers. The LynxOS commands `drivers` and `devices` are used to list installed device drivers and devices.

Drivers

Drivers are referenced using a unique driver identification number. This number is assigned automatically during kernel configuration. 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.

Following is a sample output of the `drivers` command.

```
# drivers
id   type     major devs.   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
```

Figure 1-5: Sample `drivers` **Command Output**

Devices

Each device is identified by a pair of major/minor numbers. LynxOS automatically assigns the major numbers during kernel generation. 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. A sample output of the `devices` command is shown below. The `id` column of contains the major number of the device.

```
#devices
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
```

Figure 1-6: Sample `devices` **Command Output**

Minor devices are identified by the minor device number. These numbers may be used to indicate devices with different attributes. Minor device numbers are only necessary 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 use the minor device number in different ways: device type, SCSI target ID (e.g., a SCSI disk controller driver), or a partition (e.g., an IDE disk controller driver).

Application Access to Devices and Drivers

Like UNIX, LynxOS is designed so that devices and drivers appear as special device files in the file system. Applications can access devices and drivers using the 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.

Below is a sample listing of the `/dev` directory using the `ls -l` command (a heading as been added for clarity). The size column shows the major and minor numbers of the devices, respectively. Special device files are created with the `mknod` utility.

```
Permissions    Links     Owner     Maj,Min #  Mod. Date       Dev. 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         chris     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
```

Figure 1-7: Sample `/dev` Directory Listing

Mapping Device Names to Driver Names

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

1. 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 device name. For example, in the example above, the device `com1` would be a character device with a major device number of 6 and a minor device number of 0.
2. 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. After locating the entry for the driver in question, look in the driver column for the driver ID. For example, in the sample drivers output above, `com1` has a driver ID of 8.
3. 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 as shown in the sample drivers listing above.

CHAPTER 2 *Entry Point Functions*

This chapter describes the entry point functions and provides basic examples on their usage.

Entry Point Functions

The table below lists LynxOS entry point functions and summarizes their usage.

Table 2-1: LynxOS Entry Point Functions

Entry Point Function	Description
`install()`	Initializes the hardware and allocates shared memory buffers.
`uninstall()`	Deallocates shared memory and clears used interrupt vectors.
`open()`	Initializes minor devices.
`close()`	Called only when the last open file descriptor pointing to a minor device is closed.
`read()`	Reads data from the device.
`write()`	Sends data to the device.
`ioctl()`	Executes a device-specific command.
`select()`	Supports I/O polling or multiplexing.
`strategy()`	Schedules read and write operations on a block device.
`mmap()`	Maps data to memory.

Required Functions

Not all entry point functions are required for a device driver. The functions to implement are determined by the inherent characteristics of the device, the device driver type (block or character), and whether the device driver is statically or dynamically installed. Following are some general guidelines.

- All device drivers require `install()`.
- The `uninstall()` routine is only required in dynamically installed device drivers.
- Block device drivers use the `strategy()` entry point and character types do not.
- `strategy()` replaces `read()` in block device drivers.
- The `select()` entry point is used to support I/O polling or multiplexing.
- For entry point routines not implemented, an empty routine or one that simple returns the system defined constant, `OK`, should be provided.

Declaring the Entry Point Functions

The entry point functions are declared within the driver code module and their addresses are identified to the kernel upon driver installation. Because device drivers can be installed statically or dynamically, two distinct processes exist for registering the entry point function names with the kernel.

In a static installation, the driver code and data are incorporated into the kernel image. (This process follows the conventional UNIX model.) The entry point function names are also declared in a configuration file, which is subsequently incorporated into the kernel build process. Device driver installation is covered in detail in Chapter 8, “Installation and Debugging.”

A device driver can also be installed dynamically using the `drinstall` program or the `dr_install()` system call. A data structure named `entry_points` is used to pass the addresses of the entry point functions to the kernel. This data structure is initialized within the driver code module. Its type is `dldd`, which is defined in `<dldd.h>`. The dldd data structure is described in Chapter 1, “Device Driver Basics.” Device driver installation is covered in detail in Chapter 8, “Installation and Debugging.”

install()

The `install()` entry point function is invoked each time the device driver is installed for a major device. This entry point is responsible for initializing the major device (see “Major and Minor Device Designations” on page 15). Block- and character-type device drivers use slightly different versions of `install()`.

For character-type device drivers, the prototype for `install()` is:

```
char *install(devinfo *info)
```

where:

`info`	Is a pointer to a device information structure

For block-type device drivers, the prototype for `install()` is:

```
char *install(devinfo *info, statics *s)
```

where:

`info`	Is a pointer to a device information structure
`s`	Is a pointer to a statics data structure

`info` points to a device information data structure. (See “Device Information Data Structure” on page 8.) This data structure characterizes the major device that the instantiation of the driver supports. Typically, this structure holds configuration information such as IRQ level, I/O address, or available resources. The device driver uses this information to initialize the major device.

At a minimum, the `install()` entry point should initialize the major device and in character-type device drivers, and allocate memory for the statics data structure. (See “Statics Data Structure” on page 9.)

The `install()` entry point should also initialize the statics data structure, register the interrupt handler and kernel threads. The data received from the device information data structure should also be copied to the statics data structure for use by the other entry points, if appropriate.

The `install()` entry point returns either `SYSERR` or a pointer to a statics data structure. If a bus error occurs while `install()` is being executed, it is automatically aborted with the same effect as if a `SYSERR` has been returned.

install() Example

In the example below, the *`dev`*`_install()` routine checks for the existence of the device using `device_is_present()` (which is a supporting routine located elsewhere in the driver code). If the device is present, the statics structure (s) is allocated and initialized; otherwise `dev_install()` returns `SYSERR`.

Next, `dev_install()` attempts to initialize the hardware with the supporting routine `device_init()`. If the initialization succeeds, `dev_install()` returns the address of the statics structure; otherwise, the statics structure is deallocated and `dev_install()` returns `SYSERR`.

```
char *dev_install(devinfo *info)
{
  struct statics *s;
  int error_found = 0;

  /* Check for existence of device. If present
     allocate and initialize statics struct.  */
  if (device_is_present(info))
    {
      s = (struct statics *)sysbrk((long)sizeof(*s));
      bzero(s, sizeof (*s));
      s->io_addr = info->io_addr;
      s->intr_vec = info->intr_vec;
      ...
    }
  else
    return ((char *)SYSERR);

  /* Initialize device */
  error_found = device_init(s);
  if (error_found)
    {
      sysfree(s, (long) sizeof (*s));
      return ((char *)SYSERR);
    }
  else
    return ((char *) s);
}
```

uninstall()

The `uninstall()` entry point function is invoked when a device driver is dynamically removed from the system by way of the `devinstall` system command or the `cdv_uninstall()` or `bdv_uninstall()` system calls.

The prototype for `uninstall()` is:

```
int uninstall(statics *s)
```

where:

`s`	Is a pointer to a statics data structure

The `uninstall()` entry point must free up the statics structure, **s**, (see “Statics Data Structure” on page 9) and any other resources allocated or set by the device driver. Dynamically allocated memory for data structures or queues must be deallocated and all used interrupt vectors must be cleared. Also, if applicable to the hardware, an attempt should be made to put the device into an inactive state. The `uninstall()` entry point must return either `OK` or `SYSERR`.

uninstall() Example

In the following example, the *`dev`*`_uninstall()` routine first attempts to put the hardware into an initialized state with the `device_init()` (a supporting routine located elsewhere in the driver code). Next the interrupt vector used by the device driver is cleared and the statics structure is deallocated.

If `device_init()` succeeds, *`dev`*`_uninstall()` returns `OK`; otherwise it returns `SYSERR`.

```
int dev_uninstall(statics *s)
{
  int error_found = 0;

  /* Attempt to put the device into an initialized state */
  error_found = device_init(s);

  /* clear interrupt vector then deallocate statics structure*/
  iointclr(s->vector);
  sysfree(s, (long) sizeof(*status));

  if (error_found)
        return SYSERR;

  return OK;
}
```

open()

The `open()` entry point function is called in response to each open system call made by an application. It is used to perform minor device initialization (see “Major and Minor Device Designations” on page 15) and can also be used to register ISRs and kernel threads.

The prototype for `open()` is:

```
int open(statics *s, int devno, file *f)
```

where:

`s`	Is a pointer to a statics data structure
`devno`	Contains the major and minor device numbers
`f`	Is a pointer to a file structure

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 execute the entry point code at a time for a particular device. Therefore, synchronization between tasks is not necessary in this entry point.

`devno` contains the major and minor device numbers (`devno` is the same as `f->dev`). To extract the major and minor device numbers from `devno`, use the `major()` and `minor()` macros, respectively. Refer to man pages for more information on macros `major()` and `minor()`.

The file pointer **`f`** is defined in `<file.h>`. The `open()` entry point must return either `OK` or `SYSERR`.

open() Example

```
#include <file.h>

int dev_open(statics *s, int devno, file *f)
{
  int minDevNo, majDevNo;

  minDevNo = minor(devno);
  majDevNo = major(devno);
    /* perform initializing specific to minor device */
return (OK);
}
```

close()

The `close()` entry point function is called when the last open file descriptor pointing to a minor device is closed.

Specific allocation of memory done in the `open()` entry point routine is deallocated in the `close` entry point. As with the `open()` entry point, the `close()` entry point is not reentrant.

The prototype for `close()` is:

```
int close(statics *s, file *f)
```

where:

`s`	Is a pointer to a statics data structure

`f`	Is a pointer to a file structure

The file pointer **`f`** is defined in `<file.h>`. The `close()` entry point must return either `OK` or `SYSERR`.

close() Example

```
#include <file.h>

int dev_close(statics *s, file *f)
{

    /* perform de-initializing specific to minor device */
return (OK);
}
```

read()

The `read()` entry point function is invoked in response to a `read()` system call. This entry point function is required to copy a specified amount of data from the device into a buffer designated by the calling routine.

The prototype for `read()` is:

```
int read(statics *s, file *f, char *buf, int count)
```

where:

`s`	Is a pointer to a statics data structure
`f`	Is a pointer to a file structure
`buf`	Is a pointer to a character buffer
`count`	Specifies the number of bytes to copy

The file pointer **`f`** is defined in `<file.h>`. The `read()` entry point routine attempts to copy from the input device `count` bytes of data into character buffer `buf`. If fewer bytes of data are copied than requested, `read()` returns the number of bytes actually copied, including zero, if appropriate. If any errors occur, `SYSERR` is returned.

read() Example

```
#include <file.h>

int dev_read(statics *s, file *f, char *buf, int count)
{
  int byteCt;

    /* perform copy operation */
return (byteCt);
}
```

write()

The `write()` entry point function is invoked in response to a `write()` system call. This entry point function is required to copy a specified amount of data from a buffer designated by the calling routine to the output device.

The prototype for `write()` is:

```
int write(statics *s, file *f, char *buf, int count)
```

where:

`s`	Is a pointer to a statics data structure
`f`	Is a pointer to a file structure
`buf`	Is a pointer to a character buffer
`count`	Specifies the number of bytes to copy

The file pointer **`f`** is defined in `<file.h>`. The `write()` entry point routine attempts to copy to the output device `count` bytes of data from character buffer `buf`. If fewer bytes of data are copied than requested, `write()` returns the number of bytes actually copied, including zero, if appropriate. If any errors occur, `SYSERR` is returned.

write() Example

```
#include <file.h>

int dev_write(statics *s, file *f, char *buf, int count)
{
  int byteCt;

    /* perform copy operation */
return (byteCt);
}
```

ioctl()

The `ioctl()` entry point function 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.

For character type device drivers, the prototype for `ioctl()` is:

```
int ioctl(statics *s, file *f, int command, char *arg)
```

where:

`s`	Is a pointer to a statics data structure
`f`	Is a pointer to a file structure
`command`	Specifies the command to execute
`arg`	Is a pointer to a command argument

For block type device drivers, the prototype for `ioctl()` is:

```
int ioctl(statics *s, int devno, int command, char *arg)
```

where:

`s`	Is a pointer to a statics data structure
`devno`	Indicates a device number
`command`	Specifies the command to execute
`arg`	Is a pointer to a command argument

The file pointer **`f`** is defined in `<file.h>`.

The driver defines the meaning of `command` and `arg` except for `FIOPRIO` and `FIOASYNC`, which are predefined and used by LynxOS 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()`. (See “Validating Addresses” on page 46.)

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 Chapter 6, “Kernel Threads and Priority Tracking.”).

The `ioctl()` entry point returns `OK` or `SYSERR`.

ioctl() Example

Character type device driver:

```
int dev_ioctl(statics *s, file *f, int command, char *arg)
{
/* depending on the command, copy relevant
        information to or from the arg structure */
}
```

Block type device driver:

```
int dev_ioctl(statics *s, int devno, int command, char *arg)
{
/* depending on the command copy relevant
        information to or from the arg structure */
}
```

select()

The `select()` entry point function supports I/O polling or multiplexing.

The prototype for `select()` is:

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

where:

`s`	Is a pointer to a statics data structure
`f`	Is a pointer to a file structure
`which`	Specifies the condition to monitor
`ffs`	Is a pointer to a `sel` data structure

The file pointer **`f`** is defined in `<file.h>`.

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

The following fields are required in the statics structure to support the `select()` system call:

```
struct statics
{
...
    int *rsel_sem;      /* sem for select read */
    int *wsel_sem;      /* sem for select write */
    int *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 is a pointer to a flag that indicates whether the condition being polled by the user task is true or not. The `sel_sem` field is a pointer to a semaphore that the driver signals at the appropriate time (see below). The value of the semaphore itself is managed by the kernel and should *never* be modified by the driver. 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. For example:

```
/* 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);
}
```

select() Example

```
int dev_select (statics *s, file *f, int which, 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);
    }
```

strategy()

The `strategy()` entry point function is used only in block device drivers. It is used in place of the `read()` and `write()` entry point functions, which occur only in character device drivers.

When a process attempts to read or write from a file, the file system composes a linked list of data structures. Each data structure is of data type `struct buf_entry`, and describes the operation required for a single logical block of data. The entire linked list of these structures defines all the actions required of the device driver to complete the read or write operation.

The prototype for `strategy()` is:

```
int strategy(char *s, buf_entry *bp)
```

where:

`s`	Is a pointer to a statics data structure
`bp`	Is a pointer to the first structure in a linked list of `buf_entry` structures

The interface between the LynxOS file system and the device driver's `strategy` function is illustrated in the following diagram.

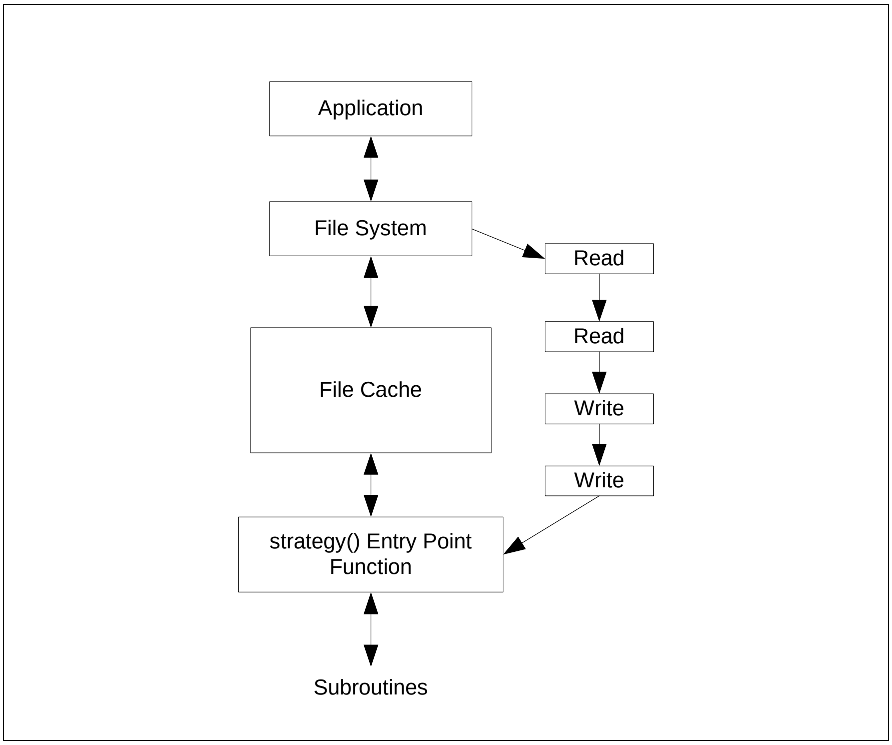


Figure 2-1: LynxOS File System and strategy() Interface

When the application uses a `read()` or `write()` system call, the file system module identifies which logical blocks of the file need to be read or written. It composes a linked list of `buf_entry` structures, each one describing the processing to be done for each logical block.

Memory blocks in the file system cache are set aside for the read or write. For a write, the data to be written into the disk block is copied to the cache location. For a read, the cache location is empty and receives the block of data when it is fetched from the disk. In each `buf_entry` structure, the pointer named `memblock` points to the corresponding cache block.

The linked list of `buf_entry` structures is used as the second argument to the `strategy()` entry point function. The `strategy()` entry point is responsible for causing the data to be moved between the cache and the disk. Typically, the actual data transfer is not accomplished by the `strategy()` entry point function itself. Instead, the `buf_entry` structures are passed to other functions that perform the transfer.

The `buf_entry` structure is defined in `/usr/include/disk.h`. It is defined as follows:

```
struct buf_entry {
    int b_device;                 /* major/minor */
    char *memblk;                 /* src or dest */
    long  b_number;               /* block number */
    int b_status;                 /* op and status */
    int b_rwsem;                  /* done sem */
    int b_error;                  /* error flag */
    struct buf_entry *av_forw;    /* next */
    struct buf_entry *av_back;    /* avail */
    int w_count;                  /* avail */
};
```

where:

`b_device`	Encoded major and minor device number
`memblk`	Memory source or destination returned by `mmchain()`
`b_number`	Source or destination block on the device
`b_status`	Specifies the type of transfer to perform, and the status of the transfer - The device is read if `B_READ` is set, or written to if `B_READ` is not set. To specify the status of the transfer, set `B_DONE` prior to signaling `b_rwsem` semaphore if the transfer is successful.
`b_rwsem`	Semaphore to `ssignal()` when the transfer is complete
`b_error`	Set non-negative if the transfer fails
`av_forw`	Pointer to the next buffer to transfer, or `NULL` if end of list (may be changed by driver)
`av_black`, `w_count`	Fields available to driver for its own purposes (LynxOS drivers use `w_count` to store the priority for priority tracking.)

NOTE: Set `B_ERROR` in addition to the `b_error` field if the transfer fails.

mmap()

The `mmap()` entry point is used to access memory-mapped character devices. The `mmap()` entry point is called as a result of the `mmap(2)` system call, allowing a character device to be mapped to memory space.

The prototype of the mmap entry point is described below:

```
mmap(struct file *f, kadd_t st, size_t len,
     int prot, int flags, off_t off)
```

Where:

`f`	a pointer to a file
`st`	the kernel address
`len`	the size of the memory space
`prot`	the bit field that specifies the protection bits
`flags`	used to set flags
`off`	the offset of memory

The file pointer **`f`** is defined in `<file.h>`. The `mmap()` entry point must return either `OK` or `SYSERR`. The `prot` field can include:

`PROT_READ`	Read access
`PROT_WRITE`	Write access
`PROT_EXEC`	Executable
`PROT_USER`	Specific User
`PROT_ALL`	Any User

mmap() Example

```
#include <rcsid.h>

/* memdrvr.c - memread */

#include <kernel.h>
#include <errno.h>
#include <sys/file.h>
#include <inode.h>
#include <memobj.h>

extern MemoryObject *Phys_mem_object;

int memopen(char *stats, int dev, struct file *f)
{
        f->inode->i_memobj = Phys_mem_object;
        return (OK);
}
int memclose(char *stats, struct file *f)
{
        f->inode->i_memobj = NULL;
        return (OK);
}

int memread(char *s, struct file *f, char *buff, int count)
{
        int i;
        char *seekp;

        seekp = (char *)(unsigned int)f->position;    /* Don’t need a  */
                                                       /* long long to  */
                                                       /* access memory */
        if (recoset()) {
                noreco();
                pseterr(EFAULT);
                return SYSERR;
        }
        for (i = 0; i < count; i++) {
               buff[i] = *seekp++;          /* If a fault occurrs seekp */
                                             /* is the fault address */
        }
noreco();
        f->position = (long long)((unsigned int)seekp);
        return i;
}

int memwrite(char *s, struct file *f, char *buff, int count)
{
        int i;
        char *seekp;

        seekp = (char *)(unsigned int)f->position;

        if (recoset()) {
                noreco();
                pseterr(EFAULT);
                return SYSERR;
        }
        for (i = 0; i < count; i++) {
               *seekp++ = buff[i];
        }
        noreco();
        f->position = (long long)((unsigned int)seekp);
        return i;
}

kaddr_t memmmap(struct file *f, kaddr_t st, size_t len,
     int prot, int flags, off_t off)
{
        /* pass through the request to the real mapping function */
        return phys_memory_mmap(f, st, len, prot, flags, off);
}
```

CHAPTER 3

Memory Management

This chapter covers memory management issues and the LynxOS system calls that support device driver memory management. It describes the LynxOS memory model, supported address types, memory allocation, memory locking, address translation, accessing user space from interrupt handlers and kernel threads, and hardware access.

LynxOS Virtual Memory Model

LynxOS uses a *virtual addressing* architecture. All memory addresses generated by the CPU are translated by the hardware MMU (Memory Management Unit). Each user task has its own protected virtual address space that prevents tasks from 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 is mapped into the lower part, the kernel into the upper part. Only the part of the virtual map occupied by the user task is remapped during a context switch. Applications cannot access the part 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 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. Because of this, precautions should be taken to restrict the access of the device driver only to necessary structures.

Please see “Accessing Hardware” on page 49 for detailed LynxOS virtual memory maps of the currently supported platforms.

DMA Transfers

All addresses generated by the CPU are treated as virtual and are converted to physical addresses by the MMU. This makes the programming of DMA transfers slightly more complicated because memory that is contiguous in virtual space can be mapped to non-contiguous 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 DMA controller.

In addition, user memory may be paged, which can lead to changes in the virtual to physical address mapping and to a physical page being reallocated to another user task. Paging must therefore be suppressed on user memory that is involved in a DMA transfer by locking the memory region.

The sections “Memory Locking” on page 41 and “Address Translation” on page 43 covers memory locking and address translation issues and functions relating to DMA transfers.

LynxOS Address Types

A LynxOS driver must deal with several different address types. These include:

Table 3-1: Address Types

Type	**Description**
User Virtual	These addresses are passed to the driver from a user application. Typically they are addresses of buffers or data structures used to transfer data between the application and a device. They are valid only when the user task that passed the address is the current task.
Kernel Virtual	These are 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 device driver. They are not accessible from an application.

Table 3-1: Address Types (Continued)

Type	**Description**
Kernel Direct Mapped	A region of the kernel virtual space that is directly mapped to physical memory (that is, contiguous virtual addresses map 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 non-system buses (PCI or VME for example) is not accessible by way of `PHYSBASE`.
Physical	Physical memory is the non-translated address for memory. Physical addresses are used when a driver needs to set up pointers to physical memory for controllers that bypass the MMU (DMA controllers, for example).
Device	Device addresses include I/O port addresses, PCI addresses, VME addresses, and so on.

Allocating Memory

The the following table summarizes the LynxOS memory allocation and deallocation functions that support device drivers. Note that these system calls *cannot* be called from within an ISR.

Allocated memory comes from the kernel address space and is not accessible to applications. Refer to section “Address Translation” on page 43 for address translation considerations. A complete description of these functions is available in their respective man pages.

Table 3-2: Memory Allocation Functions

System Call	Summary
`sysbrk()`	`sysbrk()` is useful for allocating shared static data structures and queues used by the device driver. It is recommended that `sysbrk()` be used primarily in the `install()` and `open()` entry point routines though it can be used in any of the others. The prototype for `sysbrk()` is: `char * sysbrk(long size)` `size` is the number of bytes required. `sysbrk()` returns `NULL` if no memory is available. **NOTE:** `malloc()` cannot be used within device drivers.
`sysfree()`	`sysfree()` deallocates memory allocated by `sysbrk()` and returns it to the free list. The prototype for `sysfree()` is: `void sysfree(char *p, long size)` **p** is the pointer to the memory to free and `size` is its size.
`get1page()`	`get1page()` allocates one page of physical memory and returns a virtual address that can be used to access it. `get1page()` is useful for obtaining buffers for DMA. The virtual address is above `OSBASE`, and is thus valid in the context of any task including interrupt service routines. The page of memory is `PAGESIZE` in length (4096 bytes for example) and is defined in `<kernel.h>`. The prototype for `get1page()` is: `char * get1page()` `get1page()` returns `NULL` if no memory is available.
`free1page()`	`free1page()` releases a page of memory allocated by `get1page()`. The prototype for `free1page()` is: `void free1page(char * addr)` `addr` is pointer to the page of memory to free.

Table 3-2: Memory Allocation Functions (Continued)

System Call	Summary
`alloc_cmem()`	`alloc_cmem()` returns a block of contiguous physical memory. It is useful for obtaining DMA buffers for devices incapable of scatter/gather transfers. It is recommended that `alloc_cmem()` be used primarily in the `install()` entry point routine, or if necessary, in the `open()` entry point, though it can be used any of the others. The prototype for `alloc_cmem()` is: `char * alloc_cmem(int size)` `size` is the number of bytes to allocate. `alloc_cmem()` returns a pointer to a block of pages contiguous in memory. The pointer it returns is a virtual address. If insufficient memory is available, `alloc_cmem()` returns `NULL`.
`free_cmem()`	`free_cmem()` releases the memory allocated by `alloc_cmem()`. The prototype for `free_cmem()` is: `void free_cmem(char * p, int size)` **p** is the pointer to the memory to free and `size` is its size.

Memory Locking

LynxOS provides `mem_lock()` and `mem_unlock()` system calls for locking and unlocking user memory. These functions are provided to support DMA transfers.

User memory may be paged, which can lead to changes in the virtual to physical address mapping and to a physical page being reallocated to another user task. For DMA transfers, paging must be suppressed on user memory.

If paging has not been activated, either by the `vmstart` command or a system call, these routines have no effect. Because a device driver cannot determine if paging is active, it should always lock memory used with DMA transfers.

mem_lock()

The `mem_lock()` system call is used to prevent a region of memory from being paged. `mem_lock()` cannot be used from within an ISR.

The prototype for `mem_lock()` is:

```
int mem_lock(int pid, long size, char * uaddr)
```

where

`pid`	Specifies a process ID
`size`	Specifies a memory size
`uaddr`	Specifies a start address

The arguments specify the start address (`uaddr`) and size (`size`) of a memory region in the process specified by `pid.` These typically correspond to the arguments passed to the `read()` or `write()` entry point functions of the device driver.

The `getpid()` system call can be used to get the process ID of the current task. For example:

```
/* Lock user memory to prevent paging */
if (mem_lock (pid = getpid(), uaddr, size) == SYSERR)
{
                  pseterr (ENOMEM);
                  return (SYSERR);
}
```

If paging is activated and any virtual addresses in the specified range are not currently mapped to physical memory, `mem_lock()` attempts to allocate and map the address into physical memory. If there is not enough physical memory to do this, `SYSERR` is returned. Otherwise, `OK` is returned. A successful return means that the specified virtual address range is mapped to physical memory and is locked. If paging is not activated, `mem_lock()` returns `OK`.

It is permissible to lock the same address multiple times. This may occur, for example, when locking overlapping regions. However, the memory must be unlocked the same number of times.

NOTE: Kernel memory is not paged. Therefore, memory allocated with `sysbrk()`, `get1page()` or `alloc_cmem()` does not require locking.

---

mem_unlock()

The `mem_unlock()` function is used to unlock memory locked by `mem_lock()`. `mem_unlock()` cannot be used from within an ISR.

The prototype for `mem_unlock()` is:

```
int mem_unlock(int pid, char * vaddr, long size, int dirty)
```

where:

`pid`	Specifies a process ID
`vaddr`	Specifies a start address
`size`	Specifies a memory size
`dirty`	Specifies whether or not to write memory to a swap file

```
/* Unlock user memory */
if (mem_unlock (pid, uaddr, size, dirty) == SYSERR)
{
                return (SYSERR);   /* error set to EINVAL by
                           mem_unlock */
}
```

Once memory is unlocked it becomes eligible for paging again. Attempting to unlock memory that was not previously locked causes the routine to return `SYSERR`. A successful unlock returns `OK`. If paging is not activated, `mem_unlock()` returns `OK`.

If the memory contents are modified by another processor, such as a DMA controller, the `dirty` flag should be set to 1. This ensures that the contents are written out to the swap file if the memory region is subsequently paged out.

A driver should always ensure that any locked memory is released when a task closes the device or when the device is uninstalled. Pages that remain locked after a task has exited, though usable by other tasks, will not be paged.

Address Translation

Device drivers must convert virtual addresses to their corresponding physical addresses before passing them to a DMA controller. However, the physical address range of virtual memory segments are not guaranteed to be contiguous as depicted in the figure below.

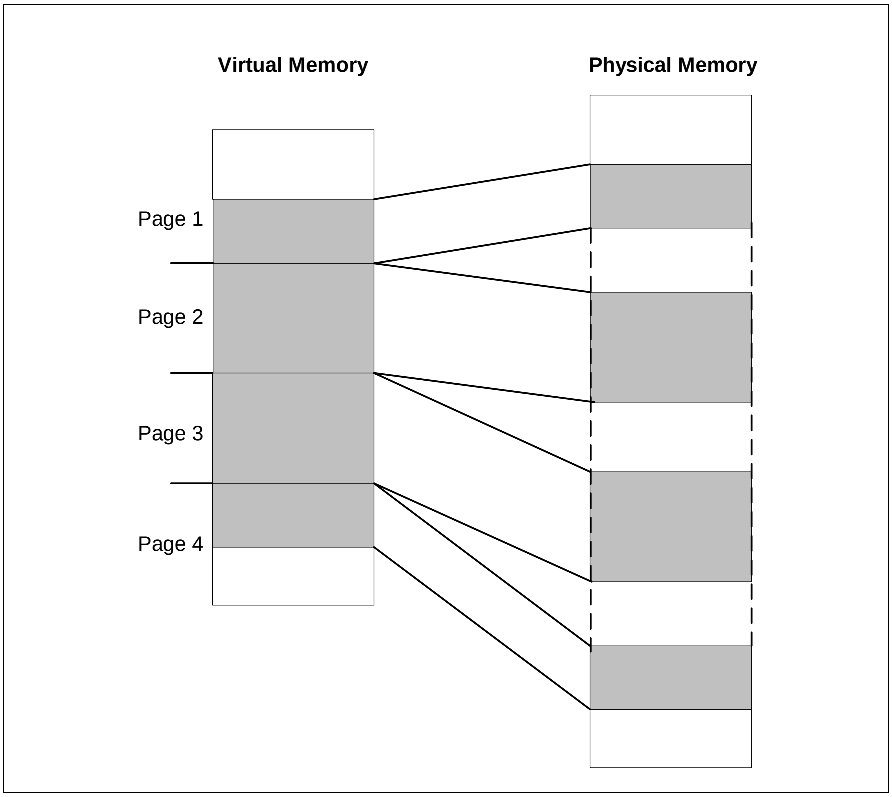


Figure 3-1: Virtual to Physical Memory Mapping

The `mmchain()` system call is used to obtain the physical addresses and sizes of all memory pages that make up a contiguous virtual memory segment. `mmchain()` cannot be used from within an ISR.

The prototype for `mmchain()` is:

```
int mmchain(dmachain *array, char *vaddr, long size)
```

where:

`array`	Pointer to a `dmachain` object
`vaddr`	Start address of a virtual memory segment
`size`	Size of the virtual memory segment

`mmchain()` writes into `array` the physical addresses and sizes of all memory pages that make up the contiguous virtual memory segment described by `vaddr` and `size`, and returns the number of elements written into `array`. If no physical memory is mapped to a virtual address, `mmchain()` sets the converted address to 0. To ensure valid mappings, `mem_lock()` should be used prior to `mmchain()`.

`array` is an array of `dmachain` structures (defined in `<mem.h>`). The size of `array` must be one more than the number of memory pages contained in `size`. The size of `array` can be computed as follows:

```
{(size + PAGESIZE - 1) / PAGESIZE} + 1
```

The physical addresses returned are offset by `PHYSBASE`. For maximum portability, the physical address should be offset by the system constant, `drambase`. The start of RAM is contained in `drambase`.

For example:

```
#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 segment used with `mmchain()` refers to the memory space owned by the current process. The system call `mmchainjob()` can be used to obtain a `dmachain` array for a specific process. Refer to the man page for `mmchainjob()` for more information.

Virtual Address Conversion

A single virtual address can be converted to its physical addresses using `get_phys()`. `get_phys()` cannot be used from within an ISR.

The prototype for `get_phys()` is:

```
char *get_phys(kaddr_t vaddr)
```

where:

`vaddr`	Is a virtual address

The address returned is a kernel direct mapped address that is offset by `PHYSBASE`. To convert this address to its physical address, `PHYSBASE` must be subtracted and `drambase` added. For example:

```
    physaddr = (get_phys (vaddr) - PHYSBASE + drambase);
```

The start of RAM is contained in `drambase`. On most platforms, this is 0, but for maximum portability, it should be used in the calculation.

Validating Addresses

The addresses passed into the `ioctl()` entry point function by application code must be validated before they can be used. The functions `rbounds()` and `wbounds()` are used for this purpose. These calls may be used in an interrupt routine.

The prototype for `rbounds()` is:

```
    long rbounds(unsigned long addr)
```

where:

`addr`	The address to be inspected

The prototype for `wbounds()` is:

```
    long wbounds(unsigned long addr)
```

where:

`addr`	The address to be inspected

The return value from `rbounds()` should be compared to the size of the object the device driver expects to be referenced. The error code `EFAULT` should be returned if `addr` is found to be erroneous.

`rbounds()` returns the number of bytes to the next boundary of non-readable memory in the virtual address space of the calling process. In other words, it returns the number of bytes readable starting at `addr`.

- If the address is in the text segment, the distance from `addr` to the end of the text segment is returned.
- If the address is in the BSS, the distance from `addr` to the current break is returned.
- If the address is in the user stack, the distance to the beginning (top) of the stack is returned.
- If the address is in a shared memory segment, the distance to the end of that shared memory segment is returned.

- If the address lies anywhere else (between the break and the current stack pointer, for example), `rbounds()` returns 0.

`wbounds()` returns the number of bytes to the next boundary of non-writable memory in the virtual address space. This works similarly to `rbounds()`, except that checking for `addr` in the text segment is not done.

Accessing User Space from Interrupt Handlers and Kernel Threads

Interrupt handlers and kernel threads execute asynchronously with respect to the user task making requests to the driver. An interrupt handler executes in the context of the task that was current when the interrupt occurred. Kernel threads execute in the context of the null process (process 0). Because the null process has no user context associated with it, the switch to a kernel thread is much quicker than to a user thread. It is sometimes necessary for an interrupt handler or a kernel thread to access a location in a user task, even though the target task may not be the currently mapped task. This requires some special considerations. The key to understanding how this can be done is the fact that there is a second virtual address that can be used to access an address in user space.

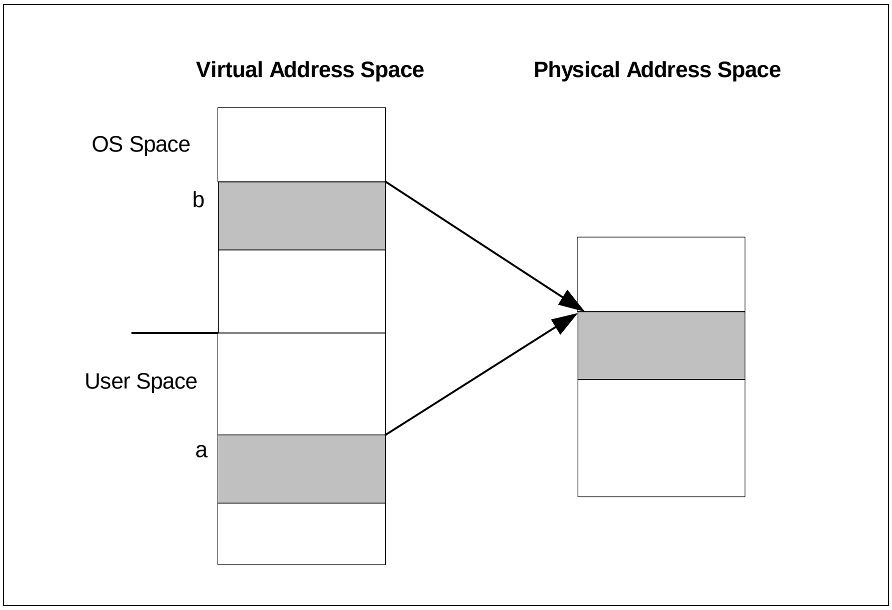


Figure 3-2: Aliasing a User Virtual Address

As the figure above shows, for any address in user space there are in fact two virtual addresses mapped to the physical address. One is the user virtual address (a). However, this mapping is valid only when the user task is the current task. So, it cannot be used from an interrupt handler or kernel thread. This mapping may also be changed if paging is activated. The second mapping is the direct mapped kernel virtual address (b). This mapping is permanent so it can safely be used anywhere in a driver.

Therefore, care must be taken when accessing user space from an interrupt handler or kernel thread. The driver must first convert the user virtual address to its corresponding kernel direct mapped address in the top-half routine (usually the `ioctl()` entry point) and then pass this address to the interrupt handler or kernel thread by way of the statics structure (see “Statics Data Structure” on page 9). The user address must also be locked to prevent the address mapping from being changed by the paging system.

In the following example, `u_addr` and `size` specify the virtual address and size of the user memory to be accessed; this is passed to the driver from the application.

```
dev_ioctl (s, f, cmd, arg)
struct statics *s;
struct file *f;
char *arg;
{
  char *kvaddr;
  ...
  if (mem_lock (getpid (), u_addr, size) == SYSERR)
  {
    pseterr (ENOMEM);
    return (SYSERR);
  }
kvaddr = get_phys (u_addr);   /* convert user addr  */
s->u_status = kvaddr;         /* save for later use */
s->u_size = size;
s->u_addr = u_addr;    /* used for unlocking memory */
s->pid = getpid ();
...
}
```

The pointer can now be used from an interrupt handler or kernel thread to access the user memory.

```
kernel_thread (s)
struct statics *s;
{
  ...
  if (s->u_status)
  {
    *(s->u_status) = status;   /* pass info to user task */
    /* unlock user memory */
    mem_unlock (s->pid, s->u_addr, s->size, 0);
    s->u_status = NULL);
  }
...
}
```

Accessing Hardware

This section describes how device drivers access the hardware layer and illustrates the virtual address mappings used by LynxOS on different hardware platforms.

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

In general, the kernel has permissions to access the full virtual address space while the user processes have restricted access. The table below shows a generalization of this concept.

Table 3-3: 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

Using permap()

The function `permap()` allows a driver to map a memory-mapped device into the kernel virtual address map so that the device can be accessed from the driver. If the memory region in which the device resides is already mapped, then it is not necessary to use `permap()`.

The virtual address `PHYSBASE` is always mapped to the physical address corresponding to the start of RAM. The size of the mapped region is equal to the system RAM size (up to 512 MB). The `permap()` function must be used to access devices that are outside of the pre-mapped region (640 KB to 1 MB). For example, in a VMEbus-based PC, the VMEbus is often mapped in the high end of the physical address space, above 512 MB. The code to map this using `permap()` would be similar to:

```
/* VMEADDR: physical address where VMEbus appears */
#define VMEADDR  0x40000000

/* VMESIZE: window size onto VMEbus */
#define VMESIZE  0x00100000

/* vme_addr: virtual address 32 bit VMEbus accesses */
unsigned long *vme_vaddr;
vme_vaddr=(unsigned long *) permap ((long)VMEADDR,(long)VMESIZE);
```

The physical address passed to `permap()` must be aligned on a page boundary. The size, in bytes, must be a multiple of `PAGESIZE`.

Device Access on x86 Systems

Reading and Writing Device Registers

The majority of devices for x86 systems exist in the CPU's I/O space, which is accessed with the `in` and `out` instructions. The file `port_ops_x86.h` (located in `$ENV_PREFIX/sys/include/family/x86/`) contains macros that can be called to read and write device registers.

The memory on I/O devices in the 640 KB to 1 MB range can be directly accessed using the `PHYSBASE` offset. The constant `PHYSBASE` is defined in `kernel.h` (in `$ENV_PREFIX/sys/include/kernel/`). For example, to access I/O devices using the `PHYSBASE` offset:

```
/* RAMBASE: RAMbase address of Ethernet controller */
#define RAMBASE 0xCC000
unsigned long * vaddr;
vaddr = (unsigned long *) (PHYSBASE + RAMBASE);
```

The following two figures illustrate the LynxOS virtual memory model on the x86 platform.

0xFFFFFFFF PHYSBASE_END	**Direct Mapped Physical Memory** (512 MB)	PHYSBASE virtual memory is mapped 1-1 with the first 512 MB physical memory
0xE0000000 PHYSBASE PERLIMIT	**permap () Region Address Space** (64 MB)	Device mapping space for non-DRM managed devices (grows down). permap() maps a physical address into the region and returns a pointer to the virtual address. The kernel also uses this space for kernel thread stacks Memory mapped in PERLIMIT is not cached. CAUTION!: permap() is never returned to the virtual map. Once the 64 MB space is exhausted, no more permap virtual space is available.
0xDC000000	**PCI Device Memory** (512 MB)	DRM maps the PCI device physical memory addresses to this space.
0xBC000000 OSEND OSLIMIT PCI_MEM_SPACE	**Kernel Memory** (text, data, BSS, heap) (64 MB)	Contains kernel text, data, BSS, heap, and the interrupt stack. Kernel's stack is currently executing context's stack. ISRs initially use current context's stack, then switch to the interrupt stack before executing driver ISR code.
0xB8000000 OSBASE	**SPECPAGE** (4 KB)	Unique process information; contains the structure pssentry.
0xB7FFF000		

Figure not to scale

Figure 3-3: LynxOS x86 Kernel Access Virtual Memory Map

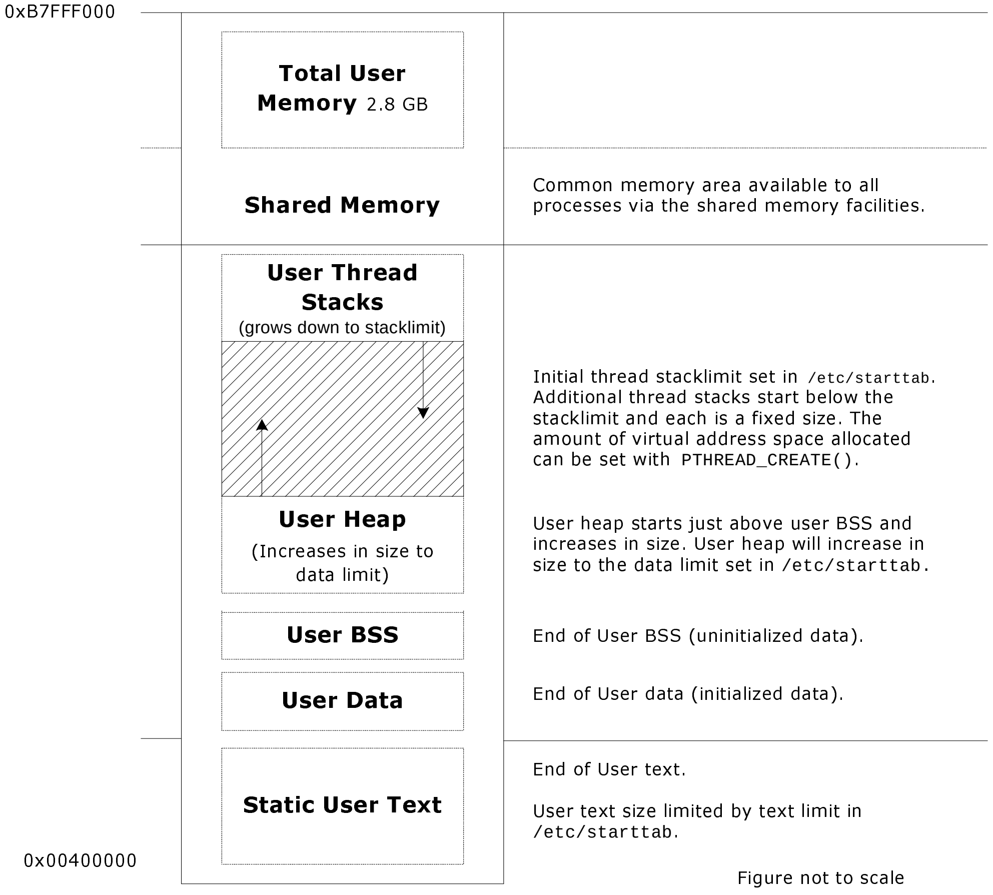

0xB7FFF000
Total User Memory 2.8 GB
Shared Memory
Common memory area available to all processes via the shared memory facilities.
User Thread Stacks
(grows down to stacklimit)
Initial thread stacklimit set in /etc/starttab. Additional thread stacks start below the stacklimit and each is a fixed size. The amount of virtual address space allocated can be set with PTHREAD_CREATE().
User Heap
(Increases in size to data limit)
User heap starts just above user BSS and increases in size. User heap will increase in size to the data limit set in /etc/starttab.
User BSS
End of User BSS (uninitialized data).
User Data
End of User data (initialized data).
Static User Text
End of User text.
User text size limited by text limit in /etc/starttab.
0x00400000
Figure not to scale

Figure 3-4: LynxOS x86 User Access Virtual Memory Map

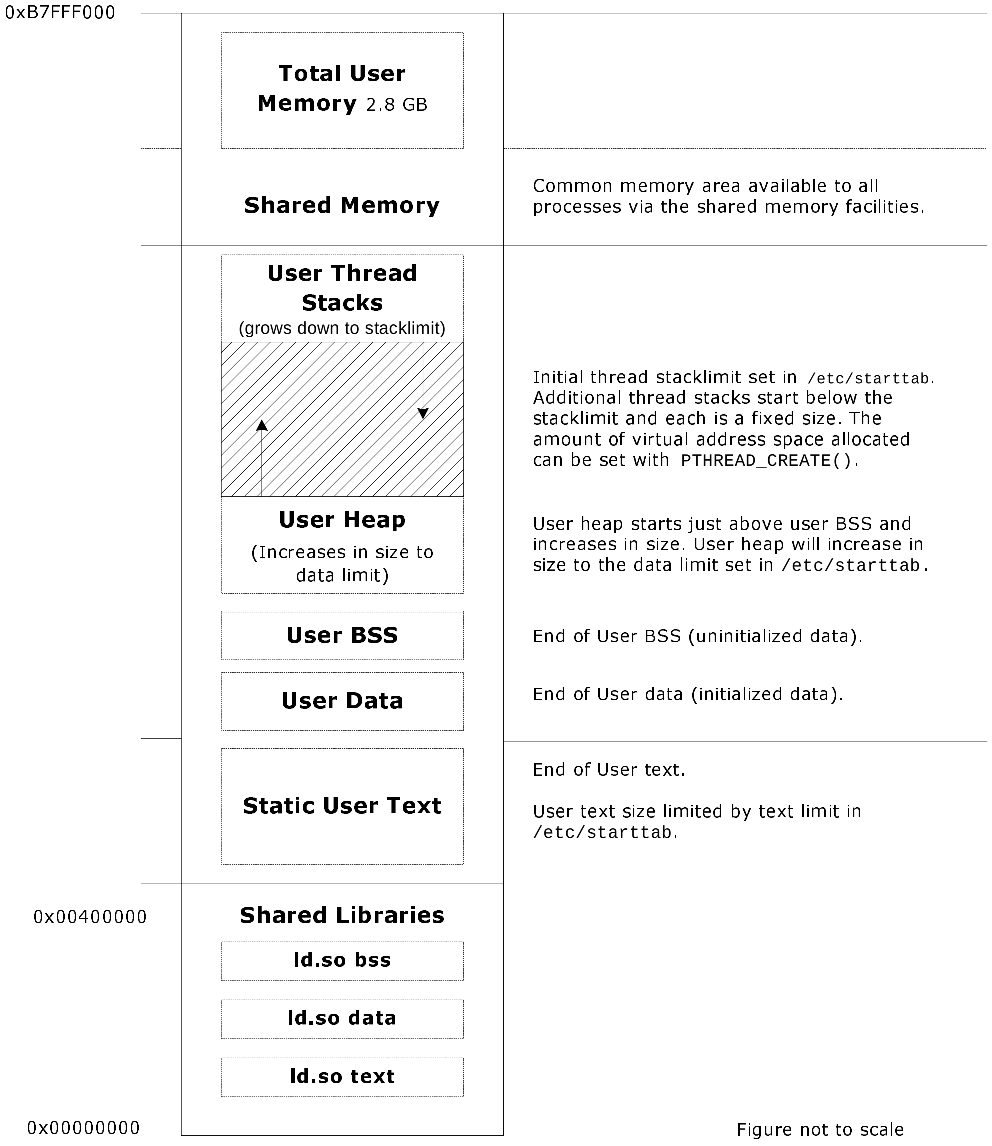

0xB7FFF000
Total User Memory 2.8 GB
Shared Memory
Common memory area available to all processes via the shared memory facilities.
User Thread Stacks
(grows down to stacklimit)
Initial thread stacklimit set in /etc/starttab. Additional thread stacks start below the stacklimit and each is a fixed size. The amount of virtual address space allocated can be set with PTHREAD_CREATE().
User Heap
(Increases in size to data limit)
User heap starts just above user BSS and increases in size. User heap will increase in size to the data limit set in /etc/starttab.
User BSS
End of User BSS (uninitialized data).
User Data
End of User data (initialized data).
Static User Text
End of User text.
User text size limited by text limit in /etc/starttab.
0x00400000
Shared Libraries
ld.so bss
ld.so data
ld.so text
0x00000000
Figure not to scale

Figure 3-5: x86 Shared Applications

LynxOS static applications are linked at a default starting address of `0`.

LynxOS dynamic user applications (applications that use shared libraries) are linked at a default starting address of `0x00400000` (4 MB).

Linux dynamic user applications are linked at a default starting address of `0x080000000` (128 MB).

`ld.so` is loaded into location `0x0` and the virtual address space between the end of `ld.so` and the start of the application is used for shared libraries.

Device Access on PowerPC Systems

ISA Bus Access

The PowerPC reference platform contains a primary PCI bus and a secondary ISA bus for system I/O. All access to the ISA bus goes through the PCI bus and the PCI-to-ISA bridge hardware. On Motorola VME systems, the VME bus interfaces to the PCI bus through the VME-to-PCI bridge hardware. The PReP reference memory map defines all physical addresses above 2 GB to be directed to the PCI bus and all physical addresses less than 2 GB as memory access (see following figure). The PCI devices on the PCI bus are configured to claim address ranges. The ISA Bridge hardware claims all unclaimed PCI addresses. This is referred to as subtractive decoding.

The physical address of all ISA devices are mapped to 2GB + *default ISA address on x86* in a 64 K address range. For example, the serial ports COM1 and COM2 reside at 0x3F8 and 0x2F8 on x86-based PCs. On the PReP reference hardware, COM1 and COM2 are mapped to 0X800003F8 and 0X800002F8. It is possible to think of ISA devices being shifted by 2 GB address as memory-mapped devices rather than I/O-mapped devices. To ensure that access to the I/O devices occurs as desired, it is necessary to add the PowerPC eieio (enforce in-order execution of I/O) instruction for access to the device. A service function `eieio()` is provided by the LynxOS kernel. Also, if there is a necessity to ensure completion of all writes, the PowerPC instruction `sync` is available as a driver service call `do_sync()`.

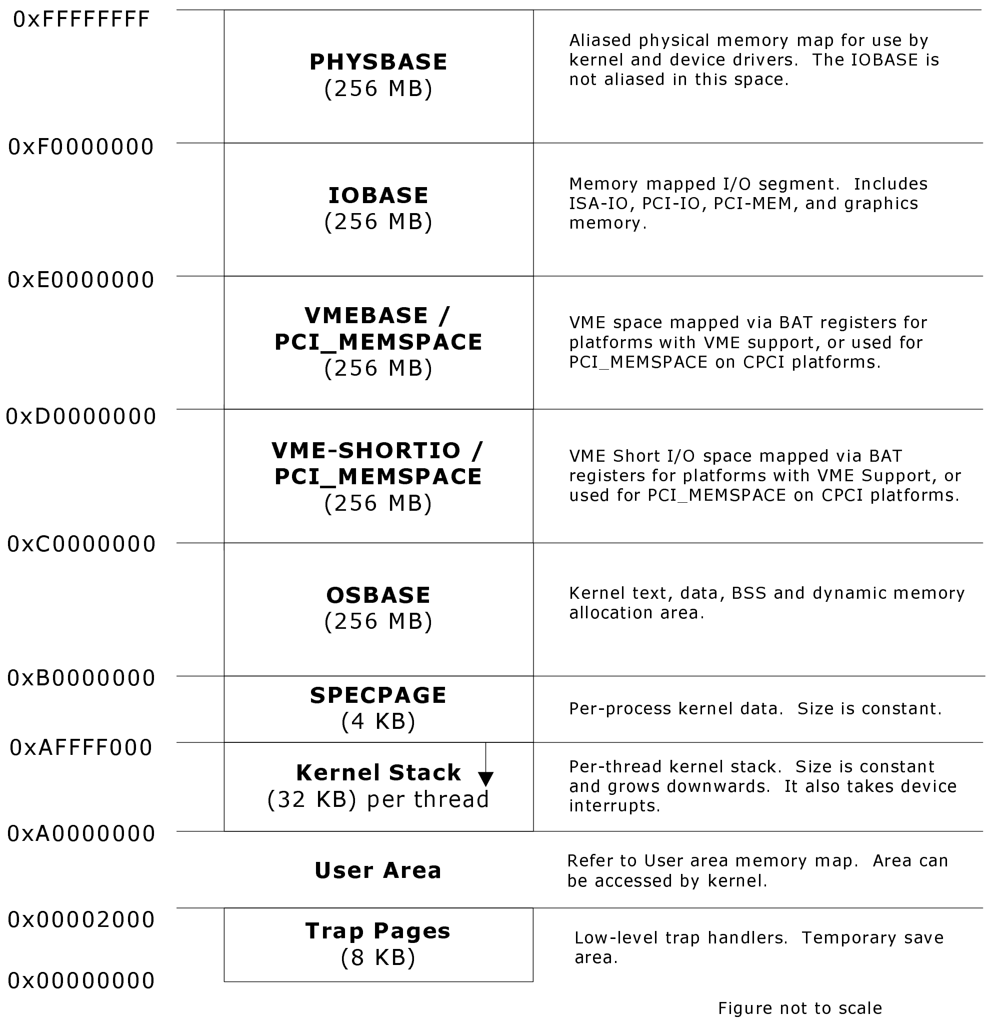
PCI Support Facilities

Earlier releases of LynxOS provided a set of functions called PCI support facilities, for accessing PCI devices on PCI Bus0. These functions have been superseded by the Device Resource Manager. New device drivers should be written using the DRM, not the PCI support facilities.

Device Resource Manager

Device Resource Manager is a LynxOS module that 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 9 describes the DRM in detail.

The following figures illustrate the LynxOS virtual memory model on the PowerPC platform.



Aliased physical memory map for use by kernel and device drivers. The IOBASE is not aliased in this space.

Memory mapped I/O segment. Includes ISA-IO, PCI-IO, PCI-MEM, and graphics memory.

VME space mapped via BAT registers for platforms with VME support, or used for PCI_MEMSPACE on CPCI platforms.

VME Short I/O space mapped via BAT registers for platforms with VME Support, or used for PCI_MEMSPACE on CPCI platforms.

Kernel text, data, BSS and dynamic memory allocation area.

Per-process kernel data. Size is constant.

Per-thread kernel stack. Size is constant and grows downwards. It also takes device interrupts.

Refer to User area memory map. Area can be accessed by kernel.

Low-level trap handlers. Temporary save area.

Figure 3-6: LynxOS PowerPC Kernel Access Virtual Memory Map

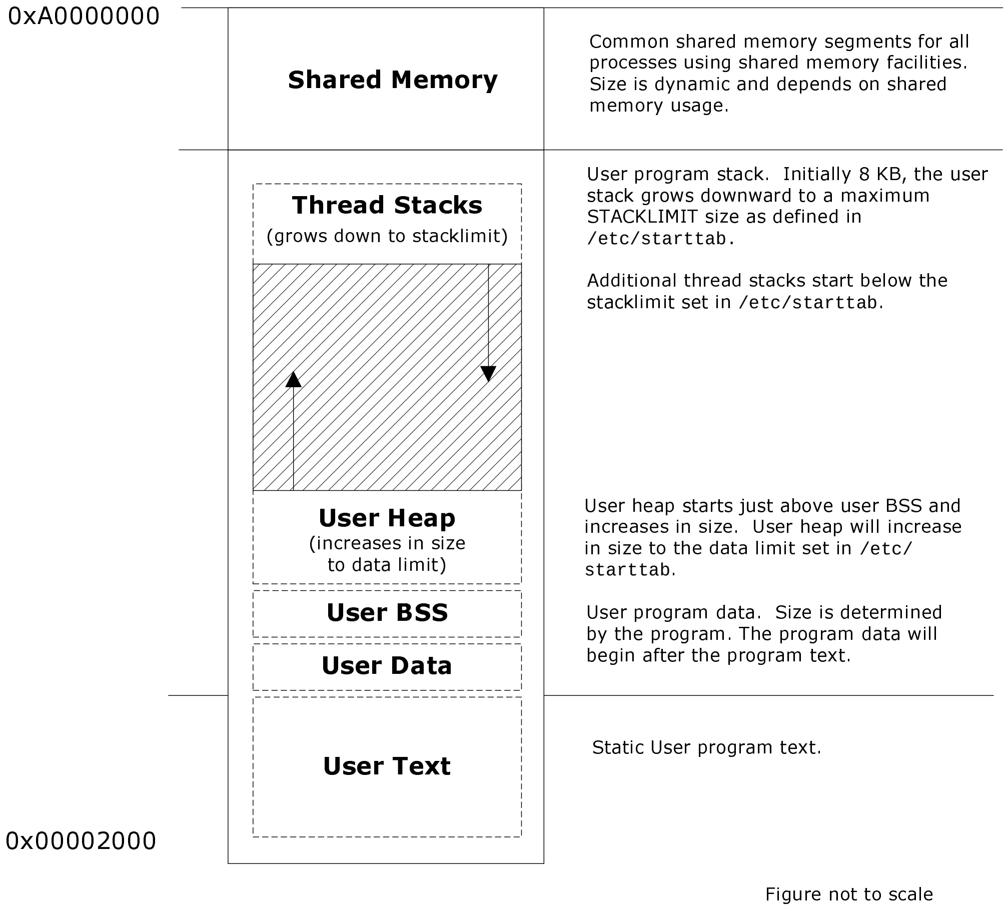

0xA0000000
Shared Memory
Common shared memory segments for all processes using shared memory facilities. Size is dynamic and depends on shared memory usage.
Thread Stacks
(grows down to stacklimit)
User program stack. Initially 8 KB, the user stack grows downward to a maximum STACKLIMIT size as defined in /etc/starttab.
Additional thread stacks start below the stacklimit set in /etc/starttab.
User Heap
(increases in size to data limit)
User heap starts just above user BSS and increases in size. User heap will increase in size to the data limit set in /etc/starttab.
User BSS
User Data
User program data. Size is determined by the program. The program data will begin after the program text.
User Text
Static User program text.
0x00002000
Figure not to scale

Figure 3-7: LynxOS PowerPC User Access Virtual Memory Map

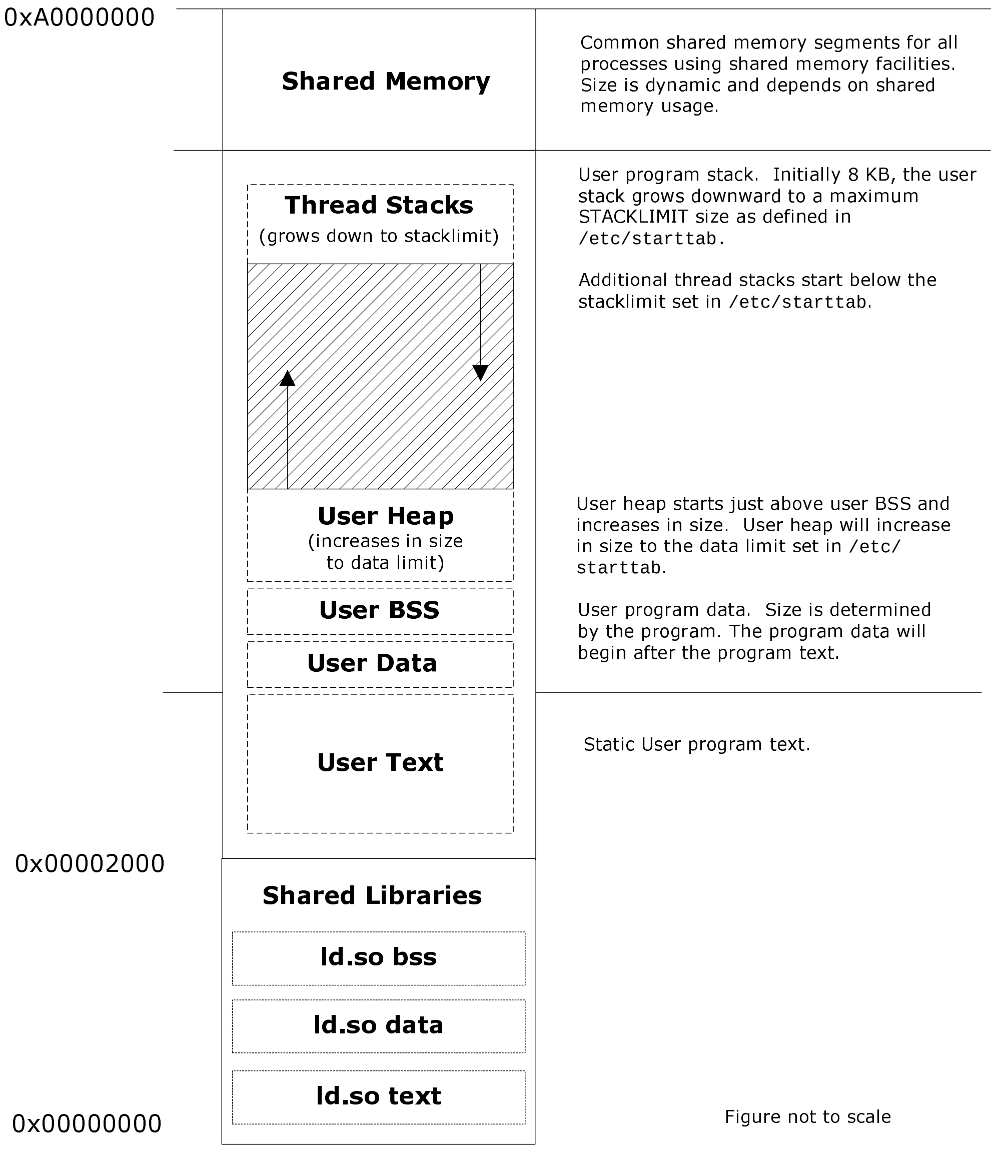

0xA0000000
Shared Memory
Common shared memory segments for all processes using shared memory facilities. Size is dynamic and depends on shared memory usage.
Thread Stacks
(grows down to stacklimit)
User program stack. Initially 8 KB, the user stack grows downward to a maximum STACKLIMIT size as defined in /etc/starttab.
Additional thread stacks start below the stacklimit set in /etc/starttab.
User Heap
(increases in size to data limit)
User heap starts just above user BSS and increases in size. User heap will increase in size to the data limit set in /etc/starttab.
User BSS
User Data
User program data. Size is determined by the program. The program data will begin after the program text.
User Text
Static User program text.
0x00002000
Shared Libraries
ld.so bss
ld.so data
ld.so text
0x00000000
Figure not to scale

Figure 3-8: PowerPC Shared Applications

LynxOS static applications are linked at a default starting address of `0x2000`.

LynxOS dynamic user applications (applications that use shared libraries) are linked at a default starting address of `0x00400000` (4 MB).

Linux dynamic user applications are linked at a default starting address of `0x080000000` (128 MB).

`ld.so` is loaded location 0x2000 and the virtual address space between the end of `ld.so` and the start of the application is used for shared libraries.

CHAPTER 4 *Synchronization*

This chapter describes synchronization issues and the LynxOS mechanisms available to device drivers to handle these issues.

Introduction

There are a number of synchronization mechanisms that can be used in a LynxOS device driver. These include:

- Kernel Semaphores
- Disabling interrupts
- Disabling preemption

Kernel semaphores can be used to protect critical code regions as well as to manage shared data and resources in a controlled manner. The functions supporting kernel semaphores include: `swait()`, `ssignal()`, `ssignaln()`, and `sreset()`.

Disabling interrupts and preemption are mechanisms used to protect code segments that are considered atomic and must be completed without interruption. The calls that support disabling of interrupts and preemption include: `disable()`, `restore()`, `sdisable()`, and `srestore()`.

The following table summarizes the LynxOS synchronization functions that support device drivers. A complete description of these functions is available in their respective man pages.

Table 4-1: Synchronization Support Functions

Call	Summary
`swait()`	`swait()` causes the calling process to wait on a semaphore. The prototype for `swait()` is: `int swait(int *s, int flag)` **`s`** is a pointer to a semaphore and `flag` is an argument that specifies whether or not signals are delivered to the process while it is waiting. `swait()` cannot be used from within an ISR.
`ssignal()`	`ssignal()` increments a semaphore and wakes up one process that is waiting on that semaphore. Processes are awakened in priority order. The prototype for `ssignal()` is: `int ssignal(int *s)` **`s`** is a pointer to a semaphore.
`disable()`	`disable()` disables interrupts and task preemption. The prototype for `disable()` is: `void disable(int ps)` **`ps`** must be a local stack variable of the invoking function.
`restore()`	`restore()` restores interrupts and task preemption. The prototype for `restore()` is: `void restore(int ps)` **`ps`** is the same variable used in the corresponding `disable()` call.
`sdisable()`	`sdisable()` disables task preemption. The prototype for `disable()` is: `void disable(int ps)` **`ps`** must be a local stack variable of the invoking function.

Table 4-1: Synchronization Support Functions (Continued)

Call	Summary
`srestore()`	`srestore()` restores task preemption. The prototype for `srestore()` is: `void srestore(int ps)` **`ps`** is the same variable used in the corresponding `sdisable()` call.
`sreset()`	`sreset()` wakes up all processes that are waiting on a semaphore and sets the semaphore value to zero. The prototype for `sreset()` is: `void sreset(int *s)` **`s`** is a pointer to a semaphore.
`signaln()`	`ssignaln()` signals a semaphore a specified number of times. The prototype for `signaln()` is: `int signaln(int *s, int count)` **`s`** is a pointer to a semaphore and `count` is the number of times to signal the semaphore.
`scount()`	`scount()` returns the value of the semaphore. The prototype for `scount()` is: `int scount(int *s)` If the value of **`s`** is negative, it indicates the number of processes that are waiting on the semaphore. If **`s`** is zero, no processes are waiting on the semaphore. If the value is greater than zero, it represents the number of times the semaphore can be waited on without having to wait for the semaphore to be signaled (see `ssignal()`).
`pi_init()`	Used to initialize a priority inheritance semaphore.

What is Synchronization?

Synchronization ensures that certain events occur in a definite order within a non-deterministic environment (such as a concurrent, preemptive operating system). In a device driver this usually means ensuring that shared resources such as devices, buffers, queues, an so on are accessed in a protected and controlled manner so that processes do not interfere with each other’s access to shared resources.

Synchronization provides:

- Methods that support the coordinated use of shared resources by causing processes to suspend execution when a shared resource is not available.
- Protection to critical code sections. These are code segments considered to be atomic that must be all completed or not at all.
- Mechanisms to prevent system failure due to inherent conditions of concurrent and preemptive operating system environments such as race conditions and deadlock.

Managing Shared Data Resources

Semaphores are a mechanism available to LynxOS device drivers to manage shared resources (statics structure and shared buffers and queues, for example). Semaphores can partition the device driver code into critical code regions that must obtain access to a shared resource before continuing to execute. The semaphore is a mechanism used to lock and release a shared resource. Code that must access the shared resource can only do so if the resource is unlocked. If the shared resource is unlocked, the code locks it and proceeds. If the shared resource is locked, the code must wait (block) until the resource becomes free.

The mechanism of locking and releasing shared resources with semaphores is described in more detail in “Kernel Semaphores” on page 67.

Protecting Critical Code Sections

Within a device driver, it is necessary to prevent interrupt routines from accessing shared data or resources such as buffers or queues that are being modified by a process. To accomplish this, interrupts can be disabled with the `disable()` function and subsequently re-enabled with the `restore()` function.

It is important to keep the code being executed between the `disable()` and `restore()` functions short in order to avoid degradation of the overall system response to interrupts. (Note that `disable()` also disables task preemption.)

Following is a basic example using `disable()` and `restore()`:

```
int ps;
disable (ps);   /* disable all interrupts */
...
...
restore (ps);     /* restore interrupt state */
```

The variable **`ps`** must be a local variable and should never be modified by the driver. Each call to `disable()` must have a corresponding call to `restore()`, using the same variable.

NOTE: The `restore()` function actually restores the state that existed before `disable()` was called. So, if interrupts were already disabled when `disable()` was called, the first call to `restore()` does not re-enable them.

The `sdisable()` and `srestore()` functions are used to disable task preemption only. Disabling of task preemption is necessary to prevent the kernel, other drivers, or applications from accessing shared data and resources while they are being modified by a device driver process. The kernel continues to handle interrupts while preemption is disabled.

The `sdisable()` and `srestore()` functions are used in much the same way as `disable()` and `restore()`. Following is a basic example of `sdisable()` and `srestore()`:

```
int ps;
sdisable (ps);  /* disable task preemption */
...
...
restore (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: The `srestore()` function actually restores the state that existed before `sdisable()` was called. So if interrupts were already disabled when `sdisable()` was called, the (first) call to `srestore()` does not re-enable them.

A critical code region is blocked out by the `disable()`/`restore()` or `sdisable()`/`srestore()` calls. Within a device driver, the critical code region should only contain the instructions necessary to complete an atomic transaction on a shared resource and interrupts and task preemption must be re-enabled immediately after the transaction is complete.

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()`/`srestore()` is completely redundant, as preemption is already disabled by the outer `disable()`.

Avoiding Deadlock & Race Conditions

Deadlock typically occurs when two semaphores are not accessed in the same order in two different processes (or threads). As a result, each process is holding a semaphore and is waiting to gain access to the semaphore that the other process is holding. In this condition the processes wait forever for a semaphore that will never be released.

Deadlock can be avoided by ensuring that multiple semaphores are always acquired in the same order by every process. This ensures that two processes do not gain access to two different semaphores and wait indefinitely for the other to release the second semaphore.

Race conditions occur when two or more processes access the same shared resource at the same time. In particular, problems occur when a process that is accessing a shared resource gets preempted by another process that accesses the same resource and changes the state of that resource before the first process has completed its transaction on the resource. The result is that the first process is now working with a compromised version of the shared resource.

To avoid race conditions, shared data and resources must be accessed in a controlled manner. The code that accesses shared resources should be considered a critical code region, which can be protected from preemption by disabling interrupts or preemption.

Kernel Semaphores

A kernel semaphore is an integer variable that is declared by the device driver. Semaphores must be visible in all contexts. This means that the memory for a semaphore must not be allocated on the stack.

Kernel semaphores are counting semaphores, they can be initialized to any non-negative value. A semaphore is acquired using the `swait()` function.

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 wait queue of the semaphore. Tasks on this queue are kept in priority order.

A semaphore is signaled using the `ssignal()` function. 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 initialized 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.

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.

Other Kernel Semaphore Functions

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

`ssignal(n)`	Used to signal a semaphore *n* times. This is equivalent to calling `ssignal()` *n* times.
`sreset()`	Resets the semaphore value to 0 and wakes up all tasks that are waiting on the semaphore.
`scount()`	Returns the semaphore value.

Using Kernel Semaphores for Mutual Exclusion

When used to protect a critical code region, the kernel semaphore should be initialized to 1 or -1. 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.

Priority Inheritance Semaphores

In a multi-tasking 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 preempted by a higher priority task that 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 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);
```

This feature is should only used in the context of a kernel semaphore being used as a mutex mechanism.

Event Synchronization

A kernel semaphore is the mechanism used to implement event synchronization in a LynxOS driver. The value of the semaphore 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

Because there is often no guarantee that an event will occur, 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 is no way to interrupt the task in the case of problems, so the task can remain blocked indefinitely. The driver must check the return code from `swait()` to determine whether a signal has been 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); /* prepare for possible termination by signal handler */
  deliversigs (); /* may never return */
}
```

Using sreset() with Event Synchronization Semaphores

Two example uses of `sreset()` discussed below are:

- 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 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 with `sreset()` uses 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 below.

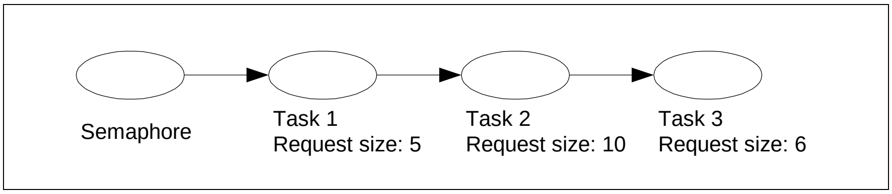


Figure 4-1: Synchronization Mechanisms

When 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 then consume the available data according to their priorities. Tasks that are awakened but find no data 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. Because 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 design where an unknown number of tasks could be blocked on a semaphore. One possibility is to wake tasks in a cascade manner. The call to `sreset()` is replaced by a call to `ssignal()`, which wakes up the first blocked task. This task is then responsible for unblocking the next blocked task, which unblocks 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);
}
```

Because 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 with using this technique.

Resource Pool Management

LynxOS kernel semaphores can also be used as a counting semaphore 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 (s)
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 statics *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 an actual driver, the scenarios are often far more complex and require combining different techniques. The following sections discuss when and how synchronization mechanisms should be combined.

Manipulating a Free List

This example illustrates the use of interrupt disabling to remove an item from a free list, but in particular, what the driver can 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 (s)
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 (s->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:

- The example uses `SEM_SIGIGNORE` for simplicity. If `SEM_SIGABORT` is used, the return value from `swait()` must be checked.
- The example uses the `disable()/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()/srestore()` can be used instead.
- 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 signalled 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.
- Blocking with interrupts disabled may seem at first like a dangerous thing to do. This is necessary, as restoring interrupts before the `swait()`

would introduce a race condition. LynxOS 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.

- `swait()`/`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 possibility of a deadlock. To avoid a deadlock, `sdisable()`/`srestore()` or `disable()`/`restore()` should be used as the mutex mechanism rather than `swait()`/`ssignal()`. But, once again, the critical code region must be kept as short as possible to avoid having an 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()`/`ssignal()` to be used as the mutex mechanism.
- A call to `ssignal()` in `put_item()` may make a higher priority task eligible to execute but the context switch will not occur until preemption is re-enabled 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:

- A kernel semaphore used as a counting semaphore incorporates the functionality of an event synchronization semaphore. `swait()` blocks when no items are available and `ssignal()` wakes up waiting tasks.
- The example uses the `disable()`/`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()`/`srestore()` can be used instead.
- The event synchronization is outside of the critical code region so there is no possibility of deadlock. Therefore, `swait()`/`ssignal()` could be used as the mutex mechanism if the code does not need to be called from an interrupt handler.
- 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 69 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 `disable()`/`restore()` or `sdisable()`/`srestore()`. 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 re-enable interrupts or 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 pre-emption before delivering signals */
  deliversigs (); /* may never return */
  sdisable (ps);
}
...
srestore (ps);     /* leave critical region */
```

CHAPTER 5

Interrupt and Timeout Handling

This chapter discusses issues related to the design and implementation of interrupt service routines (ISRs) and timeout handlers.

Introduction

Interrupts are external hardware exception conditions that are delivered to the processor to indicate the occurrence of a specific event. ISRs are useful for:

- Indication of the completion of an operation

  For example, an interrupt could be generated indicating the completion of a DMA (Direct Memory Access) 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, for example.

- Device ready for a command

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

Timeout Interrupts

LynxOS timeout handlers are called by the clock interrupt handler and therefore are considered to be similar to any interrupt handler. Instructions for setting up timeout handlers are provided in "Timeout Handlers" on page 97.

Interrupts and Real-Time Response

A task, regardless of its priority, is interrupted if an interrupt is pending and interrupts are enabled. This could result in low priority interrupt service routines executing before high priority tasks that have real-time constraints.

To offload processing from interrupt-based sections of a device driver, LynxOS offers a feature known as kernel threads. Kernel threads are independently schedulable entities that closely resemble processes but do not have the memory overhead associated with processes.

Using kernel threads, delays are significantly reduced. Instead of the interrupt service routine handling all the servicing of the interrupt, a kernel thread is used to perform the function previously performed by the interrupt routine. A kernel thread 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. The use of kernel threads is covered in detail in Chapter 6, "Kernel Threads and Priority Tracking."

LynxOS Interrupt Handlers

Interrupt handlers in LynxOS are specified in the `install()` or `open()` entry point functions and are cleared in the `uninstall()` or `close()` entry points. Interrupt handlers run before any other kernel or application processing is completed.

Interrupt handlers are declared and reset using the functions `iointset()` and `iointclr()`. The table below summarizes `iointset()` and `iointclr()`.

Table 5-1: iointset() and iointclr() Summary

`iointset()`	The `iointset()` function specifies the routine to run when an interrupt occurs.
`iointclr()`	The `iointclr()` function removes an interrupt vector from the interrupt vector table.

Interrupt handlers 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. Refer to Chapter 3, “Memory Management.” for more information on address translation.

iointset()

The device driver registers its interrupt handler routine with the LynxOS interrupt dispatcher using the `iointset()` function call. The interrupt dispatcher subsequently calls the interrupt handler routine when an interrupt occurs. `iointset()` cannot be called from within an ISR.

The prototype for `iointset()` is:

```
int iointset(int vector, int (*function)(), char *argument)
```

where:

`vector`	Is an interrupt vector number
`function`	Is a pointer to the interrupt handler function
`argument`	Is a pointer to an argument string

The interrupt vector number used by the hardware sending the interrupt is specified by the `vector` argument. The interrupt dispatcher calls the routine given by `function`. The interrupt dispatcher passes `arguments` to the interrupt handler.

On x86 and SPARC systems, `iointset()` returns the index of the previous interrupt vector. This can be used with `ioint_link()` to share interrupt vectors.

NOTE: When using DRM, register interrupt handlers with `drm_register_isr()`, not `iointset()`.

iointclr()

The `iointclr()` function clears an interrupt vector from the stack of interrupt handlers. `iointclr()` cannot be called from within an ISR.

The prototype for `iointclr()` is:

```
void iointclr(int vector)
```

where:

`vector`	Is an interrupt vector number

The `vector` argument specifies the interrupt vector number to clear.

For each interrupt vector, the kernel maintains a stack of interrupt handlers. If a device driver installs a new interrupt handler at a position occupied by an existing handler, the old handler is reinstalled when `iointclr()` is called for `vector`.

NOTE: The default handler for interrupts is code that causes kernel panic and a halt.

Sharing IRQs

`ioint_link()` is used on hardware where two or more drivers must share the same interrupt vector. To share an interrupt, each interrupt routine must check if its device has interrupted and then act accordingly; it must also call the next interrupt routine on the list that has the same hardware vector.

When a device driver shares an interrupt with other drivers, it can use the value returned by `iointset()` as a key to the next interrupt handler on the stack of interrupt handlers. After processing the interrupt, the interrupt handler can use `ioint_link()` to cause the next interrupt handler in the stack to be dispatched. For example:

```
dev_install(info)
{...
  s=sysbrk(...);
  ...
  s->key = iointset(vector, int_handler, s);
  ...

int_handler(s)
{
  ...
  ...
  ioint_link(s->key);
}
```

Interrupt Vector Values

x86

On the x86 platform, `iointset()` and `iointclr()` require the interrupt vector number, not the address of the vector. The interrupt vector number (0–15) must also be offset by 32. For example:

```
iointset (irq + 32, intr_handler, s);
iointclr (irq + 32);
```

PowerPC

Many PowerPC systems have two or more interrupt controllers. For example the Motorola 16xx series and the PowerStack series have an i8259-compatible interrupt controller to handle ISA interrupts and the VMECHIP2 handles VME interrupts. On the Motorola PowerPlus systems, there are three interrupt controllers. There is an MPIC device, which is the master interrupt controller, and handles interrupts from PCI, timers, and cascaded interrupt controllers. There is a i8259-compatible interrupt controller to handle ISA-based interrupts. The Universe VME controller handles the VME interrupts on some of the PowerPlus systems.

The PowerPC processor has only one interrupt input. The LynxOS interrupt dispatch routine determines the source of the interrupt and uses a table with 256 entries to dispatch the interrupt service routine. The `vector` parameter to `iointset()` is the index to this table.

To maintain compatibility with x86-based device drivers, the ISA interrupt controller uses `MASTER_BASE (32)` as the base vector. The PowerPlus systems use the `MPIC_BASE (4)` as the base vector for the MPIC. The VME controllers use the `VME_IRQBASE (48)` as the base vector.

The interrupt vector space is divided as follows:

0...3	Reserved vectors
4...28	MPIC vectors
29...31	Reserved vectors
32...48	ISA vectors
48...255	VME vectors

As in the x86-based systems, `iointset()` returns a key, which can be used in `ioint_link()` to share interrupts. For device drivers using the PCI service functions, the `pci_what_irq(slot)` returns a vector, which can be used directly in the `iointset()` function.

Interrupt Levels

When a device generates an interrupt, the interrupt signal is propagated from the device through the bridge or bus controller. Some boards may have a single interrupt controller, while others may have two or more.

The figure below shows a typical configuration on x86-based computers, where two Intel 8259 interrupt controllers are cascaded. INT1 is called the master controller, because it is closest to the CPU. INT2 is called the slave controller.

When one of the IRQ input lines of an interrupt controller is asserted, the controller asserts its int+ output line. For INT1, this signals the CPU that an interrupt has occurred. For INT2, the interrupt signal is passed to INT1, which then signals the CPU.

When the CPU is signaled, it communicates with the interrupt controllers to determine which IRQ line has been asserted. The CPU executes the interrupt handler that was most recently registered for that IRQ. When the handler terminates, the CPU continues where it left off before executing the handler.

The Intel 8259 interrupt controllers are cascaded as shown in the figure below.

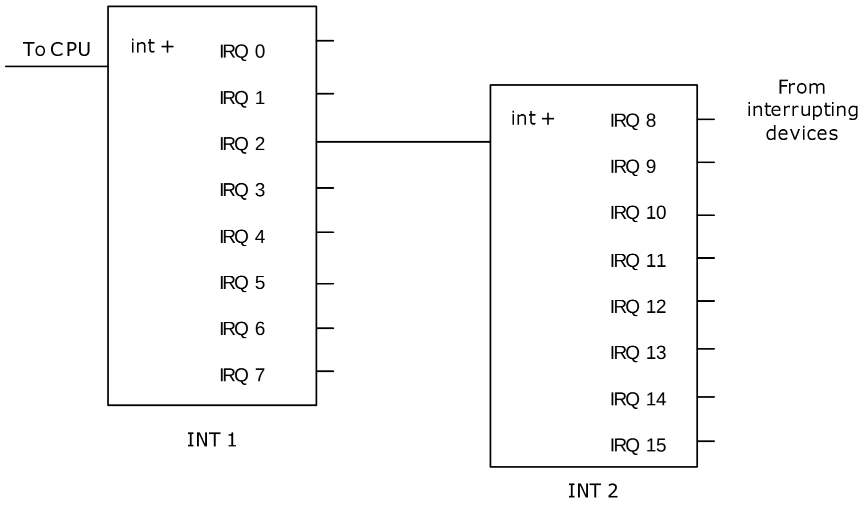


Figure 5-1: Intel 8259 Interrupt Controller Configuration

The interrupt controllers are programmed so that each IRQ input has a distinct interrupt priority. When an interrupt handler is servicing an interrupt, it may be preempted when another device asserts an interrupt with a higher priority.

The following table shows the IRQs, sorted by interrupt priority, with the most favorable priority at the top, and the least favorable at the bottom.

Table 5-2: Interrupt Priority

Interrupt Priority (1 = Highest)	IRQ
1	IRQ 1
(cascade)	IRQ 2
2	IRQ 8
3	IRQ 9
4	IRQ 10
5	IRQ 11
6	IRQ 12
7	IRQ 13
8	IRQ 14
9	IRQ 15
10	IRQ 3
11	IRQ 4
12	IRQ 5
13	IRQ 6
14	IRQ 7
15	IRQ 0

On all Board Support Packages (BSPs), IRQ 1 is used for the keyboard and has the most favorable priority. IRQ 0 is used for the real-time clock and has the least favorable priority. This means that the clock interrupt handler could be preempted by any other interrupt, whereas the keyboard handler cannot.

IRQ 2 is used to cascade interrupts from Controller INT2 to Controller INT1. The use of the other IRQs may vary for different BSPs.

The assignment of priorities to IRQs is made by the kernel code at boot time. The assignments are found in the file `/sys/bsp.xxx/bsp.intr.c`. In some BSPs, the assignment can be configured at kernel build time.

Implementing an Interrupt Handler

A typical approach to the organizational structure of a device driver that contains an interrupt handler is to divide the code into two components called *process context functions* and *kernel context functions* (see figure below).

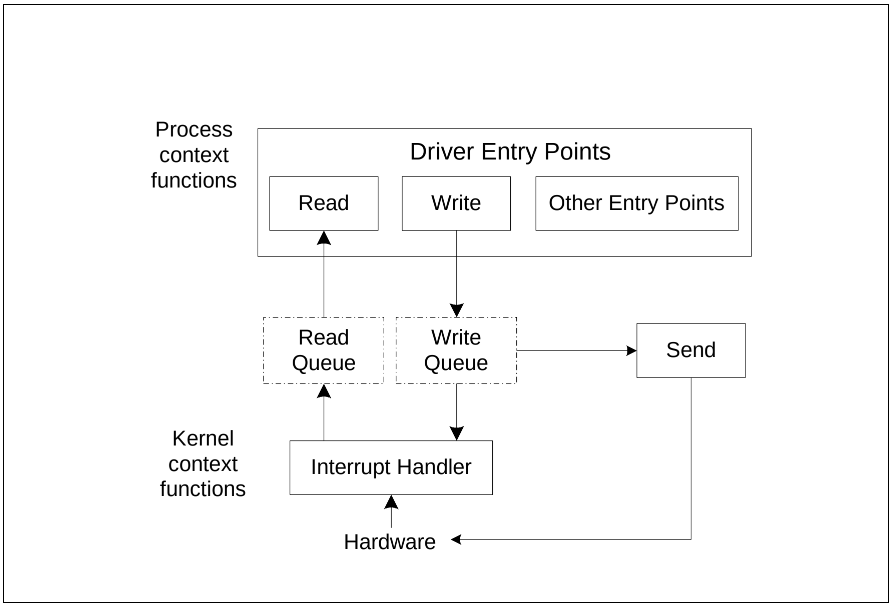


Figure 5-2: Top/Bottom Model for Device Drivers

Process context functions are the driver entry point functions and other supporting code, and the kernel context functions are the interrupt handler, and its subroutine `send`. Between the two halves are shared data structures (read and write queues in this example) internal to the device driver.

The `send` routine is a device driver supporting function used to send data to the hardware. (See “send() Routine” on page 92 for an example.)

Use of Queues

Queues are often used to communicate between the process context and kernel context functions. Examples of the use of queues to communicate between entry points and interrupt handlers are:

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

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

- For communicating from 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.

Following is an example code structure.

```
dev_read()
{
  swait(&receive_data_available,SEM_SIGIGNORE);
  disable();
  dequeue_receive_data();
  restore();
}

dev_write()
{
  disable();
  swait(&space_on_queue_available,SEM_SIGIGNORE);
  enqueue_send_data();
  if (no_interrupt_pending)
  output_data();
  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;
  }
}
```

Interrupt Handler Considerations

Following are programming considerations to use when creating an interrupt handler.

- Use `disable()` and `restore()` in the entry point functions and their subroutines to prevent interrupts from accessing data structures that are being modified.
- Application virtual addresses cannot be directly accessed from the interrupt handler.
- Translate application virtual addresses to kernel addresses in the entry point functions prior to making them available to the interrupt routines.

Example Code

Following is an example of an interrupt-based printer device driver.

Device Information Definition

```
/* ptrinfo.h */
struct ptrinfo
{
  int port;
  int irq;
  int qlen;
};
```

The device information definition has three variables associated with the interrupting driver. The `port` variable is the port number for the printer; `irq` is the interrupt line on which the printer interrupts the system; and `qlen` is the length of the queue used to communicate between the top and bottom halves of the driver.

Device Information Declaration

```
/* ptrinfo.c */
#include "ptrinfo.h"
struct ptrinfo ptrinfo0 =
{
  /* port */ 0x378,
  /* irq  */ 7,
  /* qlen */ 100
};
main()
{
  write(1, &ptrinfo0, sizeof(struct ptrinfo));
}
```

The device information declaration for the device information definition gives the port address for the printer port, which is 0x378. The IRQ line on which the printer interrupts is 7, and the queue length is 100. The program `ptrinfo.c` is compiled and executed to create a data file to be passed to the `install` routine during dynamic installation. (See Chapter 8, “Installation and Debugging.” for more on dynamic installation.)

Declaration for ioctl

```
/* ptrioctl.h */

#define PTRSTATUS 500

struct ptrstatus {
    int chars;                /* characters printed */
    int lines;                /* lines printed */
};
```

The `ioctl()` routine in this driver returns the number of characters and lines printed out so far. Thus, the user can issue the `ioctl` system call with the appropriate command and pointer to the structure defined above.

Driver Source Code

```
/* ptrdrvr.c */

#include <kernel.h>
#include <mem.h>
#include <file.h>
#include <errno.h>
#include <ioctl.h>
#include "ptrinfo.h"
#include "ptrioctl.h"

/* ports */
#define PP_DATA      0       /* data port offset */
#define PP_STATUS    1       /* status port offset */
#define PP_CONTROL   2       /* control port offset */

/* status bits */
#define PP_BUSY      0x80    /* printer busy */
#define PP_PE        0x20    /* out of paper */
#define PP_SLCT      0x10    /* printer is selected  */
#define PP_ERROR     0x08    /* printer detected error */

/* control bits */
#define PP_IENABLE   0x10    /* interrupt enable  */
#define PP_SLCTIN    0x08    /* select printer */
#define PP_INIT      0x04    /* start printer */
#define PP_AUTOLF    0x02    /* auto line feed */
#define PP_STROBE    0x01    /* strobe printer */

#define port_in(addr)        __inb             /* copy 1 byte from port */
#define port_out(data,addr)  __outb(addr,data) /* copy 1 byte to port */
```

```
typedef unsigned short ptype;
```

Statics Structure

```
struct ptrstatics {
    ptype datap;       /* data port address   */
    ptype controlp;    /* cntrl port address  */
    char control;      /* control bits        */
    int irq;           /* IRQ number          */
    int closing;       /* closing device      */
    int close_sem;     /* semaphore for close */
    int expecting;     /* expecting an int.?  */
    int nextnl;        /* output a '\r' next? */
    int chars;         /* printed since open  */
    int lines;         /* printed since open  */
    int qlen;          /* characters in queue */
    char *q;           /* base queue address  */
    int head;          /* head of queue       */
    int tail;          /* tail of queue       */
    int qdata;         /* data in the queue   */
    int free_sem;      /* free queue space    */
};
```

The statics structure for the interrupt handling printer driver is shown above. The IRQ (`irq`) number, the queue length (`qlen`) and port address (`dport`) are copied from the device information definition.

`chars` and `lines` store the number of characters and lines printed out so far. **q** is the base address of the queue. `head` and `tail` keep track of data in the queue.

`close_sem` is a semaphore used for ensuring that the output queue is fully drained before the device is closed. `expecting` is initially reset to indicate that the first character has to be output before an interrupt is received by the system. `nextnl` is used for mapping `\n` (newline) to `\r\n` (carriage return/line feed).

install() Entry Point

```
#define PERR (struct ptrstatics *) SYSERR

char *dev_install(info)
struct ptrinfo *info;
{
    struct ptrstatics *s;
    extern void ptrint();
    int i;

                           /* probe for the printer */
    port_out(1, info->port+PP_DATA);
    if (port_in(info->port+PP_DATA) != 1)
{
        return (char *) PERR;
    }

    if (!(s = (struct ptrstatics *)sysbrk ((long)sizeof *s)))
        return (char *) PERR;
    if (!(s->q = sysbrk((long)info->qlen)))
{
        sysfree(s, (long)sizeof *s);
        return (char *) SYSERR;
    }

                              /* initialize statics */
    s->datap = info->port + PP_DATA;
    s->controlp = info->port + PP_CONTROL;
    s->control = PP_SLCTIN | PP_INIT;
    s->irq = info->irq;
    s->expecting = 0;
    s->closing = s->close_sem = 0;
    s->nextnl = 0;
    s->free_sem = s->qlen = info->qlen;
    s->qdata = s->head = s->tail = 0;
    s->lines = s->chars = 0;

                             /* initialize printer */
    iointset(32+s->irq, ptrint, s);
    port_out(PP_SLCTIN, s->controlp);
    for (i = 0; i < 100; i++) ;
    port_out(s->control, s->controlp);

    return (char *) s;
}
```

The `install()` entry point function checks for the existence of the printer. The data is written onto the port and read back immediately. If the data is not the same, then there is no printer. Once the presence of the printer is confirmed, the statics structure and the queue are allocated. The fields within the structure are initialized.

`free_sem` is initialized to the queue length and the `irq` number is copied into the statics data structure. The routine `iointset()` is called to initialize an interrupt handler for the given interrupt vector. (The offset of 32 is added to the `irq` number before passing it to the routine on the x86.) Finally, an initialization sequence is sent to the printer.

Notice that there is a timing loop between the two `port_out` calls. This is required for the transmitted data to be latched properly. The pointer to the statics structure is returned.

uninstall() Entry Point

```
void dev_uninstall(s)
struct ptrstatics *s;
{
    iointclr(32+s->irq);
    sysfree(s->q, (long)s->qlen);
    sysfree(s, (long)sizeof *s);
}
```

The `uninstall()` entry point clears the interrupt vector by using the `iointclr()` function. `uninstall()` frees the memory associated with the queue and the statics structure.

open() Entry Point

```
int dev_open(s, devno, f)
struct ptrstatics *s;
int devno;
struct file *f;
{
    if (f->access_mode & FREAD)
    {
        pseterr(EINVAL);
        return SYSERR;
    }
    return OK;
}
```

The `open()` entry point checks for the access mode of the device and returns an error if the application has tried to open it in read mode.

close() Entry Point

```
int dev_close(s, f)
struct ptrstatics *s;
struct file *f;
{
    int ps;

    disable(ps);
    if (s->expecting)
{
        s->closing = 1;
        restore(ps);
        swait(&s->close_sem, SEM_SIGIGNORE);
        s->closing = 0;
    } else
{
        restore(ps);
}
    s->lines = s->chars = 0;
    return OK;
}
```

The `close()` entry point function ensures that the characters to be output are complete before the device is closed. It checks whether `s->expecting` is 1. This indicates that there are characters present in the queue. If this is true, it sets the `s->closing` flag and waits in the `swait()` routine for the interrupt handler to signal that all characters are output and the device can be closed. It also resets the `chars` and `lines` fields in the statics structure definition.

send() Routine

```
/*  assumes:
**    data in the queue
**    interrupts disabled
*/
void send(s)
struct ptrstatics *s;
```

```
{
    char c;

    if (s->nextnl) {
        c = '\r';
        s->nextnl = 0;
        s->lines++;
    } else {
        c = s->q[s->head++];
        s->head %= s->qlen;
        s->qdata--;
        ssignal(&s->free_sem);
        s->nextnl = c == '\n';
        s->chars++;
    }
    port_out(c, s->datap);
    port_out(s->control | PP_STROBE, s->controlp);
    port_out(s->control | PP_IENABLE, s->controlp);
}
```

This routine outputs a character into the port by dequeuing from the queue. If a new line is found it puts out a `\r` and resets the `nextnl` flag. If not, it dequeues from the head of the queue, adjusts the head pointer to wrap around, and signals the semaphore while keeping track of the free space in the queue. This routine then increments the number of characters `s->chars`.

The routine then puts a character onto the printer port. The `while` loop to check the status port is no longer necessary because the interrupt signifies that it is safe to write. The character is just put onto the data port. After this, to latch the byte onto the printer, a high-low strobe is put onto the control port.

Interrupt() Handler

```
void ptrint(s)
struct ptrstatics *s;
{
    if (s->qdata || s->nextnl)
    {
        send(s);
    } else
      {
        s->expecting = 0;
        if (s->closing) ssignal(&s->close_sem);
                           /* disable ptr interrupts: */
        port_out(s->control, s->controlp);
      }
}
```

If the queue has data in it, or if a new line is indicated, the routine `send()` is called to output the character onto the port. If not, it indicates that the queue is empty and thus `s->expecting` is set to zero. Further, if the `closing` flag is set, it indicates that the `close()` entry point routine is in an `swait()` state. Thus, an `ssignal` routine is called to awaken the semaphore. Finally, an initialization sequence is sent to the control port.

write() Entry Point

```
int dev_write(s, f, buff, count)
struct ptrstatics *s;
struct file *f;
char *buff;
int count;
{
    int i = count, ps;
    while (i--)
    {
        swait(&s->free_sem, SEM_SIGRETRY);
        disable(ps);
        s->q[s->tail++] = *buff++;
        s->tail %= s->qlen;
        s->qdata++;
        if (!s->expecting)
        {
            send(s);
            s->expecting = 1;
        }
        restore(ps);
    }
    return count;
}
```

The `write()` entry point has a loop for `count` characters to be output onto the queue. The `swait()` inside the loop tracks the free space in the queue. If there is space in the queue a character is queued.

The `disable()` and `restore()` function calls provide protection for the critical region of code, which is used by the interrupt handler. The character is queued to the tail of the queue. The `qdata` variable (which keeps track of the number of characters in the queue) is incremented. If `expecting` is zero, the first character on the queue is sent to the port and `expecting` is set to one.

ioctl() Entry Point

```
int dev_ioctl(s, f, command, arg)
struct ptrstatics *s;
struct file *f;
int command;
char *arg;
{
    switch (command) {
        case PTRSTATUS:
            if (wbounds(arg) < sizeof(struct ptrstatus))
            {
                pseterr(EFAULT);
                return SYSERR;
            }
            ((struct ptrstatus *)arg)->chars = s->chars;
            ((struct ptrstatus *)arg)->lines = s->lines;
            break;

        case FIOPRIO:
        case FIOASYNC:
            break;

        default:
            pseterr(EINVAL);
            return SYSERR;
    }
    return OK;
}
```

The `ioctl()` entry point handles the case for `PTRSTATUS`. The user can invoke this from an application program to determine the number of characters actually output onto the port. `arg` is the pointer to the user buffer. The check by `wbounds()` confirms that the user pointer has writable memory allocated to it.

x86 IRQ Device Defaults

The following table is a list of devices with their interrupt (also called IRQ) levels, I/O addresses and additional default information. No two devices with the same IRQ can be configured into the LynxOS kernel at the same time..

Table 5-3: x86 Default Device Configuration

IRQ[1]	IOBASE[2]	DMA Channel[3]	Device	Comments
0	N/A	N/A	`timer`	
1	0x3B4 0x3D4 0x3C4 0x3C5 0x3CE 0x3CF		`atc`	Keyboard
2	N/A	N/A		Cascade
3	0x2F8	N/A	`com2`	
3	N/A	N/A	`atcinfo`	
4	0x3F8	N/A	`com1`	
5	0x3E8	N/A	`com3`	
5	0x240	N/A	`wd3e`	
5	0x3F0	N/A	`if_3c579`	Slot number 7 TP = 0, AUI = 1, BNC = 3 (default 3)
5	0x240	N/A	`if_3c509`	Slot number 0 TP = 0, AUI = 1, BNC = 3 (default 3)
5	0x240	N/A	`if_wd3e`	On board RAM address[4]: 0xcc000 16 bit access 1
6	0x3F2-0x3F7	N/A	`flopinfo`	8237 DMA controller
7	0x378	N/A	`ptrinfo`	

Table 5-3: x86 Default Device Configuration (Continued)

IRQ[1]	IOBASE[2]	DMA Channel[3]	Device	Comments
7	0x200	N/A	`pcxe`	On board RAMbase: 0xd0000
8		N/A	`rtclock`	Real-time clock
9		N/A	`atc`	Vertical sync. interrupt
9	0x3E0	N/A	`com4`	
10				Unused by default.
11	0x330-0x332	5	`a_scsiinfo.c` `sim1542_info`	bus_on[5]: 8 bus_off[6]: 40
11	0x330[7]	5	`sim_1742_info`	Edge/Level[8]: edge(1)
11	N/A	5	`sim2742_info`	EISA slot[9]: 2
12				Unused by default.
13				Unused by default.
14	0x1F1-0x1F7, 0x3F6-0x3F7		`hdinfo`	Defaults are hard coded in driver; primary controller.
15				Unused by default.
-	0x340	N/A	`sim1522_info`	PIO mode only
[10]	N/A	N/A	`sim2940_info`	No user configurable options.

1. IRQ: interrupt used by the device.
2. IOBASE: 1 or memory locations that are used to communicate with the device.
3. DMA channel: DMA channel used by the device
4. On board RAM address: Some boards have memory mapped to a particular range in the I/O space. This number is the start of such memory. The size of memory varies from card to card.
5. bus_on- micro-seconds a device is allowed to stay on the bus.
6. bus_off- micro-seconds a device stay off after being on the bus.
7. Only applies in 1542-mode.
8. Edge/Level- A type of interrupt; LynuxWorks does not support level interrupts.
9. EISA slot- A specific slot that a card must be in for the device to work.
10. PCI card; the interrupt is assigned by the system BIOS.

Timeout Handlers

A timeout handler can be set up using the timeout driver service call, `timeout()`.

The prototype for `timeout()` is:

```
int timeout(void (*handler)(), char *arg, int interval)
```

where:

`handler`	Specifies the function to call
`arg`	Specifies an argument for function handler
`interval`	Specifies a timeout interval (10 millisecond granularity)

The `timeout` causes the function `handler` to be called with one argument, `arg`, after `interval` has expired. `timeout()` returns a non-negative timeout ID if there is a timer available. This ID can be used to track or cancel the timeout.

A timeout can be canceled before the routine is called using `cancel_timeout()` and passing it the timeout ID. Care should be taken to cancel only a pending timeout. The check for a timeout expiration and cancelling timeout should be done atomically with interrupts disabled.

NOTE: `timeout()` and `cancel_timeout()` calls may be used in an interrupt routine, but should not be used in a device driver `install()` entry point routine.

Following is a basic timeout handler algorithm.

```
entry_point()
{
  int ps;
  sem = 0;
  /* start device operations */
  timeoutID = timeout(timeoutHandler, arg, 1);
  swait(&sem, -1);
  disable(ps);
  if (timeoutID)
  {
    cancel_timeout(timeoutID);
    timeoutID = 0;
  }
  restore(ps);
}

timeoutHandler(arg)
{
  /* do timeout processing */
  timeoutID = 0;
  ssignal(&sem);
}
```

```
ISR()
{
  ssignal(&sem);
}
```

LynxOS timeout handlers are called by the clock interrupt handler. Timeout routines are handled in the kernel using a delta queue to check for all expired timers. Because the timeout handler is called from the clock interrupt handler, it should not execute for a long period of time. If a lengthy timeout processing is needed inside the timeout handler routine, it is better to handle the timeout inside a kernel thread. (See Chapter 6, “Kernel Threads and Priority Tracking.” for more information on kernel threads.) The timeout handler can simply increment a variable indicating the number of timeouts accumulated and signal the kernel thread. The kernel thread wakes up and handles the more complicated processing associated with the timeout.

Using kernel threads ensures that the thread can call routines, which can use semaphores for mutual exclusion instead of interrupt disabling, thus improving real-time response. The thread can handle all accumulated timeouts and once this is completed it blocks on the semaphore.

NOTE: Expired timers are checked in the clock interrupt handler and timeout handlers are called from there. Thus, a timeout handler cannot block on a semaphore.

Following is an example timeout handler algorithm implemented with kernel threads.

```
entry_point()
{
  event_sem = 0;
  timeoutCt = 0;
  tid = timeout(timeoutHandler, arg, ticks);
}

timeoutHandler(arg)
{
  timeoutCt++;
  ssignal(&event_sem);
}

Thread()
{
  int ps;
  int touts;
  for (;;)
    {
      swait(&event_sem, -1);
      disable(ps);
      touts = timeoutCt;
      timeoutCt = 0;
      restore(ps);
      while (touts--)
        timeout_processing();
    }
}

timeout_processing()
{
    swait(&mutex_sem, -1);
    /* do timeout processing */
    ssignal(&mutex_sem);
}
```

CHAPTER 6

Kernel Threads and Priority Tracking

Kernel threads keep drivers from interfering with the real-time response of the overall system. LynxOS kernel threads are designed specifically to increase driver functionality while decreasing driver response time, task response time, and task completion time. This chapter covers the implementation of kernel threads within a device driver.

Device Drivers in LynxOS

Device drivers form an important part of any operating system, but even more so in a real-time operating system such as LynxOS. 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, 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 interrupt dispatcher (usually negligible).

Driver Response Time

The *driver 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 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 developer should be aware of all delays that 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.

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 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 servicing 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 signalling 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), 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 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. 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 that 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 once in the `install` or `open` entry point. The advantage of starting it in `open` is that, if the device is never opened, the driver doesn't use up kernel resources unnecessarily. However, 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 thread function specifies a C function to 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 the "Priority Tracking" on page 109. It is usual to create the thread with a priority of one.

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.

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 the particular device is used. For the purposes of illustration, two different driver designs are discussed.

- *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.
- *Multiple access*: Multiple user tasks are permitted to have the device open and make requests.

Exclusive Access

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

- The top-half entry point (`read`/`write`) starts the data transfer on the device, then blocks, waiting for I/O completion.
- The interrupt handler signals the kernel thread when the I/O completes.
- The kernel thread consists of an infinite `for` loop, which does the following:
    - Waits for work to do
    - Processes interrupt
    - Wakes up user tasks

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

Top-Half Entry Point

The `read`/`write` entry point code is not 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.

```
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 transfered */
}
```

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 wakes up the user task that is waiting for I/O completion. For best system real-time performance, the kernel thread should re-enable 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 only perform 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:

- 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.
- 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 preemptive, 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. The next figure illustrates the overall design.

A data structure containing variables for event synchronization, error status, etc., is used to describe each request. The pending request queue and list of free request headers are part of the statics structure. The interrupt handler code is the same as in exclusive access design.

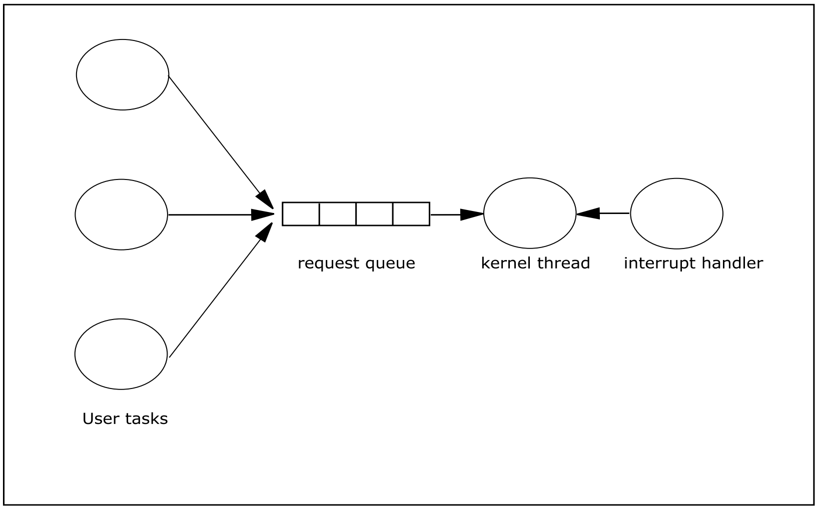


Figure 6-1: Interrupt Handler Design

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.

The next figure shows two possible scenarios in a system using a static thread priority. In both scenarios, Task A uses a device that generates work for the kernel thread. Other tasks with different priorities exist in the system. These are represented by Task B.

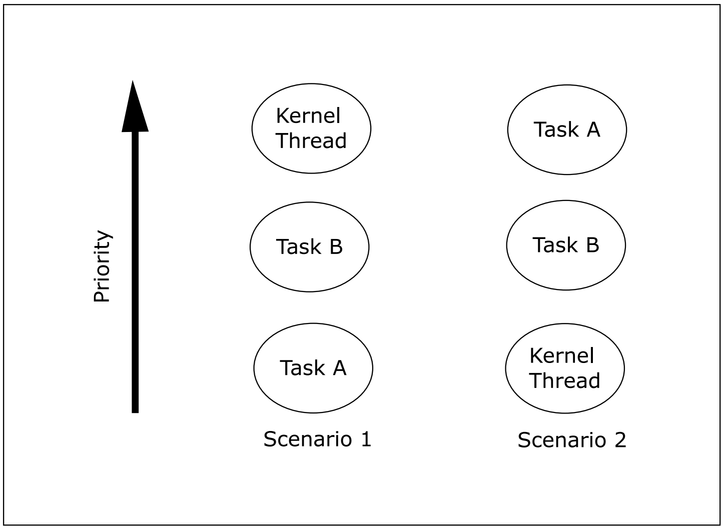


Figure 6-2: Scheduling with Static Thread Priorities

In the first scenario, Task B has a priority higher than Task A but lower than that of 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 allow 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 the figure below.

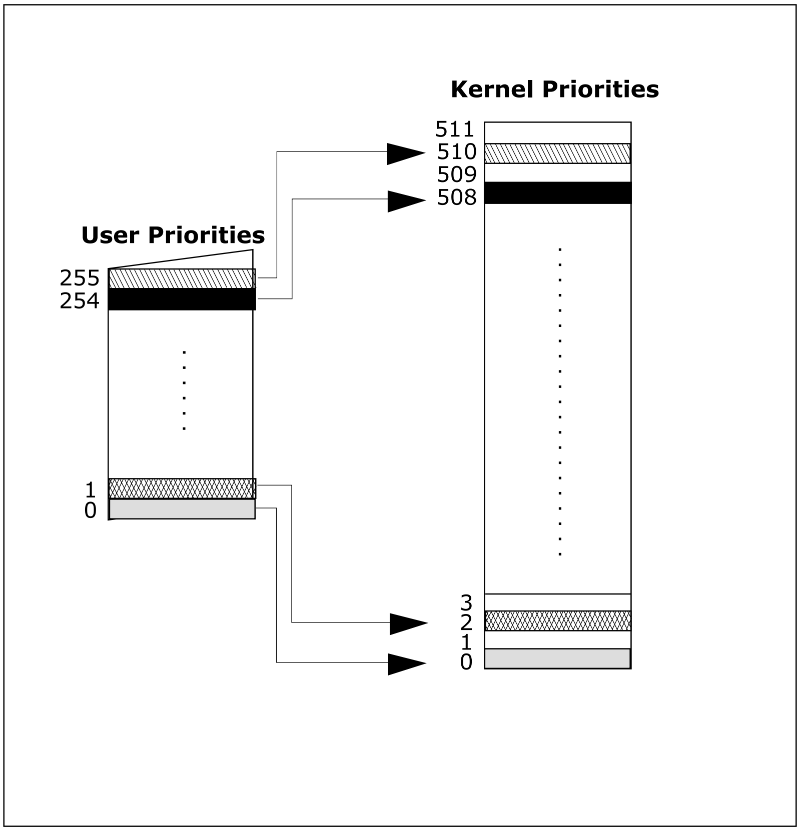


Figure 6-3: User and Kernel Priorities

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 following examples discuss 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 transfered */
}
```

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 the figure below.

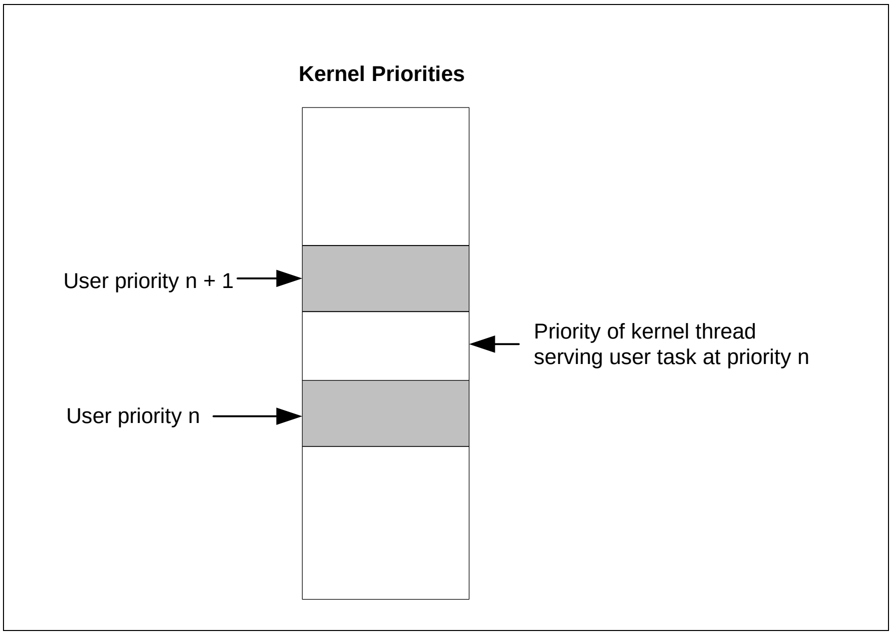


Figure 6-4: Kernel Thread Priorities

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 previously discussed, the driver maintains a queue of pending requests from a number of user tasks. These tasks probably have different priorities. Therefore, the driver must ensure that the kernel thread is always running at the priority of the highest priority user task that has a request pending. If the requests are queued in priority order this ensures 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 so, the driver must use a special data structure, `struct priotrack`, defined in `st.h`. Basically, the structure is a set of counters, one 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, 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);
            ...
        }
    }
```

Non-Atomic Requests

The previous examples implicitly assumed that requests made to the driver are handled atomically. That is to say, the device handles an arbitrary-size data transfer. This is not always the case. Many devices have a limit on the size of the 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 only transfer 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_addn`. 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 discussed concerning drivers that perform all interrupt processing in the interrupt handler is that in certain circumstances, a device can generate a series of interrupts in rapid succession. For many devices, the use of a kernel thread and priority tracking illustrated above resolves the problem.

Take, for example, a disk driver. The figure below 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 are completed, a higher priority task begins execution. But the 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.

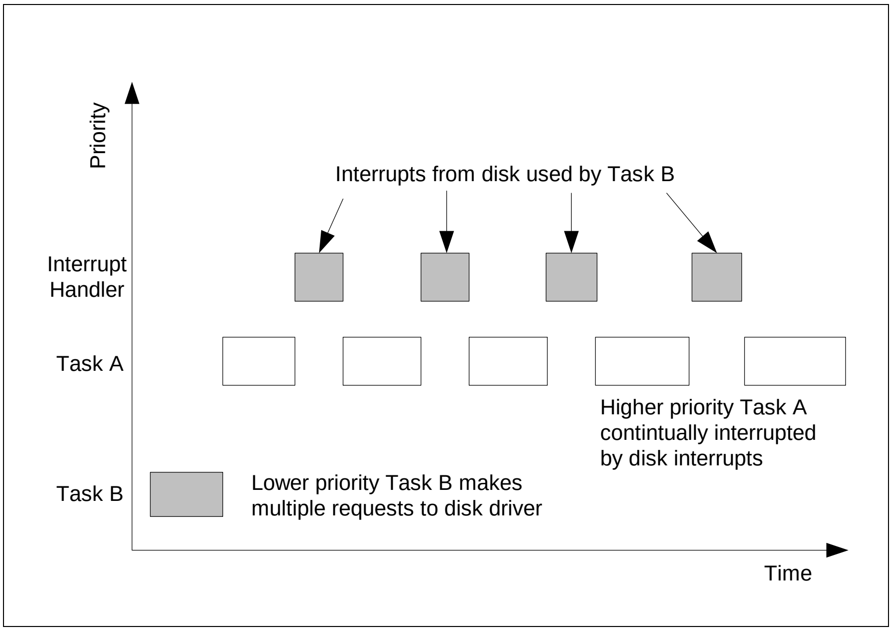

Priority
Interrupts from disk used by Task B
Interrupt Handler
Task A
Higher priority Task A contintually interrupted by disk interrupts
Task B
Lower priority Task B makes multiple requests to disk driver
Time

Figure 6-5: Interrupt Handling without Kernel Threads

The next figure shows the same scenario using kernel threads. The important thing to note is that the higher priority task can only be interrupted 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.

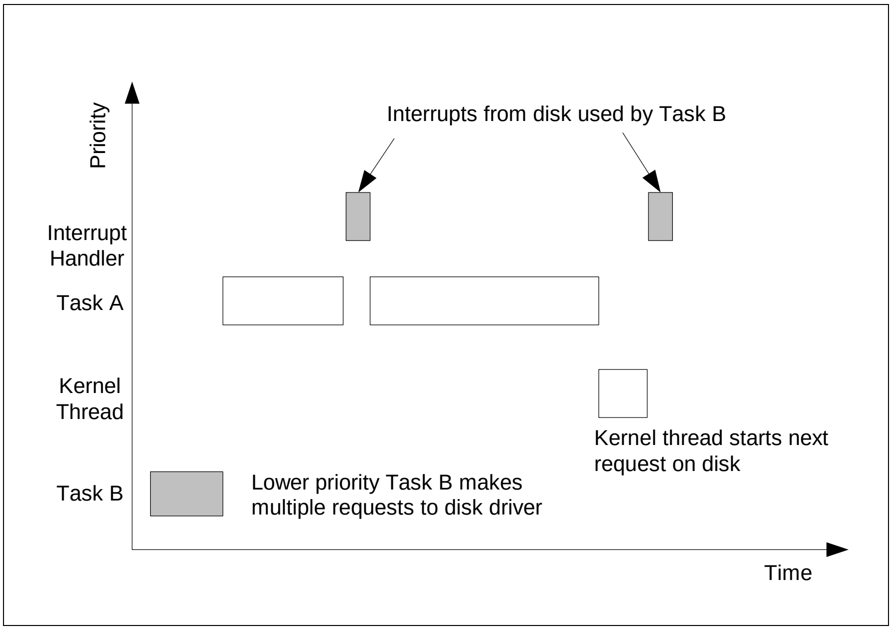

Priority
Interrupts from disk used by Task B
Interrupt Handler
Task A
Kernel Thread
Kernel thread starts next request on disk
Task B
Lower priority Task B makes multiple requests to disk driver
Time

Figure 6-6: 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. 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 re-enabled 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.

CHAPTER 7 Network Device Drivers

A network driver is defined as a link-level device driver that interfaces with the LynxOS TCP/IP module. Unlike other drivers, a network driver does not interface directly with user applications. It interfaces instead with the LynxOS 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 networking software, not only within the drivers, but also as part of the TCP/IP module.

The example code below illustrates these points 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 its use is further illustrated.

A driver must include the following header files, which define these structures and various symbolic constants 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

This 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

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

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. `struct ifnet` is defined in `/usr/include/bsd/if_var.h`. 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 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, tokenring etc */
    u_char ifi_addrlen;    /* media address length */
    u_char ifi_hdrlen;      /* 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;     /* ouput 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`, and `IFF_BROADCAST` 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, explained in “ioctl Entry Point” on page 134.

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 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 size 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. For example:

```
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; LynxOS does not currently support such protocols.

Data is removed from the head or tail of an mbuf by simply incrementing the m_data pointer 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 or MGET specifies whether the function should block or return an error when no mbufs are available. A driver should use M_DONTWAIT, which causes the mbuf pointer to be set to 0 if no mbufs are free.

The third argument to m_get or 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, there is benefit to be gained from using a cluster if the packet is larger than `MCLBYTES/2`.

Freeing mbufs

Because there is a limited number of `mbufs` in the system, the driver must take care to free `mbufs` at appropriate points. These are listed below:

Packet Output:

- Interface is down
- Address family not supported
- No `mbufs` for Ethernet header
- `if_snd` queue is full
- After packet has been transferred to interface

Packet Input:

- Not enough `mbufs` to receive packet
- Unknown Ethernet packet type
- Input queue is full

The sections “Packet Input” and “Packet 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 7-1: 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, the `mbuf` successor (pointed to by `m->m_next`) is stored in the second argument.
`MGETHDR`	Allocate an `mbuf` and initialize it as a packet header.

Table 7-1: Summary of Commonly Used mbuf Macros (Continued)

Macro	Description
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.
dtom	Convert the data pointer within mbuf to mbuf pointer.
mtod	Convert mbuf pointer to data pointer of specified type.

NOTE: MCLGET, MFREE, MGETHDR, MH_ALIGN, and M_PREPEND are protected by network semaphore lock.

Table 7-2: Summary of Commonly Used mbuf Functions

Function	Description
m_adj	Remove data from mbuf at start or 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 functional version of MGET macro.
m_getclr	Get an mbuf and clear the buffer.
m_gethdr	A function version of the MGETHDR macro.
m_pullup	Pull up data so that a 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 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 situation 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`, or `IF_DROP`.

Driver Entry Points

A network driver contains the following entry points:

Table 7-3: Network Driver Entry Points

Entry Point	Description
`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.
`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, etc.), 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/lynx.os` 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: This method only works for statically installed drivers. Dynamically installed drivers do not have an entry in the `ucdevsw` table.

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 appropriate network interface attach routine, for example `ether_ifattach()` for Ethernet drivers.

```
struct statics *s;
struct ifnet *ifp;

ifp->if_timer = 0 ;
ifp->p = (char *) s;/* address of statics structure */
ifp->if_unit = 0;
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;

ether_ifattach (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 when 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 `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 `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 normally has to test whether the transmitter is idle and ready to accept a new packet. It typically dequeues a packet (which is enqueued by `ether_output`) and transmits it.

```
if (ds->xmt_pending) {
/* if already one transmission is in progress */
  return 1;
}
IF_DEQUEUE(&ifp->if_snd, m);
if (m == 0) {
  return 0;
}
ds->xmt_pending = 1;

    ...
    ...
/* Initiate transmission if using Berkeley packet filter */
if (ifp->if_bpf)
  bpf_mtup(m)
return 0;
```

One important point to consider is that the start routine can now be called by the TCP/IP module (by way of `ether_output`) and the driver stream task 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. For example:

```
ssignal(&s->thread_sem);
return (0);
```

Also note that the total data available in an `mbuf` can be obtained from the `mbuf` packet header. For example:

```
/* 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. A collision is not necessarily an error condition. An interface normally makes 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 re-enabled by the driver’s kernel thread. The processing of input packets involves the following activities:

- Determining packet type

- Copying data from interface into `mbufs`
- Stripping off Ethernet header
- Enqueueing packet on input queue
- Re-enabling receiver interrupts
- Maintaining 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 packet received. In the following code, 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, which 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`, 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 can not 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 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 in the following syntax:

```
drvr_ioctl (ifp, cmd, arg)
struct ifnet *ifp;
int 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 allows 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 brought up, the driver should call the `drvr_init` function to initialize the interface. When the interface is 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 that 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 non-zero value to cause the watchdog function to be called again. Setting the `if_timer` value to 0 disables 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 purposes. The thread should handle interrupts first and then the packets on the output queue. The general structure of the thread looks something like:

```
struct statics *s;

for (;;) {
  swait (&s->threadsem, SEM_SIGIGNORE);
  handle_interrupts (s);/* handle any interrupts */
  output_packets (s);
  /* start tranmitter if necessary */
}
```

The precise details of the thread code depend on the hardware architecture. The function for processing interrupts contains the packet input code discussed above. It also maintains the various statistics counters. Also, receiver interrupts, if disabled by the interrupt handler, are re-enabled at this point. The output function performs the tasks discussed above in the “Packet Output” section.

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 that the kernel thread should be set to. For example:

```
drvrsetprio (ifp, prio)
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/lynx.os` 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:

```
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. For example:

```
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.

- If the interface supports IP multicasting, the `install` routine should set the `IFF_MULTICAST` flag. For example:

  ```
  ifp->if_flags = IFF_BROADCAST | IFF_MULTICAST;
  ```

  `ifp` is a pointer to the interface `ifnet` structure.

- Two new `ioctls` 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. For example:

  ```
  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;
  }
  ```

- 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 a 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);
}
```

- 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`. For example:

```
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);
```

CHAPTER 8

Installation and Debugging

This chapter discusses the two methods of device driver installation in LynxOS: static and dynamic.

Static Versus Dynamic Installation

This section provides a comparison the two methods of device driver installation to assist the developer in choosing the type of installation 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 manner can be removed; however its text and data segments remain within the body of the kernel.

The advantages of static installation are:

- Devices are instantly available upon 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, it is recommended to use dynamic installation during the development of a new driver. The device driver can be installed statically after it has been fully tested.

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.

The advantages of dynamic installation are as follows:

- Dynamic installation is useful when the device driver is being written. The ease of installation and uninstallation makes it ideal for faster development and debugging.
- More than one driver can be used for the same device. If there is a need to use two drivers for the same device, they can be installed according to system needs.
- Memory is not wasted on seldom-used drivers. They are allocated only when needed.

Static Installation Procedure

The code organization for static installation is shown in the table below.

Table 8-1: Code Organization for Static Installation

Directory	File	Description
`/`	`lynx.os`	LynxOS kernel
`/sys/lib`	`libdrivers.a`	Drivers object code library
	`libdevices.a`	Device information declarations
`/sys/dheaders`	`devinfo.h`	Device information definition
`/sys/devices`	`devinfo.c`	Device configuration file
	`Makefile`	Instructions for making `devlib.a`
`/sys/drivers/drvr`	driver source	The source code for driver `drvr` to be installed
`/sys/lynx.os`	`CONFIG.TBL`	Master device and driver configuration file.
	`Makefile`	Instructions for making `/lynx.os`
`/etc`	`nodetab`	Device nodes
`/sys/cfg`	`driv.cfg`	Configuration file for `driv` driver and its devices

The following steps describe how to implement 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 `devinfo.c` file and update the `/sys/lib/libdevices.a` library file to include `devinfo.o`. This may also be automated by adding `devinfo.c` to the `Makefile`. For example:

   ```
   DEVICE_FILES=atcinfo.x dtinfo.x flopinfo.x devinfo.x
   ```
4. To update `/sys/lib/libdevices.a`, enter:

   ```
   make install
   ```

Driver Source Code

Assuming the new driver is called `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/lynx.os` directory.

The `CONFIG.TBL` file is used with the `config` utility to produce driver and device configuration tables for LynxOS. 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 for use with the `mknod` utility.

Configuration File: CONFIG.TBL

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

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

The special characters in the configuration file are

Table 8-2: Special Characters

Character	Description
`#`	Indicates a comment in the configuration file. The rest of the line is ignored when this is the first character in any line.
`\`	The continuation character to continue a line even within a comment
`:`	If the `:` is the first character in the line, it is ignored.
`I:`*`filename`*	Indicates 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
N:minor_device2:minor_number
```

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 kernel, type the following commands:

```
cd /sys/lynx.os
make install
```

For the applications programs to use a device, a node must be created in the file system with **`mknod`**. This can be done automatically by using the `nodetab` file created by **`config`**.

When the system is rebooted to use the newly-created operating system, the **`reboot`** command should be given the **`N`** flag:

```
reboot -aN
```

The **`N`** flag instructs **`init`** to run **`mknod`** and create all the nodes mentioned in the new `nodetab`.

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. The installation of the dynamically-loaded device driver need not be done manually. A shell script can be written or a C program can be used to install the device driver 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>
static struct dldd entry_points = { open, close, read
    write, select, ioctl, install, uninstall, 0}
```

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

```
static struct dldd block_entry_points =
 { b_open, b_close, b_strategy, ionull, ionull, b_ioctl,    b_install,
   b_uninstall, 0}
```

The include 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 was used for STREAMS drivers, which are no longer supported by LynxOS. STREAMS functionality can be replicated with `mmap()`. See the `mmap()` man page for details.

NOTE: On the PowerPC platform, the dldd structure 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 ();
}
nullread(s, f, buff, count)
char *s;
struct file *f;
char *buff;
register int count;
{
    return 0;
}
nullwrite(s, f, buff, count)
char *s;
struct file *f;
char *buff;
register int count;
{
    return (count);
}
nullioctl()
{
    pseterr (EINVAL);
    return (SYSERR);
}
```

```
nullselect()
{
    return (SYSERR);
}
int nullinstall()
{
    return (0);
}
int nulluninstall()
{
    return (OK);
}
static struct dldd entry_points = {
  ionull,
  ionull,
  nullread,
  nullwrite,
  nullselect,
  nullioctl,
  nullinstall,
  nulluninstall,
  (kaddr_t *) 0
};
```

Note that calls to a driver entry point replaced by `ionull` 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 point functions in the above example. This causes 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, follow these recommendations for compiling and installing dynamic device drivers on a specific LynxOS platform.

x86

LynxOS supports dynamic driver installation with the GNU C compiler. To compile a dynamically-loadable device driver, the command to be used depends on whether you are compiling on a Lynx OSa native development system, or on a cross development host:

On a LynxOS native development system:

```
/usr/i386-coff-lynxos/usr/bin/gcc -c -o driver.obj\
 driver.c -I/sys/include/kernel\
 -I/sys/include/family/x86  -D__LYNXOS
```

On a LynxOS cross development host:

```
$(ENV_PREFIX)/cdk/platform-coff-x86/usr/bin/gcc -c -o\
 driver.obj driver.c -I/sys/include/kernel\
 -I/sys/include/family/x86 -D__LYNXOS
```

where `platform` is

`sunos`	For SunOS targets
`win32`	For Windows targets
`hpux`	For HP/UX targets
`linux`	For Linux targets

NOTE: The current release of LynxOS does not support `a.out` format dynamic drivers.

PowerPC

LynxOS for PowerPC supports dynamic driver installation with the GNU C compiler but requires a special import file for the linker. The *import file* contains a list of driver service calls used by the device driver. An example is shown in the steps below.

1. Compile the driver with GNU C. The command for compiling the driver depends on whether the target is a native target, or if it is an AIX, SunOS, Windows, HP/UX, or Linux cross target.

   A)For LynxOS native development systems:

   ```
   /usr/ppc-xcoff-lynxos/usr/bin/gcc -c -o driver.o\
    driver.c -I/sys/include/kernel\
    -I/sys/include/family/ppc -D__LYNXOS
   ```

   B)On cross development systems:

```
$(ENV_PREFIX)/cdk/platform-xcoff-ppc/usr/bin/gcc\
 -c -o driver.o driver.c -I/sys/include/kernel\
 -I/sys/include/family/ppc -D__LYNXOS
```

where *platform* is

`sunos` For SunOS targets

`win32` For Windows targets

`hpux` For HP/UX targets

`linux` For Linux targets

2. Create an import file. For example, the file `driver.import` contains:

```
sysbrk
iointset
fclear
iointclr
get1page
free1page
```

3. Now create a dynamically-loadable object module using **ld**:

   - On a LynxOS native development system:

   ```
   ld -bM:SRE -bimport:driver.import -o driver.obj\
    driver.o
   ```

   - On a cross development host:

   ```
   $(ENV_PREFIX)/cdk/platform-xcoff-ppc/usr/bin/ld\
    -bM:SRE -bimport:driver.import -o driver.obj\
    driver.o
   ```

   The option `-BM:SRE` tells the linker this is a shared reusable module. The option `-bimport` tells the linker the name of the import file. *platform* is the same as described in Step 1.

This will create an object module that has a loader section for dynamic linking and stub functions for kernel callbacks.

All Platforms

Once a dynamic driver object module has been created, this object module can now be dynamically installed.

For character device drivers, enter:

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

For block device drivers, enter:

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

If successful, `drinstall` or `dr_install` returns the unique `driver-id` that is defined internally by the LynxOS 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 `drinstall()`.

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 a program has to be written to instantiate the device information definition.

Assuming the device information definition appears as:

```
struct my_device_info {
    int address;
    int interrupt_vector;
};

/* myprogram.c */

struct my_device_info devinfo = { 0xd000, 4 };

main()
{
    write(1, &devinfo, sizeof(struct \
        my_device_info));
}
```

This program can be compiled and executed while redirecting the output to a file. When compiling the program, use the default ELF-based compiler.

On a LynxOS native development system:

```
gcc -o myprogram myprogram.c
```

On a cross development host:

```
$(ENV_PREFIX)/bin/gcc -o myprogram myprogram.c
```

Then run the program on the target computer. Redirect standard output to a file.

```
./myprogram > mydevice_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` system calls. For example:

```
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 system call. This installs the appropriate device with the corresponding driver and assigns a major device number to it (in this case, we assume this is `major_no`).

Node Creation

Unlike the static installation, there is no feature to automatically generate the nodes under dynamic installation. This should be done manually using the **`mknod`** command. (See the *LynxOS User's Guide*.)

By convention, the node is typically created in the `/dev 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/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

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 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. 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.

- 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

This section describes some of the techniques and mechanisms available to assist with the debugging process.

Communicating with the Device Driver

Because device drivers are not attached to a particular control terminal, ordinary `printf()` statements do not work. LynxOS provides the device driver service routines `kkprintf()` and `cprintf()` to assist in debugging. Both have the same syntax as the `printf()` system call.

The `kkprintf()` routine always outputs to the debug terminal. The `kkprintf()` routine can be used in interrupt routines, however, with caution. If the operating system configuration includes a device that uses the hardware used by `kkprintf()`, programs that access the device may hang the system.

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

Following are some tips for using `kkprintf()` and `cprintf()`.

- The debug terminal is configured as `COM2` on the x86 platform and `TTY0` on the PowerPC. Currently, the debug terminals are not configurable.
- Insert `kkprintf()` statements in the `install()` entry routine of the new driver. After the `devinstall` command is executed, the `install()` entry point is invoked. This is a good way of identifying if the `install()` entry point is invoked, and if the device information declaration passed is received properly.
- Insert `kkprintf()` statements in the `uninstall()` entry point. After every uninstall the `uninstall()` entry point is invoked and the `kkprintf()` statements should be seen.
- Initially, it is advisable to put a `kkprintf()` statement at the beginning of every entry point to make sure it is invoked properly. Once an entry point function is working properly the `kkprintf()` statements can be removed.
- Using `kprintf()` 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 (see below) to investigate this area in memory.

Simple Kernel Debugger (SKDB)

The Simple Kernel Debugger 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. A serial connection to a second machine running kermit to capture debugging output from SKDB is also possible. See the *LynxOS Total/db Guide* for more information on SKDB.

Handling Bus Errors

A bus error occurs when access to an invalid address is made. The function `recoset()` can be used to change the default system behavior when a bus error occurs. By default, the bus fault handler calls `panic()`, which displays a message that a serious problem has occurred and attempts to shut down the system.

Before attempting a process that may cause a bus error, use `recoset()`. If a bus error occurs, program execution continues as if returning from `recoset()` with a non-zero return value. A branch to the code to handle the error condition can then be made. Restore the default bus error handler with `noreco()`.

For example:

```
...
p = <some device address>
if (!recoset()) /* setup recoset() and establish branch back to point */
{
  x = *p /* possible bus error, if device not present */
}
else
{
/* code to handle bus error, if it occurs */
}
noreco();
...
```

In the example, if a bus error occurs at `x = *p`, program execution changes with a branch back to `if (!recoset())`. At this point the **`if`** condition is evaluated as if `recoset()` returned a negative value, and program flow continues with the code to handle the error.

NOTE: The `install()` entry point function is protected from crashing if a bus error occurs and pointers passed to the `read()` and `write()` entry points are validated by the OS so bus errors will not occur. Pointers passed to the `ioctl()` entry point function must be checked with `rbounds()` or `wbounds()`.

Probing for Devices

It is very common for the device driver to test for the presence of a device during the `install()` entry point. For this reason, LynxOS handles bus errors during execution of the `install` routine, thus relieving 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 is taken to mean that the device is not present and user tasks will not be permitted to open it.

Additional Notes

- Statically installed device drivers in LynxOS can also be uninstalled dynamically. However, the memory reclamation of the TEXT section is not done in this case.
- Symbols from two or more dynamically-loaded device drivers cannot be resolved. 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. Thus, any memory that is dynamically allocated must be freed before uninstalling the device driver.

CHAPTER 9

Device Resource Manager (DRM)

The Device Resource Manager (DRM) is a LynxOS 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.

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
- Using DRM Facilities from Applications
- Advanced Topics
- PCI Bus Layer
- Example Driver
- Sample Application

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 the figure below. To support Hot Swap environments, DRM nodes are inserted and removed from the device tree, mimicking Hot Swap insertion and extraction of system devices.

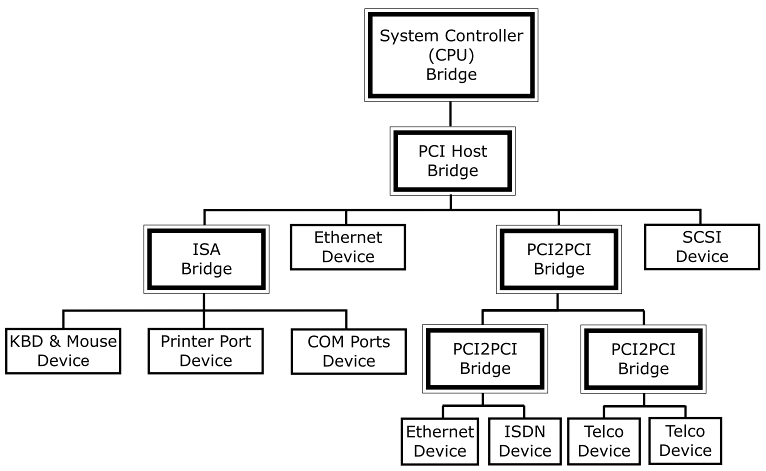


Figure 9-1: Device Tree

DRM Components

A module view of DRM and related components is shown in the following figure. A brief description of each module is given below the figure.

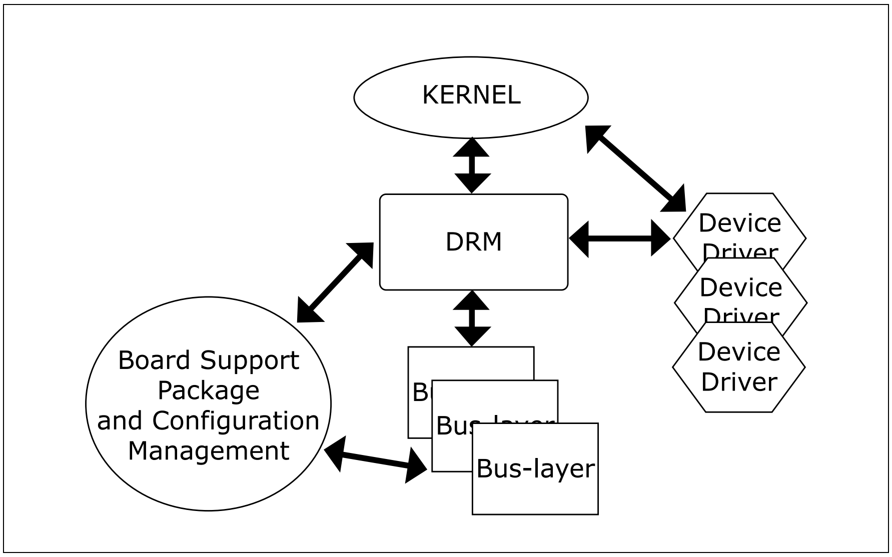


Figure 9-2: Module View

- **DRM** - DRM provides device drivers with a generalized device management interface.
- **KERNEL** - The LynxOS 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. 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 supports only one bus layer, which is used for managing PCI and CompactPCI 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.

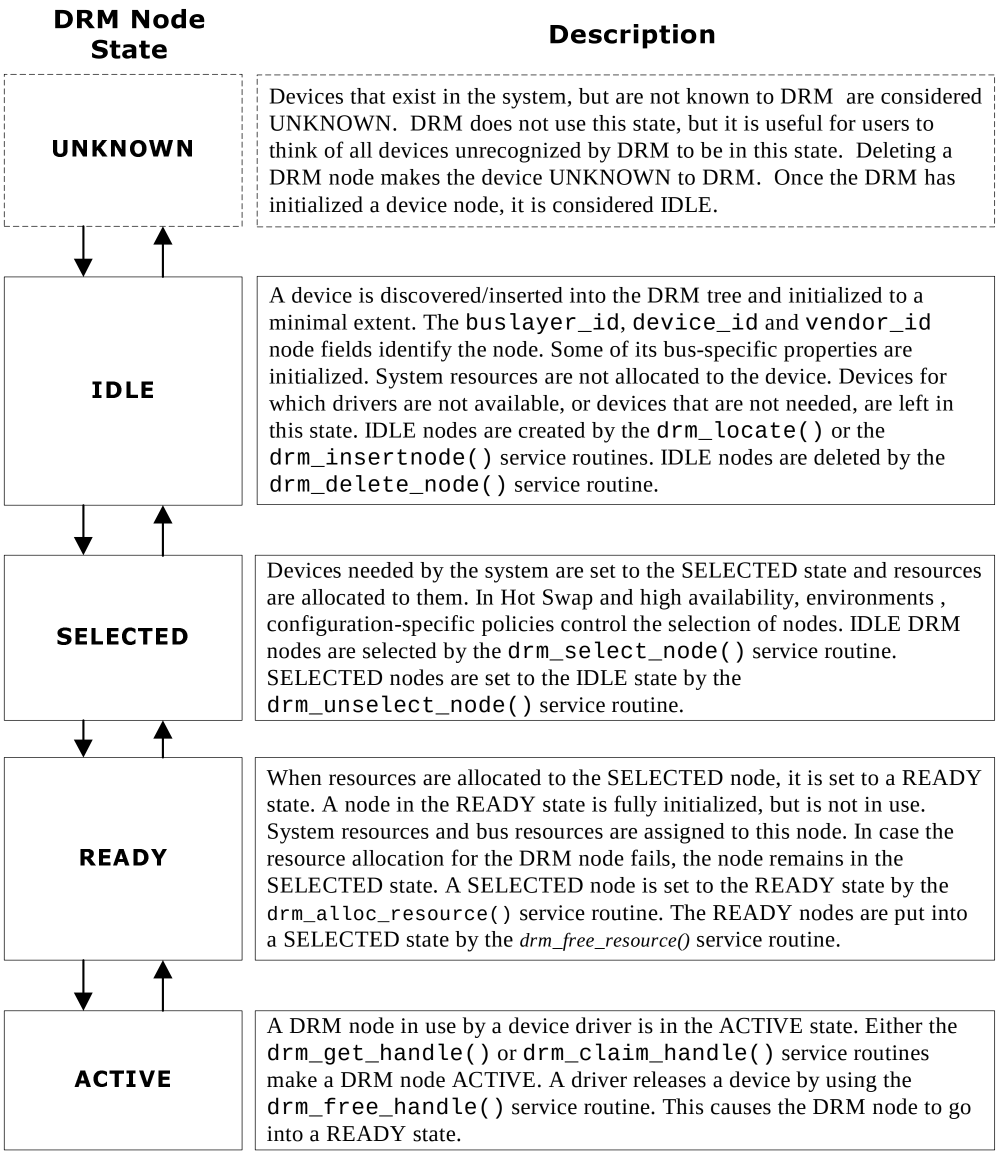
NOTE: Subsequent coding examples in this chapter make reference to a data structure of type `drm_node_s`. This structure is a data item used internally by the LynxOS kernel as the software representation of a DRM node and is not intended to be accessed at the driver or user level. LynxOS does not export a definition of this structure. The coding examples use opaque pointers, which are passed around and are not meant to be dereferenced.

Table 9-1: DRM Node Fields

Field Name	Description
`vendor_id`	This field is used for device vendor identification.
`device_id`	This field identifies the DRM node.
`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 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 the state 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 a READY state. A node in the READY state is fully initialized, but is 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 `drm_alloc_resource()` service routine. 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 9-3: DRM Node States

DRM Initialization

The DRM module is initialized during LynxOS 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.

LynxOS provides the ability to control PCI resource allocation. PCI resources can be allocated either by the BIOS or by DRM. By default, LynxOS x86 distributions use the BIOS to handle resource allocation. For other platforms, DRM handles the resource allocation. Because DRM uses the same set of interfaces, whether or not it handles resource allocation, device drivers do not need to change.

For more information on PCI resource allocation and DRM, see the chapter “PCI Resource Allocator for LynxOS” in the *LynxOS User’s Guide*.

DRM Service Routines

DRM service routines are used by device drivers to identify, setup 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 177.

Table 9-2: 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.	All Drivers
`drm_unregister_isr`	Clears an interrupt service routine.	All Drivers
`drm_map_resource`	Creates an address translation for a device resource.	All Drivers

Table 9-2: Summary of DRM Services (Continued)

Service	Description	Usage
`drm_unmap_resource`	Removes an address translation for a device resource.	All Drivers
`drm_device_read`	Performs a `read` on a device resource.	All Drivers
`drm_device_write`	Performs a `write` on a device resource.	All Drivers
`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

Table 9-2: Summary of DRM Services (Continued)

Service	Description	Usage
`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
`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 man pages for a complete description.

Table 9-3: DRM Service Routine Interface Specification

Name	Synopsis
`drm_locate()`	`int drm_locate(struct drm_node_s *handle)`
`drm_insertnode()`	`int drm_insertnode(struct drm_node_s *parent_node, void *prop, struct drm_node_s **new_node)`
`drm_delete_subtree()`	`int drm_delete_subtree(struct drm_node_s *handle)`
`drm_prune_subtree()`	`int drm_prune_subtree(struct drm_node_s *handle)`
`drm_select_subtree()`	`int drm_select_subtree(struct drm_node_s *handle)`
`drm_unselect_subtree()`	`int drm_unselect_subtree(struct drm_node_s *handle)`
`drm_select_node()`	`int drm_select_node(struct drm_node_s *handle)`
`drm_unselect_node()`	`int drm_unselect_node(struct drm_node_s *handle)`
`drm_alloc_resource()`	`int drm_alloc_resource(struct drm_node_s *handle)`

Table 9-3: DRM Service Routine Interface Specification (Continued)

Name	Synopsis
`drm_free_resource()`	`int drm_free_resource(struct drm_node_s *handle)`
`drm_get_handle()`	`int drm_get_handle(int buslayer_id, int vendor_id, int device_id, struct drm_node_s **handle)`
`drm_claim_handle()`	`int drm_claim_handle(struct drm_node_s *handle)`
`drm_free_handle()`	`int drm_free_handle(struct drm_node_s *handle)`
`drm_register_isr()`	`int drm_register_isr(struct drm_node_s *handle, void (isr)(), void *args)`
`drm_unregister_isr()`	`int drm_unregister_isr(struct drm_node_s *handle)`
`drm_map_resource()`	`int drm_map_resource(struct drm_node_s *handle, int resource_id, addr_t *vaddr)`
`drm_unmap_resource()`	`int drm_unmap_resource(struct drm_node_s *handle, int resource_id)`
`drm_device_read()`	`int drm_device_read(struct drm_node_s *handle, int resource_id, unsigned int offset, unsigned int size, void *buffer)`
`drm_device_write()`	`int drm_device_write(struct drm_node_s *handle, int resource_id, unsigned in offset, unsigned int size, void *buffer)`
`drm_getroot()`	`int drm_getroot(struct drm_node_s **root_handle)`
`drm_getchild()`	`int drm_getchild(struct drm_node_s *handle, struct drm_node_s **child)`
`drm_getsibling()`	`int drm_getsibling(struct *handle, struct drm_node_s **sibling)`
`drm_getparent()`	`int drm_getparent (struct drm_nodes_s *handle, struct drm_node_s **parent)`
`drm_getnode()`	`int drm_getnode (struct drm_nodes_s *src, struct drm_node_s **dest)`
`drm_setnode()`	`int drm_setnode (struct drm_nodes_s *handle)`

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:

```
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 to 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.

```
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.

```
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 facilitate the selection and dynamic loading of needed drivers and provides them with node handles for use.

Device Interrupt Management

DRM maintains all interrupt routing data for a device node. Drivers use the `drm_register_isr()` service routine to register an interrupt service routine and the `drm_unregister_isr()` service routine to clear a registration. Typically, this service routine is 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. The returned `link_id` is `NULL`, and the `iointlink()` kernel service routine does not perform any dispatches.

The following code segments illustrate the use of these DRM service routines:

```
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);
}

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:

```
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_RESID_BAR1,
                       &scsi_vaddr);
  if(ret)
  {
    /* Bogus resource_id ? */
    /* resource not mappable */
    /* invalid device handle ? */
  }
  scsi_control_regs = (struct scsi_control *)(scsi_vaddr);
  ret =drm_unmap_resource(handle,PCI_RESID_BAR1);
  if(ret)
  {
    /* Bogus handle */
    /* resource is not mappable */
    /* resource was not mapped */
    /* invalid resource_id */
  }
}
```

Device I/O

DRM provides service routines to perform read and write to 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 `machine/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` parameters 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 <machine/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. Most of the DRM facilities are accessed by user mode programs using the `sysctl()` interface.

Using DRM Facilities from Applications

User mode applications use the `sysctl()` interface to get access to DRM facilities. It is possible to traverse the DRM tree, get node data, and to perform insertions and deletions using the `sysctl()` interface. The `sys/drm_sysctl.h` header file defines the MIB names and `sysctl()` data structures that are used. See the `sysctl()` man page for details on how to use the system call. The `sysctl()` call is invoked as:

```
(int) ret = sysctl(int *name, u_int namelen,
                   void *oldp, size_t *oldlenp,
                   void *newp, size_t newlen);
```

`sysctl()` parameters are described in the table below.

The top-level MIB name to access DRM-related information is `CTL_HW`. The second-level MIB name to access DRM information is `HW_DRM`. The third-level names provide DRM-specific facilities as described in the following table.

Table 9-4: sysctl() Parameters and DRM-Specific Facilities

DRM Facility	Description

DRM_GET_ROOT	This provides the sysctl() interface to the drm_getroot() service routine. The handle to the root of the DRM tree is returned in oldp. In the current implementation of DRM, the handle is returned as a (void *) pointer with a size of 32 bits. A buffer that is sufficient to hold a (void *) pointer needs to be provided in oldp to hold the returned handle. For example, the sysctl() system call is made as follows:
	`{`
	`  int mib[3];`
	`  void *handle;`
	`  int len;`
	`  mib[0] = CTL_HW;`
	`  mib[1] = HW_DRM;`
	`  mib[2] = DRM_GET_ROOT;`
	`  len = sizeof(handle);`
	`  ret = sysctl(mib,3,&handle,`
	`               &len, 0,0);`
	`}`

DRM_GET_PARENT	This provides the sysctl() interface to the drm_getparent() service routine. The fourth-level MIB name is set to the handle of the reference node for which a parent node is desired. The parent handle is returned in oldp in a manner similar to DRM_GET_ROOT. An example of the sysctl() system call follows:
	`{`
	`  int mib[4];`
	`  void *parent_handle;`
	`  int len;`
	`  mib[0] = CTL_HW;`
	`  mib[1] = HW_DRM;`
	`  mib[2] = DRM_GET_PARENT;`
	`  mib[3] = handle;`
	`  /* handle to the reference node */`
	`  len = sizeof(parent_handle);`
	`  ret = sysctl(mib,4,&parent_handle,`
	`        &len, 0,0);`
	`}`

DRM_GET_CHILD	This provides the sysctl() interface to the drm_getchild() service routine. The fourth-level MIB name is set to the handle of the reference node. The child handle is returned in oldp in a manner similar to DRM_GET_PARENT.

Table 9-4: sysctl() Parameters and DRM-Specific Facilities (Continued)

DRM Facility	Description
`DRM_GET_SIBLING`	This provides the `sysctl()` interface to the `drm_getsibling()` service routine. The fourth-level name is set to the handle of the reference node. The sibling handle is returned in `oldp` in a manner similar to `DRM_GET_PARENT`.

`DRM_GET_NODE`	This call provides the DRM node data to the application. The fourth-level name is set to the handle of the reference node. The DRM node data is returned in a `drm_sc_node` structure given in `oldp`. Look for the definition of `drm_sc_node` in `sys/drm_sysctl.h`. A typical use of `DRM_GET_NODE` is as follows:
	`{`
	`  int mib[4];`
	`  struct drm_sc_node node;`
	`  int len;mib[0] = CTL_HW;`
	`  mib[1] = HW_DRM;`
	`  mib[2] = DRM_GET_NODE;`
	`  mib[3] = handle;  /* handle to the reference node */`
	`  len = sizeof(node);`
	`  /* get the data */`
	`  ret = sysctl(mib,4,&node, &len, 0,0);`
	`  printf("Vendor ID =`
	`         %x\n",node.vendor_id);`
	`  printf("Device ID = %x\n",`
	`          node.device_id);`
	`  printf(" Primary buslayer ID =`
	`         %d\n",node.pbuslayer_id);`
	`  printf(" Node type =`
	`         %d\n",node.node_type);`
	`  printf(" Node state = %d\n",node.state);`
	`  if(node.node_type & DRM_BUS)`
	`  {`
	`    printf("Secondary buslayer_id`
	`    d\n",node.sbuslayer_id);`
	`  }`
	`}`

Hot Swap Management Applications

There are special facilities available for Hot Swap management applications. These facilities are specified as commands to the `DRM_CMD` third-level facility. The table below lists the commands that are available at this level.

Table 9-5: Hot Swap Facilities

Facility	Description
CMD_PROBE	Provides a `sysctl()` interface to the `drm_locate()` service routine. The fifth-level name provides a reference node for the probe.
CMD_SELECT	Provides the `sysctl()` interface for the `drm_select_subtree()` and `drm_select_node()` service routines.
CMD_ALLOC	Provides a `sysctl()` interface to the `drm_alloc_resource()` service routine.
CMD_PRUNE	Provides a `sysctl()` interface to the `drm_prune_subtree()` service routine.
CMD_UNSELECT	Provides a `sysctl()` interface to `drm_unselect_subtree()` and `drm_unselect_node()` service routines.
CMD_INSERT	Provides the `sysctl()` interface to the `drm_insertnode()` service routine.
CMD_FREE	`drm_free` resource service routine

The following code fragments illustrate how this interface is used.

```
do_probe(void *ref_node) {
  int  mib[5];
  int  retval;
  int len;
  int ret;
  mib[0] = CTL_HW;
  mib[1] = HW_DRM;
  mib[2] = DRM_CMD;
  mib[3] = CMD_PROBE;
  mib[4] = ref_node;
  len = sizeof(retval);
  /* perform the probe */
  ret = sysctl(mib,5,&retval,&len,0,0);
...
}
```

A sample of the `CMD_INSERT` code is shown below. The `prop` structure is filled in with bus layer-specific data that provides information on the node being inserted. The `ref_node` is the parent node of the node to be inserted.

```
do_insert(void *ref_node, void *prop, prop_len) {
  int  mib[5];
  int  handle;
```

```
    int len;
    int ret;
    mib[0] = CTL_HW;
    mib[1] = HW_DRM;
    mib[2] = DRM_CMD;
    mib[3] = CMD_INSERT;
    mib[4] = ref_node;
    len = sizeof(handle);
    /* perform the insert */
    ret = sysctl(mib,5,&handle,&len,prop,prop_len);
    ...
  }
```

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;
};

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));

#else /* Hot Swap capable */

    rv = drm_claim_dhandle(info->handle);
    static_ptr->handle = info->handle;

#endif

    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 */

        /* Succesfull 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 */

    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 */

abc_open(…) {
}
```

```
abc_read(…) {
}

abc_write(…) {
}

abc_ioctl(…) {
}

abc_close(…) {
}

abc_isr(...) {
}
```

Sample Application

```
/*
******************************************************
   This program lists all boards in a Motorola 8216 chasis
******************************************************
*/

#include "errno.h"
#include <pci_resource.h>
#include "sys/sysctl.h"
#include "sys/drm_sysctl.h"
#include "sys/pci_sysctl.h"

#define TRUE 1
#define FALSE 0

#define OK TRUE
#define NOT_OK FALSE

#define DEV_NODE8
#define BUS_NODE4

#define MAX_SLOT 16
unsigned int root_node = 0;
unsigned int curr_node = 0;

struct drm_sc_node sc_node;
struct pci_sc_node pci_node;
struct pci_sc_busnode pci_busnode;
int mib[12];
int miblen;

int retval;
int cpx_slot;
struct drm_node_s *slot_handle;
struct drm_node_s *domainA_handle;
int slot_tbl[MAX_SLOT+1];

main()
{
    int ret;
    int dev_no;
    unsigned int len;

    printf("Slot#State\n");

    /* Get the root node */

    mib[0] = CTL_HW;
    mib[1] = HW_DRM;
    mib[2] = DRM_GET_ROOT;

    len = sizeof(root_node);

    ret = sysctl(mib,3,&root_node,&len,NULL,0);

    if(ret)
    {
        perror("hsls-1:");
        return(1);
    }

    curr_node = root_node;

    /* Get the DomainA Bridge */

    while(OK == get_next_node())
    {
        if(check_node(0x26,0x1011) == OK) break;
    }

    domainA_handle = (struct drm_node_s *)curr_node;

    /* Init slot table */

    for(cpx_slot = 1; cpx_slot < MAX_SLOT+1;
            cpx_slot++)
    {
        slot_tbl[cpx_slot] = 0;
    }

    /* Get the child of the Domain Bridge */


    ret =get_handle(domainA_handle,DRM_GET_CHILD,
            &slot_handle);

/* Get all the siblings and populate the slot table */
/*
The slot number (index to slot_tbl) is derived from the
pci-device number from the following Motorola 8216
specific formula:
        slot_number = 15 - pci_device_no ;
And depends on the wiring of the cPCI backplane.
*/

    while(slot_handle)
    {
        dev_no = get_devno(slot_handle);
        slot_tbl[15-dev_no] = (int)slot_handle;
        ret = get_handle(slot_handle,DRM_GET_SIBLING,
            &slot_handle);
        if(ret == NOT_OK) break;
    }

    /* Display the slot table */

    for(cpx_slot = 1; cpx_slot < MAX_SLOT+1; cpx_slot++) {
        switch(cpx_slot) {
            case 7:
            case 9:
            printf("%4.4d    %s\n",cpx_slot,
            "System Controller");
            break;
            case 8:
            case 10:
            printf("%4.4d    %s\n",cpx_slot,
            "Not Present");
            break;
            default:
            if(slot_tbl[cpx_slot])
            {
            printf("%4.4d    %s\n",cpx_slot,
            "occupied");
            } else {
            printf("%4.4d    %s\n",cpx_slot,
            "empty");
            }
            break;
        }
    }
}

/* Traverse the DRM Tree */

int get_next_node() {

    if(TRUE == has_child())
    {
        get_child();
        return OK;
    }

    if(TRUE == has_sibling())
    {
        get_sibling();
        return(OK);
    }

    while(TRUE == has_parent())
    {
        get_parent();
        if(TRUE == has_sibling())
        {
            get_sibling();
            return(OK);
        }
    }
    return(NOT_OK);
}

/* Get a parent,child or sibling node */

int get_node(type)
int type;
{
```

```
    int ret;
    unsigned int node;
    unsigned int len;


    mib[0] = CTL_HW;
    mib[1] = HW_DRM;
    mib[2] = type;
    mib[3] = curr_node;

    len = sizeof(node);
    ret = sysctl(mib,4,&node,&len,NULL,0);

    if(ret)
    {
        perror("hsls-2:");
        return(1);
    }

    return(node);

}

/* Check if currnode has children */

int has_child()
{
    unsigned int node;
    node = get_node(DRM_GET_CHILD);

    if(!node || node == -1)
        return(FALSE);
    else
        return(TRUE);
}

/* Get child of current node */

get_child()
{
    return curr_node = get_node(DRM_GET_CHILD);
}

/* Check if curr node has sibling */

int has_sibling()
{
    unsigned int node;
    node =  get_node(DRM_GET_SIBLING);
    if(!node || node == -1)
        return(FALSE);
    else
        return(TRUE);
}

/* Get sibling of current node */

get_sibling()
{
    return curr_node = get_node(DRM_GET_SIBLING);
}
```

```
/* Check if current node has a parent */

int has_parent()
{
   unsigned int node;
   node = get_node(DRM_GET_PARENT);
   if(!node || node == -1)
       return(FALSE);
   else
       return(TRUE);
}

/* Get current node’s parent */

get_parent()
{
   return curr_node = get_node(DRM_GET_PARENT);
}

/* Check if current node matches given vendor,device id */


check_node(int dev,int vend)
{
   int ret;
   unsigned int len;


   mib[0] = CTL_HW;
   mib[1] = HW_DRM;
   mib[2] = DRM_GET_NODE;
   mib[3] = curr_node;

   len = sizeof(sc_node);
   ret = sysctl(mib,4,&sc_node,&len,NULL,0);

   if(ret)
   {
       perror("hsls-3:");
       return(1);
   }


   if((dev == sc_node.device_id) &&
      (vend == sc_node.vendor_id)) return (OK);

   return(NOT_OK);
}

/* Get the PCI device number given a node handle */

int get_devno(handle)
void *handle;
{
   int ret;
   unsigned int len;


   mib[0] = CTL_HW;
   mib[1] = HW_DRM;
   mib[2] = DRM_GET_NODE;
   mib[3] = (int)handle;
```

```
    len = sizeof(sc_node);
    ret = sysctl(mib,4,&sc_node,&len,NULL,0);

    if(ret)
    {
        perror("hsls-4:");
        return(NOT_OK);
    }


    if((sc_node.pbuslayer_id == PCI_BUSLAYER) &&
      ((sc_node.node_type & DEV_NODE) == DEV_NODE))
    {
        mib[0] = CTL_HW;
        mib[1] = HW_DRM;
        mib[2] = DRM_PCI;
        mib[3] = PCI_GET_DEVNODE;
        mib[4] = (int)handle;

        len = sizeof(pci_node);
        ret = sysctl(mib,5,&pci_node,&len,NULL,0);

        if(ret)
        {
            perror("hsls-5:");
            return(NOT_OK);
        }

        return(pci_node.device_no);
    }

    if (((sc_node.node_type & BUS_NODE) == BUS_NODE) &&
        (sc_node.pbuslayer_id == PCI_BUSLAYER) &&
        (sc_node.sbuslayer_id  == PCI_BUSLAYER))
    {
        mib[0] = CTL_HW;
        mib[1] = HW_DRM;
        mib[2] = DRM_PCI;
        mib[3] = PCI_GET_BUSNODE;
        mib[4] = (int)handle;

        len = sizeof(pci_busnode);
        ret = sysctl(mib,5,&pci_busnode,&len,NULL,0);

        if(ret)
        {
            perror("hsls-6:");
            return(NOT_OK);
        }

        return(pci_busnode.device_no);
    }

    return(NOT_OK);
}

/* Given a reference handle, get the handle of it's parent,child or sibling */

int get_handle(handle,type,result)
void *handle;
int type;
void **result;
{
    int ret;
    unsigned int node;
    unsigned int len;


    mib[0] = CTL_HW;
    mib[1] = HW_DRM;
    mib[2] = type;
    mib[3] = (int)handle;

    len = sizeof(node);
    ret = sysctl(mib,4,&node,&len,NULL,0);

    if(ret)
    {
        perror("hsls-7:");
        return(NOT_OK);
    }

    *result = (void *)node;
    return(OK);

}
```

CHAPTER 10

Writing Flash Memory Technology Drivers (MTDs)

Access to a linear flash device in LynxOS is implemented through a two layer model. The upper layer, `flash_mgr(4)`, implements the user interface and encompasses all common algorithms required to access any linear flash device. This includes argument checking, memory mapping (if required) and adjustments, synchronization, and so on. The lower layer is a Memory Technology Driver (MTD), which is a device driver responsible for implementing hardware-specific details of programming a particular flash device.

The MTD does not interface directly with user applications. It interacts with the `flash_mgr` module, through an interface defined by a set of entry points and data structures, described in the following sections.

This two layer model provides a unified interface to any flash device and removes the need for redundant algorithms within MTD modules.

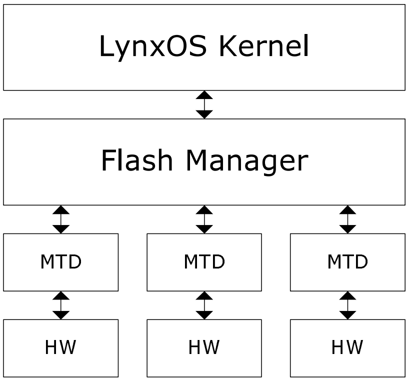


Figure 10-1: Flash Manager & MTD Overview

Cache Management

The cache of a flash device can be managed by either the MTD or `flash_mgr`. For a flash device with unequal segment sizes, the MTD must mange the cache. `flash_mgr` requires that the sector sizes beof equal size.

Interface Overview

An MTD is implemented as a character device driver. From the point of view of the device driver structure, an MTD is very simple - there are only two entry points required by the LynxOS character device interface that are used in an MTD. These are `install` and `uninstall`.

In the `install` routine, an MTD registers a callback routine, along with some related data, with the `flash_mgr` module. The callback routine is responsible for implementing a fixed set of flash memory operations for the particular flash device. `flash_mgr` invokes the callback routine any time there is the need to access the flash device at the physical level.

In the `uninstall` routine, the MTD deregisters the callback, thus notifying the `flash_mgr` that the MTD is no longer available for device accesses.

While there is the single `flash_mgr` component in the kernel, there may be multiple active (registered) MTDs, each implementing access to its own flash device. `flash_mgr` supports multiple opens to different flash devices and concurrent I/O operations for them. The Flash ID, passed by an MTD to `flash_mgr` at registration time, is used as a key for mapping application requests to a particular MTD.

The following sections provide a detailed description of the interface between the `flash_mgr` module and an MTD. All entry points and data structures are defined in the header file `$ENV_PREFIX/sys/dheaders/flash_mtd.h`.

Registering with flash_mgr

An MTD registers with `flash_mgr` by invoking the `flash_mtd_register` entry point. A pointer to the MTD registration data is passed as the only parameter to the routine.

MTD Registration Data

The MTD registration data, as passed by an MTD to `flash_mgr`, is described by the following data structures:

```
typedef struct flash_mtd_area_s
{
   u_int        offset;    /* Offset from Flash Base*/
   u_int        size;      /* Size in Bytes         */
}
flash_mtd_area_t;

typedef struct flash_mtd_register_s
{
  int          flash_id;   /* Flash ID              */
  u_int        mtd_attr;   /* MTD Attributes        */
  u_int        flash_addr; /* Flash Base            */
  u_int        sector_size;/* Sector Size (bytes)   */
  u_int        flash_size; /* Flash Size (bytes)    */
  char *       info_str;   /* Device Info String    */
  flash_mtd_area_t         /* Device Control        */
                           /* Registers             */
               spec_locs[FLASH_MTD_SPEC_LOC_NUM];
  flash_mtd_area_t         /* Partitions Info       */
               parts[FLASH_MTD_PART_NUM];
  void *       user_param; /* MTD Specific Data     */
  flash_mtd_callback_t     /* Pointer to Callback   */
               callback;
}
flash_mtd_register_t;
```

To register with `flash_mgr`, an MTD must fill in a structure of type `flash_mtd_register_t` with the registration data. The various fields in the structure should be initialized to appropriate values. Once the structure has been initialized, the MTD makes itself known to `flash_mgr` by calling `flash_mtd_register` and passing the address of the structure as the parameter. If registration is successful, `flash_mtd_register` returns zero. For example:

```
char * flash_example_install(
       flash_example_info_t * info )
{
   flash_mtd_register_t reg_data;
   ...
     /* Fill in the registration data.
      */
   reg_data.flash_id = ...;
   ...
     /* Register with the flash manager.
      */
   if ( flash_mtd_register( & reg_data ) != 0 )
   {
     return (char *) SYSERR;
   }
   ...
}
```

The following provides example registration code for the Intel i28F400 flash device.

```
/*
**  Install entry point
**
**  Dependency: This function has Falcon-specific code.
**
*/
char *
i28F400_r_install(i28F400_info_t *info)
{
    int i;
    i28F400_statics_t *s;
    flash_mtd_register_t reg;
    u_int *romAbasesiz_reg;

    /* Allocate a statics structure */
    s = (i28F400_statics_t *)sysbrk(sizeof(i28F400_statics_t));
    if (s == NULL) {
        s = (i28F400_statics_t *)SYSERR;
        return (char *)s;
    }

    /* Map the ROM A Base/Size Register */
    romAbasesiz_reg = map_romA_basesiz_reg();

    /* Calculate flash memory size */
    s->flash_size = calc_i28f400_size(romAbasesiz_reg);
    if ((int)s->flash_size == SYSERR) {
        sysfree(s, (long)sizeof(*s));
        s = (i28F400_statics_t *)SYSERR;
        goto i28F400_r_install_done;
    }

    /* Map the entire flash address space */
    s->vaddr = (char *)map_28F400(romAbasesiz_reg, s->flash_size);
    if (s->vaddr == (char *)SYSERR) {
        sysfree(s, (long)sizeof(*s));
        s = (i28F400_statics_t *)SYSERR;
        goto i28F400_r_install_done;
    }

    /* Write enable flash bank A */
    i28F400_write_enable(romAbasesiz_reg);

    /* Set the flash device to Read Array mode */
    set_read_array_28f400(romAbasesiz_reg, (u_int *)s->vaddr, s-
>flash_size);

    /* Get device id and mfr id */
    s->i28F400_ids = get_i28F400_ids(romAbasesiz_reg, (u_int *)s->vaddr,
                                     s->flash_size);
    if (s->i28F400_ids < 0) {
        sysfree(s, (long)sizeof(*s));
        s = (i28F400_statics_t *)SYSERR;
        goto i28F400_r_install_done;
    }

    /* Fill up the statics structure */

    s->flash_id = info->flash_id;
    s->romAbasesiz_reg = romAbasesiz_reg;

    /* Fill up block layout */
    fill_block_layout((s->i28F400_ids & 0x0000FFFF), &s->layout[0]);
    /* Allocate space for cache */
    s->cache = alloc_28F400_cache();
    if (s->cache == NULL) {
        sysfree(s, (long)sizeof(*s));
        s = (i28F400_statics_t *)SYSERR;
        goto i28F400_r_install_done;
    }
    s->cache->blkno = 0;
    s->cache->dirty = 0;
    memcpy(s->cache->cache, s->vaddr + s->layout[0].offset, s-
>layout[0].size);

    s->erase_sem = 1;
    s->erased_bitmap = 0;

    /* Prepare registration information */

    /* No command registers */
    for (i = 0; i < FLASH_MTD_SPEC_LOC_NUM; i++) {
        reg.spec_locs[i].size = 0;
    }

    /* Set up partitions */
    for (i = 0; i < FLASH_MTD_PART_NUM; i++) {
        reg.parts[i].offset = info->parts[i].offset;
        reg.parts[i].size = info->parts[i].size;
    }

    /* Registration attributes */
    reg.mtd_attr = FLASH_MTD_ATTR_NO_MAP    | /* Already mapped    */
                   FLASH_MTD_ATTR_READ      | /* Read Op.  */
                   FLASH_MTD_ATTR_WRITE     | /* Write Op.  */
#if defined( PSEUDO )
                   FLASH_MTD_ATTR_OPEN      | /* Open Op.  */
#endif
                   FLASH_MTD_ATTR_CLOSE     | /* Close Op.  */
                   FLASH_MTD_ATTR_ERASE     | /* Erase Op.  */
                   FLASH_MTD_ATTR_ERASE_ALL | /* Erase All Op.     */
                  FLASH_MTD_ATTR_SPECIFIC;   /* MTD specific operations */

    reg.flash_id = s->flash_id;
    reg.flash_addr = (u_int)s->vaddr;
    reg.sector_size = n_8KB * 4;

    reg.flash_size = s->flash_size;
    reg.info_str = "Intel 28F400 Flash";
    reg.user_param = s;
    reg.callback = i28F400_callback;

    /* Register with the flash manager */
    if (flash_mtd_register(&reg)) {
        free_i28F400_statics(s);
        s = (i28F400_statics_t *)SYSERR;
    }

i28F400_r_install_done:
    /* Write disable flash bank A */
    i28F400_write_disable(romAbasesiz_reg);
    return (char *)s;
}
```

Flash ID

The Flash ID is one of the most important elements of the registration data. It is an integer key used by `flash_mgr` to identify a particular MTD among all the MTD modules installed in the kernel. The flash ID number is also used to generate the minor number for the flash device. All flash devices use the same major number. To access a flash device, the application opens a corresponding device special file. The Flash ID is encoded into a 4 bit field within the minor device number. `flash_mgr` extracts it and uses it as a key to map application requests to a particular MTD module.

The following describes the minor number layout for flash device nodes:

```
76.43..0
VPPPDDDD
```

Where:

- *V* specifies the verification mode. If this bit is set, driver operates in the transparent verification-on-write-and-erase mode. Otherwise, driver operates in the no-verification mode.
- *P* is a 3-bit field specifying the partition number. A value of 0 corresonds to the entire flash device; any other value (1 to 7) defines the partition number. The partition information is passed to `flash_mgr` by the MTD at the registration time.
- *D* is a 4-bit field specifying the Flash ID. The Flash ID can range from 0 to 15. The Flash ID is used by the `flash_mgr` to map a request to the corresponding MTD. MTD passes its Flash ID to `flash_mgr` at registration time.

The Flash ID number is chosen by the developer, and set by the MTD at registration time. A Flash ID number must be unique for each MTD configured into the kernel. The 4 bit Flash ID field in the minor number allows you to install up to 16 MTDs simultaneously, with Flash IDs ranging from 0 to 15. For an easier configuration, it is recommended that the Flash ID be copied from the device information block, as shown in the example below:

```
reg_data.flash_id = info->flash_id;
```

Note that once you modify the Flash ID of an MTD, you have to change the minor numbers for all corresponding device nodes accordingly. By default, special nodes for the flash devices are installed in the configuration file `$ENV_PREFIX/sys/cfg/flash.cfg`.

For example,

```
# AM29LV160BT on-board flash MTD, id 5
N:flash_am29lv160bt.0:5:0600
N:flash_am29lv160bt.1:21:0600
N:flash_am29lv160bt.2:37:0600
N:flash_am29lv160bt.3:53:0600
```

Where:

N:<*flash_device.n*>:<*minor#*>:<*file_permissions*>

Device Info String

The `info_str` field is a pointer to the device information string. This is an arbitrary character string describing the flash device. `flash_mgr` returns this string as a part of the flash information block whenever an application issues a `FLASH_GET_INFO ioctl` command. The string must be contained in a nonautomatic variable. For example:

```
reg_data.info_str = “rpxl8xx on-board FLASH”;
```

Flash Size

The size of the flash memory is passed to `flash_mgr` through the `flash_size` field. Size is specified in bytes:

```
reg_data.flash_size = 4 * MByte;
```

Sector Size

The size of the flash device sector is passed to `flash_mgr` through the `sector_size` field. The sector size should be the smallest sector size of a device, specified in bytes:

```
reg_data.flash_size = 256 * KByte;
```

Registration Attributes

The registration attributes are passed to `flash_mgr` through the `mtd_attr` field as a bitwise combination of the following logical flags:

Table 10-1: MTD Registration Attributes

Attribute	Description
`FLASH_MTD_ATTR_READ`	MTD supports read accesses to the flash device.
`FLASH_MTD_ATTR_WRITE`	MTD supports write accesses to the flash device.
`FLASH_MTD_ATTR_ERASE_ALL`	MTD supports erase of the entire flash device. If this flag is not specified, any ioctl request for `FLASH_ERASE_ALL` is denied.
`FLASH_MTD_ATTR_ERASE`	MTD supports erase of a particular sector of the flash device. If this flag is not specified, any `ioctl` request for `FLASH_ERASE_BLOCK` is denied.
`FLASH_MTD_ATTR_SPECIFIC`	MTD supports device-specific operations. If this flag is not specified, any `ioctl` request for `FLASH_SPECIFIC` is denied.
`FLASH_MTD_ATTR_FF_ERASED`	For this device, an erased byte returns `0xFF` on a read access. Specification of this flag for a device granting this condition is likely to improve the overall driver throughput. This flag is used only if the `flash_mgr` is used to manage the cache.
`FLASH_MTD_ATTR_AUTO_READ`	MTD relies on `flash_mgr` to execute read accesses. The MTD callback routine is not invoked upon a read request from user application. Instead, `flash_mgr` reads flash memory as conventional memory. If this flag is specified, `FLASH_MTD_ATTR_READ` is ignored. This flag is used only if the `flash_mgr` is used to manage the cache.
`FLASH_MTD_ATTR_NO_MAP`	Entire flash memory is already mapped into a contiguous region of the kernel virtual space by the MTD. No mapping is needed in `flash_mgr`. If this flag is specified, `flash_mgr` interprets the flash base address as a virtual address in the kernel space. The MTD must manage the cache if the flash memory sector sizes are different.

Table 10-1: MTD Registration Attributes (Continued)

Attribute	Description
`FLASH_MTD_ATTR_MAP_SPECIFIC`	MTD notifies `flash_mgr` that whenever a `FLASH_MTD_REQ_SPECIFIC` operation is invoked on the MTD, the entire flash memory must be mapped into a contiguous region of the kernel virtual space. If this flag is not specified, any `ioctl` request for `FLASH_SPECIFIC` is denied.
`FLASH_MTD_ATTR_CACHE`	MTD requests that the special flash cache be enabled by `flash_mgr`. The flash cache resides on top of the in-memory disk cache and is useful only if flash device is accessed through the block interface of `flash_mgr`. Synchronization of a flash device with enabled flash cache requires execution of the `flash_sync(1)` utility after a write out of the disk cache. Flash cache is designed to both improve the overall flash file system access rate and prevent the flash memory exhaustion. It is especially efficient for devices with a large sector size.
`FLASH_MTD_ATTR_OPEN`	This flag requests the open MTD callback to be called.
`FLASH_MTD_ATTR_CLOSE`	This flag requests the close MTD callback to be called.

The following example shows how the `mtd_attr` field can be initialized for the registration:

```
reg.mtd_attr = FLASH_MTD_ATTR_NO_MAP    |
                       /* Already mapped            */
               FLASH_MTD_ATTR_WRITE     |
                       /* Write Op.                 */
               FLASH_MTD_ATTR_AUTO_READ |
                       /* No Read Op.               */
               FLASH_MTD_ATTR_ERASE     |
                       /* Erase Op.                 */
               FLASH_MTD_ATTR_ERASE_ALL |
                       /* Erase All Op.             */
               FLASH_MTD_ATTR_FF_ERASED |
                       /* 0xFF if erased            */
               FLASH_MTD_ATTR_CACHE;
                       /* Flash cached              */
```

Flash Base Address

The `flash_base` field is used to pass the base address of the flash memory to `flash_mgr`. Interpretation of this field depends on whether the `FLASH_MTD_ATTR_NO_MAP` flag is set in the `mtd_attr` field.

If the flag is set, `flash_mgr` interprets the base address as a virtual address in the kernel memory map and relies on the entire flash memory to be already mapped into the kernel space. `flash_base` in this case contains a starting address of the flash memory in the virtual space. In the following example MTD uses the `permap(9)` kernel service to map the entire flash into the kernel space:

```
  /* Notify flash_mgr that flash is mapped by the MTD.
   */
reg.mtd_attr   = FLASH_MTD_ATTR_NO_MAP | ... ;
  /* Map the flash; set flash_base to the
   * virtual address.
   */
reg.flash_base = (u_int) permap( FLASH_PHYS_BASE,
                                 FLASH_SIZE );
```

If the flag is not set, `flash_mgr` interprets the base address as a physical address of the flash memory in the system memory map. In this case, `flash_mgr` takes over the burden of mapping the flash memory and guarantees that whenever the MTD callback routine is invoked, part of flash memory on which the operation is executed is mapped into the kernel virtual space. The following example illustrates this approach:

```
   /* Notify flash_mgr that flash is not mapped.
    */
reg.mtd_attr = /* no FLASH_MTD_ATTR_NO_MAP */ | ... ;
   /* Set flash_base to the physical address.
    */
reg.flash_base = FLASH_PHYS_BASE;
```

NOTE: If the flash device is used to store crash dump data, the base address must be mapped manually.

Device Control Registers

An MTD is allowed to register up to three control register windows. A control register window is an address range within the flash memory that is used to access hardware control and status registers. Usually, an MTD needs to be able to access the control and status registers in order to program certain operations of the flash device (for example, flash erase).

Note that registration of control registers windows is needed only if an MTD does not use the `FLASH_MTD_ATTR_NO_MAP` attribute to register with the `flash_mgr` module. In this case `flash_mgr` takes over the responsibility of mapping the hardware registers and guarantees that any time the MTD callback routine is invoked, all registered control registers windows are mapped into the kernel virtual space.

Because the hardware control registers are located within the flash memory address range, there is no need to define control registers windows for an MTD that uses the `FLASH_MTD_ATTR_NO_MAP` attribute for the registration. As soon as the MTD creates the mapping for the entire flash memory, the control registers windows become mapped automatically.

The control registers windows are created by initializing the `spec_locs` array. If a window is unused, the corresponding `size` field should be set to zero.

The following example shows registration of two control registers windows:

```
   /* Notify flash_mgr that flash is not mapped.
    */
reg.mtd_attr = /* no FLASH_MTD_ATTR_NO_MAP */ | ... ;
   /* Define the hardware registers windows.
    */
for ( i = 0; i < FLASH_MTD_SPEC_LOC_NUM; i ++ )
{
   reg.spec_locs[i].size  = 0;
}
reg.spec_locs[0].offset  = 0x5555 * 4;
                             /* Command Reg1 */
reg.spec_locs[0].size    = 4;/* 4 bytes      */
reg.spec_locs[1].offset  = 0x2AAA * 4;
                             /* Command Reg2 */
reg.spec_locs[1].size    = 4;/* 4 bytes      */
```

Partition Information

A flash memory device can be divided in up to seven possible overlapping partitions. Unlike a conventional disk, there is no partition table or a similar partition descriptor maintained in flash memory. Instead, the partitions data is maintained in the software tables of `flash_mgr` on a per-device basis. Device partition information is passed by an MTD to the `flash_mtd` module at registration time and is effective until the MTD is deregistered.

Device partition information is passed to the `flash_mgr` through the `parts` field, which is a 7 entry table of partition descriptors. Each entry of the table contains information about one partition of the flash device. The entry with index 0 defines partition number 1, the entry with index 1 partition number 2, and so on. If an entry has the `size` field set to zero, the corresponding partition is undefined and cannot be accessed from user applications.

To get access to a flash partition, an application opens a corresponding device special node. The partition number is encoded as a 3 bit field into the device minor number. A value of 0 corresponds to the entire flash device; any other value (1 to 7) defines the partition number.

Other than passing the partition information to `flash_mgr` at the registration time, an MTD is insensitive to device partitions. Flash addresses and sector numbers passed to the MTD callback routine by `flash_mgr` are relative to the flash memory base.

To allow for easy partitioning of a flash device, it is recommended that the partitions information be copied from the device information block, as shown in the example below:

```
for ( i = 0; i < FLASH_MTD_PART_NUM; i ++ )
{
  reg.parts[i] = info->parts[i];
}
```

The user can change the device partitions by modifying the device information block and rebuilding the kernel. Shown below is a sample device block for the above example:

```
flash_example_info_t flash_example_info =
{
     ...
   /* Device Partitions :
    * there are 7 partitions;
    * there must be an entry for each.
    * If a partition is unused, size is set to 0.
    */
  0 * MByte, 1 * MByte, /* Partition 1: 0 - 1 MB    */
  1 * MByte, 3 * MByte, /* Partition 2: 1 MB - 4 MB */
  0, 0,                 /* Partition 3: unused      */
  0, 0,                 /* Partition 4: unused      */
  0, 0,                 /* Partition 5: unused      */
  0, 0,                 /* Partition 6: unused      */
  0, 0,                 /* Partition 7: unused      */
};
```

Callback Routine

A pointer to the callback routine is passed to `flash_mgr` through the `callback` field. Pointer to MTD specific data is passed through the `user_param` field. This pointer is passed back to the callback routine any time the callback is invoked by `flash_mgr`.

The following example shows registration of a callback routine. A pointer to the MTD statics structure is registered as the MTD specific data:

```
flash_statics_t * s;
...
reg.user_param = s;
reg.callback   = flash_example_callback;
```

Deregistering from flash_mgr

An MTD deregisters from `flash_mgr` by invoking the `flash_mtd_deregister` entry point. The Flash ID is passed as the only parameter to the routine. For example:

```
void flash_example_uninstall(
     flash_example_statics_t * s )
{
   ...
     /* Deregister the MTD.
      */
   flash_mtd_deregister( s->flash_id );
```

Writing Callback Routines

The MTD callback routine is invoked by the `flash_mgr` module with the following syntax:

```
int flash_mtd_rpxl_callback (
  int op,                          /* Operation Code         */
  u_int flash_base,                /* Flash in Virtual Space */
  flash_mtd_param_t * op_param,    /* Operation Parameter    */
  void * user_param,               /* MTD Specific Data      */
  int flags);                      /* MTD Flags              */
```

Operation Code

The first parameter contains an operation code, which can be any of the following:

`FLASH_MTD_REQ_READ`	Request to read a specified area of flash memory into a memory buffer
`FLASH_MTD_REQ_WRITE`	Request to write a memory buffer into a specified area of flash memory
`FLASH_MTD_REQ_ERASE`	Request to erase specified sectors of flash device
`FLASH_MTD_REQ_ERASE_ALL`	Request to erase the entire flash device
`FLASH_MTD_REQ_SPECIFIC`	Request to execute a device-specific operation - Operation code and a single operation parameter are passed through the additional parameter.
`FLASH_MTD_REQ_OPEN`	Request to open the flash device
`FLASH_MTD_REQ_CLOSE`	Request to close the flash device

Flash Virtual Base

The second parameter contains an address of the flash memory base in the kernel virtual space. The MTD should use it to convert relative flash addresses to absolute addresses in the kernel space. For example:

```
          /* Execute a flash command.
           */
    * (u_char *) ( flash_base + 0x5555 * 4 + chip ) = 0xAA;
    * (u_char *) ( flash_base + 0x2AAA * 4 + chip ) = 0x55;
    * (u_char *) ( flash_base + 0x5555 * 4 + chip ) = cmd;
```

Operation Parameter

The third parameter is supplementary to the operation code parameter and contains a description of the requested operation. The following type is used:

```
typedef union flash_mtd_param_u
{
   struct {
     void *           buf;          /* Data Buffer        */
     flash_mtd_area_t area;         /* Flash Area         */
   } rw;
   struct {
     u_int            start_sector; /* First Sector       */
     u_int            sectors_num;  /* Number of Sectors  */
   }erase;
   struct {
     int              req_code;     /* Request Code       */
     void *           req_param;    /* Request Parameters */
   }specific;
}
flash_mtd_param_t;
```

The **`rw`** structure is used when the operation code is equal to either `FLASH_MTD_REQ_READ or FLASH_MTD_REQ_WRITE`. The `buf` pointer specifies a data buffer in memory. The `area` structure specifies an area in flash. Offset is relative to the flash base.

The `erase` structure is used when the operation code is equal to `FLASH_MTD_REQ_ERASE`. The `start_sector` field specifies the first sector to erase. The `sectors_num` field specifies number of sectors to erase.

The `specific` structure is used when the operation code is equal to `FLASH_MTD_REQ_SPECIFIC`. The `req_code` field contains an MTD-specific operation code. The `req_param` field is a pointer to an operation specific parameters area.

`FLASH_MTD_REQ_OPEN` and `FLASH_MTD_REQ_CLOSE` do not use this argument.

MTD-Specific Data

The fourth parameter is a pointer to the MTD specific data. `flash_mgr` sets this parameter to the value passed by the MTD through the `user_param` field of the MTD registration data.

Return Code

If an operation completes successfully, the callback routine returns zero. Any other value indicates a failure.

Synchronization

The `flash_mgr` layer resolves all synchronization issues prior to invoking an MTD callback routine. An MTD is guaranteed that:

- Invocation of `FLASH_MTD_REQ_WRITE` is delayed until the MTD has finished all operations in progress.
- Invocation of `FLASH_MTD_REQ_ERASE` or `FLASH_MTD_REQ_ERASE_ALL` is delayed until the MTD has finished all operations in progress.
- Invocation of `FLASH_MTD_REQ_SPECIFIC` is delayed until the MTD has finished all operations in progress.
- Invocation of `FLASH_MTD_REQ_READ` is delayed until the MTD has finished write, erase or device-specific operations in progress.

CHAPTER 11

Writing PC Card Client Drivers

The LynxOS PC Card package architecture is based upon the PCMCIA/JEIDA PC Card Architecture specification. The following figure illustrates the card package architecture.

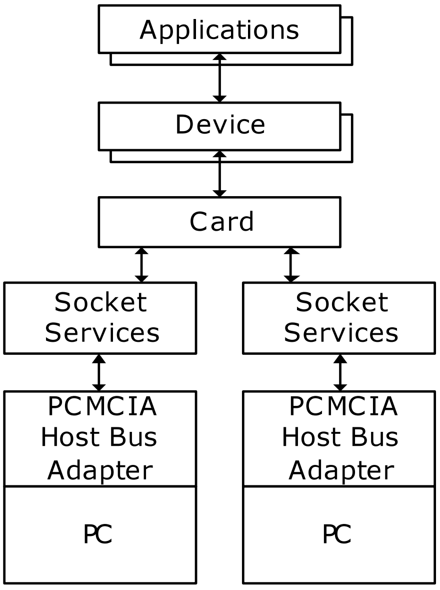


Figure 11-1: PC Card Architecture

Socket Services provides a standardized interface to manipulate PC Cards, sockets and adapters. A host system may have more than one PC Card adapter present. Each adapter has its own Socket Services instance.

Each instance of Socket Services registers with Card Services and notifies it about status changes in PC Cards or sockets.

By making all accesses to adapters, sockets, and PC Cards through the Socket Services interface, higher-level software is unaffected by different implementations of the hardware. Only the hardware-specific Socket Services code needs to modified to accommodate a new hardware implementation.

Card Services coordinates access to PC Cards, sockets and system resources among multiple clients. There is only one instance of Card Services in the system. Card Services makes all access to the hardware level through the Socket Services interface. All Socket Services status change reporting is routed to Card Services. Card Services then notifies the appropriate clients. Card Services preserves for its clients an abstract, hardware-independent view of a card and its resources.

Client Device Drivers refers to all users of Card Services. In the LynxOS PC Card architecture, users of Card Services are device drivers. They use a standardized API to access Card Services.

Card Services Overview

Card Services Initialization

Card Services is implemented as a device driver. There is only one instance of Card Services present in the system. At installation time, Card Services initializes its resource databases and prepares to handle registration requests from both Socket Services and client device drivers. The Card Services driver must be installed prior to any other PC Card-related driver.

Logical Sockets

The Card Services interface uses logical socket numbers to identify the socket a service is intended to access. The first physical socket on the first physical adapter is logical socket 0. The last logical socket is the total number of sockets less one.

Card Services Groups

The services defined by the card services interface can be divided into six functional groups. These are:

- Client Services
- Client Utilities
- Resource Management Services
- Advanced Client Services
- Bulk Memory Services
- Special Services

Client Services

The following services are used for client initialization and registration.

- `DeregisterClient`
- `GetCardServicesInfo`
- `RegisterClient`

Client Utilities

The following services are used to perform common tasks required by all clients.

- `GetFirstTuple`
- `GetNextTuple`

Resource Management Services

The following services provide basic access to available system resources. These services combine knowledge of the current status of system resources with the underlying Socket Services adapter control services.

- `GetConfigurationInfo`
- `GetFirstWindow` (not supported)
- `GetNextWindow` (not supported)
- `ReleaseConfiguration`
- `ReleaseIO`
- `ReleaseIRQ`
- `ReleaseWindow` (not supported)

- `RequestConfiguration`
- `RequestIO`
- `RequestIRQ`
- `RequestWindow` (not supported)

Advanced Client Services

The following are advanced client services.

- `AccessConfigReg`
- `GetFirstClient`
- `GetNextClient`
- `RegisterMTD` (not supported)

Bulk Memory Services

The following services provide various memory operations for memory clients that require isolation from the details of underlying memory technology hardware.

- `CloseMemory` (not supported)
- `CopyMemory` (not supported)
- `OpenMemory` (not supported)
- `ReadMemory` (not supported)
- `WriteMemory` (not supported)

Special Services

The following are services providing miscellaneous operations for client drivers.

- `ErrorName`
- `ParseTuple`
- `ServiceName`

Card Services Calling Conventions

The Card Services interface consists of the `CardServices()` function. `CardServices()` provides the API for all card services.

Header Files

All prototypes and constants needed to access card services from client device drivers are located in the following header files:

Table 11-1: Card Services Header Files

File	Description
`/sys/dheaders/pcmcia_cs.h`	Contains definitions common for all services.
`/sys/dheaders/pcmcia_cs_tuple.h`	Contains definitions required by the tuple parsing services.

Synopsis

The prototype for `CardServices()` is:

```
int CardServices( int Service, void * Handle, void *  Pointer,
                  int rgLength, void * ArgPointer );
```

The `CardServices()` parameters are defined as follows:

`Service`	Specifies the service code. Services details are documented in “Card Services Reference” on page 217.
`Handle`	Is the client handle returned by the `RegisterClient` service. It is not used by any other service. The `RegisterClient` service places a new client handle in the location pointed to by this argument.
`Pointer`	Is a service-dependent value.
`ArgLength`	Is the size of the structure pointed to by `ArgPointer`.
`ArgPointer`	Is a pointer to service-dependent data.

After invocation, a service returns a completion code, unique for a particular service, as defined “Card Services Reference” on page 217.

Client Structure

A client device driver is a device driver that uses the API defined by `CardServices()`.

The following tasks are common for client device drivers:

- Detecting the presence of card services
- Registration with card services to receive event notifications
- Identification of a PC Card
- Requesting system resources for a PC Card
- Configuration of a PC Card
- Deregistration of the client

These steps are described below and are illustrated by a sample client device driver acting as a PC Card enabler.

Detecting the Presence of Card Services

The presence of card services is detected using the `GetCardServicesInfo` service. For example:

```
int code;
pcmcia_cs_information_t cs_info;

/* Get Card Services information. */
code = CardServices( GetCardServicesInfo, ( void * )0, ( void * )0,
                     sizeof( cs_info ), &cs_info );
if ( code != PCMCIA_CS_SUCCESS )
{
  return (char *)SYSERR;
}

  /* Detect the presence of Card Services. */
if ( cs_info.signature[0] != ’C’ || cs_info.signature[1] != ’S’ )
{
  return (char *)SYSERR;
}
```

Client Registration

Following initialization, a client registers itself with card services to specify the event notification it is to receive. The `RegisterClient` service is used to register a new client. At registration time, the client specifies its type, thus defining its priority for event notification. I/O clients are notified of events first, memory technology drivers are notified next, and memory clients are notified last. Also, the client provides the event mask that defines the events to be notified of. For example:

```
pcmcia_cs_register_t reg;
int code;

  /* I/O type client, create artificial insertion event
   * for all card inserted at the registration time.
   */
reg.attributes  = PCMCIA_CS_ATTR_IO | PCMCIA_CS_ATTR_INSERT;

  /* Receive notification of PCMCIA_CS_EVENT_REMOVAL and
   * PCMCIA_CS_EVENT_INSERTION events.
   */
reg.event_mask  = PCMCIA_CS_EVENT_REMOVAL | PCMCIA_CS_EVENT_INSERTION;

  /* No client specific data is needed.
   */
reg.client_data = ( void * )0;

  /* Now register. NULL in the Handle param will prevent Card Services
   * from returning a client handle.
   */
code = CardServices( RegisterClient, ( void * )0, ex_callback,
                     sizeof( reg ), &reg );
if ( code != PCMCIA_CS_SUCCESS )
{
  return SYSERR;}
```

Client Callback

Card services notifies clients of events through a single callback routine that is specified at client registration. A client receives notification of an event along with various event-specific data.

The prototype for `callback()` is:

```
int callback( int event, int socket, void * handle, void * buffer,
             pcmcia_cs_mtd_request_t * mtd_request, void * client_param );
```

where:

`event`	Is the event code. Events are listed and described in the table, “Events.”
`socket`	Is the logical socket with which the event is associated. `socket` is meaningful only for status change events.
`handle`	Is the client handle.

`buffer, mtd_request`	These arguments are used by memory technology drivers (not supported).
`client_param`	Is client-specific data provided at registration.

Events

The following events are supported:

Table 11-2: Events

Event	Syntax/Description
`PCMCIA_CS_EVENT_REG_COMPLETE`	`callback(PCMCIA_CS_EVENT_REG_COMPLETE, 0, handle, NULL, NULL, param);` This is the first and mandatory event each client receives as soon as registration is complete. This event arrives before `RegisterClient` service completion to ensure determinism of control flow.
`PCMCIA_CS_EVENT_INSERTION`	`callback(PCMCIA_CS_EVENT_INSERTION, socket, handle, NULL, NULL, param );` This event occurs when card services detects an operational PC Card in the logical socket `socket`. The callback receives an artificial `PCMCIA_CS_EVENT_INSERTION` event for all sockets that contain a PC Card if the client has registered with the `PCMCIA_CS_ATTR_INSERTION` attribute. The artificial insertion event arrives before `RegisterClient` service completion to ensure determinism of control flow.
`PCMCIA_CS_EVENT_REMOVAL`	`callback( PCMCIA_CS_EVENT_REMOVE, socket, handle, NULL, NULL, param );` This event occurs when card services detects removal of a PC Card from the logical socket `socket`.

PC Card Identification

When a client receives the `PCMCIA_CS_EVENT_INSERTION` event, it must first identify the just inserted PC Card. PC Card identification can be accomplished by using the card and manufacturer identification numbers present in the mandatory `MANFID` tuple. This tuple is parsed automatically by card services on card insertion and is provided to client in the data returned by the `GetConfigurationInfo` service. For example:

```
pcmcia_cs_config_info_t config_info;
int code;

  /* A part of callback code.
   * Use the socket number from callback arguments.
   */
config_info.socket = socket;
code = CardServices( GetConfigurationInfo, handle, (void *)0,
                     sizeof( config_info ), &config_info );
if ( code != PCMCIA_CS_SUCCESS )
{
  return SYSERR;
}

if ( config_info.manuf_code == 0x101 &&
     config_info.manuf_info == 0x589 )
{
  /* This is a 3COM 3c589x PC Card.
   */
}
```

An alternate method to identify the PC card is to use the client utilities services to find some other tuples that contain card identification information, and parse them using the `ParseTuple` service. For example:

```
pcmcia_cs_tuple_t      tuple;
pcmcia_cs_tuple_data_t tuple_data;
int                    code;

  /* Find mandatory Version1 tuple.
   */
tuple.socket = socket;
tuple.code = PCMCIA_CS_TUPLE_VERS_1;
code = CardServices( GetFirstTuple, handle, (void *)0,
                     sizeof( tuple ), &tuple );
if ( code == PCMCIA_CS_SUCCESS )
{
  code = CardServices( ParseTuple, handle, &tuple,
                       sizeof( tuple_data ), &tuple_data );
  if ( code != PCMCIA_CS_SUCCESS )
  {
    return SYSERR;
  }
  kkprintf( "CARD DETECTED: v%d.%d\n",
            tuple_data.vers_1.major, tuple_data.vers_1.minor );
  kkprintf( "               %s\n", tuple_data.vers_1.manf_name );
  kkprintf( "               %s\n", tuple_data.vers_1.prod_name );
  kkprintf( "               %s\n", tuple_data.vers_1.lot_info );
  kkprintf( "               %s\n", tuple_data.vers_1.prog_info );
}
```

PC Card Configuration

To configure a PC Card, the client must first reserve appropriate system resources (I/O range and optional IRQ) using the `RequestIO` and `RequestIRQ` services.

Reservation is necessary to prevent possible configuration conflicts with other client device drivers. For example:

```
pcmcia_cs_request_io_t  io;
pcmcia_cs_request_irq_t irq;
int                     code;

  /* Request I/O range 0x200-0x210.
   */
io.socket      = socket;
io.base_port1  = 0x200;
io.num_ports1  = 0x10;
io.attributes1 = 0;
io.base_port2  = 0;
io.num_ports2  = 0;
io.attributes2 = 0;

code = CardServices( RequestIO, handle, ( void * )0,
                     sizeof( io ), &io );
if ( code != PCMCIA_CS_SUCCESS )
{
  return SYSERR;
}

  /* Request IRQ 14.
   */
irq.socket       = socket;
irq.assigned_irq = 32 + 14;
code = CardServices( RequestIRQ, handle, ( void * )0,
                     sizeof(irq), &irq );
if ( code != PCMCIA_CS_SUCCESS )
{
  return SYSERR;
}
```

Actual configuration of a PC Card is performed by the `RequestConfiguration` service. It applies the specified voltage to the card and reserves the I/O address ranges and IRQ for corresponding socket. For example:

```
pcmcia_cs_request_config_t config;
int                        code;

  /* Apply the voltage and configure the card
  */
config.socket   = socket;
config.vcc      = 50;
config.vpp      = 50;
config.int_type = PCMCIA_CS_INTERFACE_IO;
config.option   = 1 + 0x40;
config.present  = PCMCIA_CS_PRESENT_OPTION;
code = CardServices( RequestConfiguration, handle, ( void * )0,
       sizeof( config ), &config );
if ( code != PCMCIA_CS_SUCCESS )
{
  return SYSERR;
}
```

Client Deregistration

To deregister a client, the `DeregisterClient` service is used. The client handle obtained using the `RegisterClient` service is passed to `CardServices()`. For example:

```
int code;

code = CardServices( DeregisterClient, handle, ( void * )0, 0, ( void * )0
);
if ( code != PCMCIA_CS_SUCCESS )
{
  return SYSERR;
}
```

Sample Client Drivers

PC Card Enabler

An enabler is a client device driver that detects insertion of a certain PC Card and configures it. The enabler allows standard device drivers to work with the PC Card as a conventional ISA device. The simplest enabler is a character device driver that has only `install` and `uninstall` entry point functions. At installation, the enabler detects the presence of card services and registers with it, requesting to receive card insertion events. Upon receiving a card insertion event, the enabler identifies the PC Card. If a supported card is inserted, the enabler configures it.

As soon as the PC Card configuration is complete, an appropriate dynamic device driver can be installed. The PC Card is controlled as if it were an ISA device.

If both enabler and driver are linked statically into the kernel, the enabler is installed before the driver. The card must be inserted prior to system boot and the driver should work without any changes.

NOTE: Note that you must ensure that the driver is configured to use the same IRQ and I/O ports range as used by the enabler to configure the card.

A sample driver layout is illustrated below:

`install`	• Detects card services presence.
	• Registers I/O type client with attributes shown in the following example:
	
	`/* I/O type client, create artificial insertion event`
	` * for all card inserted at the registration time.`
	` */`
	`reg.attributes  = PCMCIA_CS_ATTR_IO | PCMCIA_CS_ATTR_INSERT;`
	`reg.event_mask  = PCMCIA_CS_EVENT_INSERTION;`

`uninstall`	• Deregisters the client.
`callback`	• Upon receipt of `PCMCIA_CS_EVENT_INSERTION` event, identifies and configure the card.

Addition to Existing ISA Device Driver

It is possible to add the enabler code to an existing device driver of a conventional ISA card. This approach allows you to create a hot swap-capable device driver for the PC Card device. The driver can be statically or dynamically linked into the kernel.

A driver with an embedded enabler should return success at installation, whether it finds the card or not. However, it should be modified to reject any user accesses to the device until the enabler detects and configures the PC Card. As soon as the PC Card configuration is complete, the driver should perform the initialization procedure, which would have been otherwise performed at installation of the conventional device driver. Upon a request to remove the card, the driver should attempt to shut down the device.

A sample driver layout is illustrated below:

`install`	• Detect card services presence.
	• Register I/O type client with attributes shown in the following example:
	`/* I/O type client, create artificial insertion event`
	` * for all card inserted at the registration time.`
	` */`
	`reg.attributes  = PCMCIA_CS_ATTR_IO | PCMCIA_CS_ATTR_INSERT;`
	`reg.event_mask  = PCMCIA_CS_EVENT_INSERTION |`
	`                  PCMCIA_CS_EVENT_REMOVAL`
	• Do not perform any device initialization, to ensure proper operation if the card is not present in the socket.
	• Set up a flag ensuring that any user access to the device is rejected. The flag can be either a global variable or contained in the statics structure of the driver.

`uninstall`	• Deregister the client.
`callback`	• Upon `PCMCIA_CS_EVENT_INSERTION` event, identify and configure the card. Perform device initialization, then de-assert the access rejection flag. Upon `PCMCIA_CS_EVENT_REMOVAL` event, safely shut down the device, terminate the device driver operation, and set the access rejection flag again.
`open`, `read`,...	• If the access rejection flag is set, return `ENXIO`. Operate as usual otherwise.

Card Services Reference

This section provides a description of the services supported by Card Services. Card Services uses the `CardServices()` API for access to all services. `CardServices()` is defined as follows:

```
int CardServices( int Service, void * Handle, void *  Pointer,
                  int rgLength, void * ArgPointer );
```

where:

`Service`	Specifies the service code.
`Handle`	Is the client handle returned by the `RegisterClient` service. It is not used by any other service. The `RegisterClient` service places a new client handle in the location pointed to by this argument.

`Pointer`	Is a service-dependent value.
`ArgLength`	Is the size of the structure pointed to by `ArgPointer`.
`ArgPointer`	Is a pointer to service-dependent data.

AccessConfigReg

`AccessConfigReg` allows a client to read or write a PC Card Configuration Register. `ArgPointer` must be a pointer to a `pcmcia_cs_access_reg_t` structure.

The `pcmcia_cs_access_reg_t` structure is defined as follows:

```
typedef struct pcmcia_cs_access_reg_s
{
  int socket; /* logical socket                 */
  int action; /* action to be performed         */
  int offset; /* offset to status register      */
  unsigned char value;  /* value to read or write         */
}
pcmcia_cs_access_reg_t;
```

where:

`socket`	Is a logical socket.
`action`	Is the code of the action to be executed. `action` can be set to either `PCMCIA_CS_ACCESS_READ` or `PCMCIA_CS_ACCESS_WRITE`. If `action` is set to `PCMCIA_CS_ACCESS_WRITE`, `value` is written to the PC Card Configuration Register. If `action` is set to `PCMCIA_CS_ACCESS_READ`, `value` is set to the current value of the PC Card Configuration Register.
`offset`	This is the byte offset to the status register. This is relative to the PC Card configuration register base.
`value`	Contains the value to read or write.

EXAMPLE

```
pcmcia_cs_access_reg_t access_reg;

/* get the value of Configuration Option Register */

access_reg.socket = 0;
access_reg.action = PCMCIA_CS_ACCESS_READ;
access_reg.offset = 0;
res = CardServices( AccessConfigReg, NULL, NULL, sizeof( access_reg ),
      &access_reg );
if ( res == PCMCIA_CS_SUCCESS )
```

```
{
  cprintf( "Configuration Option Register: 0x%x\n", access_reg.value );
}
```

RETURN CODES

`PCMCIA_CS_SUCCESS`	Request succeeded.
`PCMCIA_CS_BAD_SOCKET`	Specified socket is invalid.
`PCMCIA_CS_NO_CARD`	No PC Card in the socket.
`PCMCIA_CS_BAD_ARG_LENGTH`	`ArgLength` is invalid.
`PCMCIA_CS_BAD_ARGS`	Specified arguments are invalid.
`PCMCIA_CS_UNSUPPORTED_SERVICE`	Service is not supported.

DeregisterClient

The `DeregisterClient` service removes a client from the list of registered clients maintained by card services. The client handle is passed in the `Handle` parameter.

EXAMPLE

```
res = CardServices( DeregisterClient, NULL, NULL, 0, NULL );
name.code = res;
CardServices( ErrorName, NULL, NULL, 0, &name );

if ( res == PCMCIA_CS_SUCCESS )
{
cprintf( “I am not a client anymore\n” );
}
```

RETURN CODES

`PCMCIA_CS_SUCCESS`	Request succeeded.
`PCMCIA_CS_BAD_HANDLE`	Client handle is invalid.
`PCMCIA_CS_UNSUPPORTED_SERVICE`	Service is not supported.

ErrorName

`ErrorName` service returns a character string corresponding to a specified error code returned previously by Card Services. `ArgPointer` must be a pointer to a `pcmcia_cs_name_t` structure.

The `pcmcia_cs_name_t` structure is defined as follows:

```
typedef struct pcmcia_cs_name_s
{
  int    code;
  char * name;
}
pcmcia_cs_name_t;
```

where:

`code`	Is the error code.
`name`	Is a pointer to the string that contains the error name.

EXAMPLE

```
pcmcia_cs_name_t;

res = CardServices ( DeregisterClient, NULL, NULL, 0, NULL );
name.code = res;
CardServices( ErrorName, NULL, NULL, 0, &name );

/* should print PCMCIA_CS_BAD_HANDLE */
cprintf( “code 0x%x, name %s\n”, name.code, name.name );
```

RETURN CODES

`PCMCIA_CS_SUCCESS`	Request succeeded.
`PCMCIA_CS_UNSUPPORTED_SERVICE`	Service is not supported.

GetCardServicesInfo

`GetCardServicesInfo` service returns the number of installed logical sockets and information about Card Services that includes the vendor revision number and release compliance code. `ArgPointer` must be a pointer to a `pcmcia_cs_information_t` structure.

The `pcmcia_cs_information_t` structure is defined as follows:

```
typedef struct pcmcia_cs_information_s
{
  char   signature[2];  /* "CS"                      */
  int    count;         /* number of sockets         */
  int    revision;      /* BCD value of CS revision */
  short  cs_level;      /* BCD value of CD release  */
  char * vendor_string; /* vendor string            */
}
pcmcia_cs_information_t;
```

where:

`signature[]`	Contains ‘`CS`’ if card services is installed.
`count`	Is the number of logical sockets.
`revision`	Is the binary coded decimal (BCD) value of the `CardServices()` revision.
`cs_level`	Is the BCD value of the `CardServices()` release.
`vendor_string`	Is a vendor-specific string.

EXAMPLE

```
pcmcia_cs_information_t info;

res = CardServices( GetCardServicesInfo, NULL, NULL, sizeof( info ), &info
);

if ( res == PCMCIA_CS_SUCCESS )
{
  if ( info.signature[0] == ’C’ &&
       info.signature[1] == ’S’ )
  {
    cprintf( "Card Services detected!\n" );
  }
}
```

RETURN CODES

`PCMCIA_CS_SUCCESS`	Request succeeded.
`PCMCIA_CS_UNSUPPORTED_SERVICE`	Service is not supported.
`PCMCIA_CS_BAD_ARG_LENGTH`	`ArgLength` is invalid.

GetConfigurationInfo

`GetConfigurationInfo` service returns information about the specified socket and PC Card installed in the socket. `ArgPointer` must be a pointer to a `pcmcia_cs_config_info_t` structure.

The `pcmcia_cs_config_info_t` structure is defined as follows:

```
typedef struct pcmcia_cs_config_info_s
{
  int           socket;      /* logical socket                        */
  int           attributes;  /* bit-mapped attributes                 */
  int           vcc;         /* Vcc settings                          */
  int           vpp;         /* Vpp settings                          */
  int           int_type;    /* interface type                        */
```

```
    unsigned      config_base; /* base address of config register      */
    unsigned      manuf_code;  /* from Manufacturer ID tuple           */
    unsigned      manuf_info;  /* from Manufacturer ID tuple           */
    unsigned      base_port1;  /* base address for a range             */
    int           num_ports1;  /* number of contiguous ports           */
    int           attributes1; /* bit-mapped port attributes           */
    unsigned      base_port2;  /* base address for a range             */
    int           num_ports2;  /* number of contiguous ports           */
    int           attributes2; /* bit-mapped port attributes           */
    int           assigned_irq;/* irq assigned to PC Card              */
    unsigned char status;      /* Card Status register settings        */
    unsigned char pin;         /* Card Pin register settings           */
    unsigned char copy;        /* Card Copy register settings          */
    unsigned char option;      /* Card Option register settings        */
    int           present;     /* Card Configuration registers present */
  }
  pcmcia_cs_config_info_t;
```

where:

`socket`	Is the logical socket.
`attributes`	Is the bit mapped socket attributes. `attributes` is a bitwise combination of the constants: • `PCMCIA_CS_SATTR_ON` - The socket contains a PC Card. • `PCMCIA_CS_SATTR_CONFIGURED` - The PC Card installed in the socket has been configured using the `RequestConfiguration` service.
`vcc`	Is the voltage applied to the Vcc pin of a PC Card. The voltage is expressed in tenths of a volt.
`vpp`	Is the voltage applied to the Vpp pins of a PC Card. The voltage is expressed in tenths of a volt.
`int_type`	Interface type. Must be set to: • `PCMCIA_CS_INTERFACE_NONE` - For the simplest interface, featuring only card detection • `PCMCIA_CS_INTERFACE_MEM` - For a memory only interface • `PCMCIA_CS_INTERFACE_IO` - For a memory and I/O interface
`config_base`	Is the card base address of the configuration registers area.
`manuf_code`	Is the manufacturer number from the `MANFID` tuple.
`manuf_info`	Is the product identification number from the `MANFID` tuple.
`base_port1`	Is the base port number for I/O window 1.
`num_ports1`	Is the number of contiguous ports in I/O window 1.
`attributes1`	If set to `PCMCIA_CS_IO_ATTR_8BIT`, the I/O window has an 8 bit width.

base_port2	Is the base port number for I/O window 2.
num_ports2	Is the number of contiguous ports in I/O window 2.
attributes2	If set to PCMCIA_CS_IO_ATTR_8BIT, the I/O window has an 8 bit width.
assigned_irq	Is the IRQ assigned to the PC Card.
status	Is the card status register settings, if present.
pin	Is the card pin register settings, if present.
copy	Is the card socket/copy register settings, if present.
option	Is the card option register settings, if present.
present	Specifies if the card configuration registers are present. A bitwise combination of the following constants: • PCMCIA_CS_PRESENT_STATUS - Status register is present. • PCMCIA_CS_PRESENT_PIN - Pin register is present. • PCMCIA_CS_PRESENT_COPY - Copy register is present. • PCMCIA_CS_PRESENT_OPTION - Option register is present.

EXAMPLE

```
pcmcia_cs_config_info_t info;

info.socket = 0;
res = CardServices( GetConfigurationInfo, NULL, NULL, sizeof( info ),
      &info );
if ( res == PCMCIA_CS_SUCCESS )
{
  if ( info.attributes & PCMCIA_CS_SATTR_ON )
{
  cprintf( "Socket #0 contains a PC Card!\n" );
}
}
```

RETURN CODES

PCMCIA_CS_SUCCESS	Request succeeded.
PCMCIA_CS_UNSUPPORTED_SERVICE	Service is not supported.
PCMCIA_CS_BAD_ARG_LENGTH	ArgLength is invalid.
PCMCIA_CS_BAD_SOCKET	Specified socket is invalid.

GetFirstTuple

The `GetFirstTuple` service returns the first tuple of the specified type in the CIS for the specified socket. `ArgPointer` must be a pointer to a `pcmcia_cs_tuple_t` structure.

The `pcmcia_cs_tuple_t` structure is defined as follows:

```
typedef struct pcmcia_cs_tuple_s
{
  int socket;    /* Socket id            */
  int code;      /* Requested tuple code */
  pcmcia_cs_tuple_internal_t internal;  /* Internal state       */
}
pcmcia_cs_tuple_t;
```

where:

`socket`	Is the logical socket.
`code`	Is the tuple code. See table below.
`internal`	Is the Card Services CIS state information. This field is used internally by Card Services.

The following predefined constants can be used to specify a tuple code:

`PCMCIA_CS_TUPLE_DEVICE`	(0x01)
`PCMCIA_CS_TUPLE_INDIRECT`	(0x03)
`PCMCIA_CS_TUPLE_CONFIG_CB`	(0x04)
`PCMCIA_CS_TUPLE_CFTABLE_ENTRY_CB`	(0x05)
`PCMCIA_CS_TUPLE_LONGLINK_MFC`	(0x06)
`PCMCIA_CS_TUPLE_BAR`	(0x07)
`PCMCIA_CS_TUPLE_CHECKSUM`	(0x10)
`PCMCIA_CS_TUPLE_VERS_1`	(0x15)
`PCMCIA_CS_TUPLE_ALTSTR`	(0x16)
`PCMCIA_CS_TUPLE_DEVICE_A`	(0x17)
`PCMCIA_CS_TUPLE_JEDEC_C`	(0x18)
`PCMCIA_CS_TUPLE_JEDEC_A`	(0x19)
`PCMCIA_CS_TUPLE_CONFIG`	(0x1A)
`PCMCIA_CS_TUPLE_CFTABLE_ENTRY`	(0x1B)

PCMCIA_CS_TUPLE_DEVICE_OC	(0x1C)
PCMCIA_CS_TUPLE_DEVICE_OA	(0x1D)
PCMCIA_CS_TUPLE_DEVICEGEO	(0x1E)
PCMCIA_CS_TUPLE_DEVICEGEO_A	(0x1F)
PCMCIA_CS_TUPLE_MANFID	(0x20)
PCMCIA_CS_TUPLE_FUNCID	(0x21)
PCMCIA_CS_TUPLE_FUNCE	(0x22)
PCMCIA_CS_TUPLE_SWIL	(0x23)
PCMCIA_CS_TUPLE_VERS_2	(0x40)
PCMCIA_CS_TUPLE_FORMAT	(0x41)
PCMCIA_CS_TUPLE_GEOMETRY	(0x42)
PCMCIA_CS_TUPLE_BYTEORDER	(0x43)
PCMCIA_CS_TUPLE_DATE	(0x44)
PCMCIA_CS_TUPLE_BATTERY	(0x45)
PCMCIA_CS_TUPLE_ORG	(0x46)
PCMCIA_CS_TUPLE_FORMAT_A	(0x47)
PCMCIA_CS_TUPLE_SPCL	(0x90)

EXAMPLE

```
pcmcia_cs_tuple_t tuple;

tuple.socket = 0;
tuple.code = PCMCIA_CS_TUPLE_VERS_1;
res = CardServices( GetFirstTuple, NULL, NULL, sizeof( tuple ), &tuple );

if ( res == PCMCIA_CS_SUCCESS )
{
  cprintf( "Vers1 tuple found\n" );
}
```

RETURN CODES

PCMCIA_CS_SUCCESS	Request succeeded.
PCMCIA_CS_BAD_SOCKET	Specified socket is invalid.
PCMCIA_CS_UNSUPPORTED_SERVICE	Service is not supported.

`PCMCIA_CS_BAD_ARG_LENGTH`	`ArgLength` is invalid.
`PCMCIA_CS_NO_CARD`	No PC Card in socket.
`PCMCIA_CS_NO_MORE_ITEMS`	No tuples with specified code.

GetNextTuple

The `GetNextTuple` service returns the next tuple of the specified type in the CIS for the specified socket. `ArgPointer` must be a pointer to the `pcmcia_cs_tuple_t` structure returned by a `GetFirstTuple` or a previous `GetNextTuple` request.

EXAMPLE

```
pcmcia_cs_tuple_t tuple;

tuple.socket = 0;
tuple.code = PCMCIA_CS_TUPLE_VERS_1;
res = CardServices( GetFirstTuple, NULL, NULL, sizeof( tuple ), &tuple );

if ( res == PCMCIA_CS_SUCCESS )
{
  cprintf( "Vers1 tuple found\n" );
}

res = CardServices( GetNextTuple, NULL, NULL, sizeof( tuple ), &tuple );

if ( res == PCMCIA_CS_SUCCESS )
{
  cprintf( "Another Vers1 tuple found\n" );
}
```

RETURN CODES

`PCMCIA_CS_SUCCESS`	Request succeeded.
`PCMCIA_CS_BAD_SOCKET`	Specified socket is invalid.
`PCMCIA_CS_UNSUPPORTED_SERVICE`	Service is not supported.
`PCMCIA_CS_BAD_ARG_LENGTH`	`ArgLength` is invalid.
`PCMCIA_CS_NO_CARD`	No PC Card in socket.
`PCMCIA_CS_NO_MORE_ITEMS`	No tuples with specified code.

ParseTuple

The `ParseTuple` service parses a tuple. `Pointer` must be a pointer to the `pcmcia_cs_tuple_t` structure returned by a `GetFirstTuple` or `GetNextTuple` request. The `ParseTuple` service fills a `pcmcia_cs_tuple_data_t` structure pointed to by the `ArgPointer` parameter. The `pcmcia_cs_tuple_data_t` structure is defined as follows:

```
typedef union pcmcia_cs_tuple_data_u
{
  pcmcia_cs_tuple_device_t        device;
  pcmcia_cs_tuple_bar_t           bar;
  pcmcia_cs_tuple_checksum_t      checksum;
  pcmcia_cs_tuple_vers_1_t        vers_1;
  pcmcia_cs_tuple_altstr_t        altstr;
  pcmcia_cs_tuple_jedec_t         jedec;
  pcmcia_cs_tuple_config_t        config;
  pcmcia_cs_tuple_cftable_entry_t entry;
  pcmcia_cs_tuple_device_o_t      device_o;
  pcmcia_cs_tuple_devicegeo_t     devicegeo;
  pcmcia_cs_tuple_manfid_t        manfid;
  pcmcia_cs_tuple_funcid_t        funcid;
  pcmcia_cs_tuple_funce_t         funce;
  pcmcia_cs_tuple_swil_t          swil;
  pcmcia_cs_tuple_vers_2_t        vers_2;
  pcmcia_cs_tuple_format_t        format;
  pcmcia_cs_tuple_geometry_t      geo;
  pcmcia_cs_tuple_byteorder_t     order;
  pcmcia_cs_tuple_date_t          date;
  pcmcia_cs_tuple_battery_t       battery;
  pcmcia_cs_tuple_org_t           org;
  pcmcia_cs_tuple_spcl_t          spcl;}
pcmcia_cs_tuple_data_t;
```

Supported Tuple Codes

This section lists the supported tuple codes and field sets that support each. For detailed information about each field semantics, refer to the PCMIA\JEIDA Metaformat Specification.

`PCMCIA_CS_TUPLE_DEVICE`

`PCMCIA_CS_TUPLE_DEVICE_A`

```
typedef struct pcmcia_cs_tuple_device_s
{
  int ndev;           /* Number of device structs   */
  struct
  {
    u_char type;      /* Type of device             */
    u_char wps;       /* Write protect switch       */
    u_char units;     /* Number of mem units        */
    u_long unit_size; /* Mem unit’s size            */
    u_long speed;     /* Card’s speed               */
    u_long size;      /* Full size of mem           */
  }
  dev[MAX_DEVICES];
}
pcmcia_cs_tuple_device_t;
```

PCMCIA_CS_TUPLE_BAR

```
typedef struct pcmcia_cs_tuple_bar_s
{
  struct
  {
    u_int below     : 1; /* Below 1 Mb bit             */
    u_int cache     : 2; /* Prefetchable/Cacheable     */
    u_int addr_spc  : 1; /* Address space bit          */
    u_int indicator : 3; /* Address space indicator    */
  }
  attr;                  /* Attributes                 */
  u_long size;           /* Base Address Register size */
}
pcmcia_cs_tuple_bar_t;
```

PCMCIA_CS_TUPLE_CHECKSUM

```
typedef struct pcmcia_cs_tuple_checksum_s
{
  u_short address;   /* Checksumed address         */
  u_short length;    /* Length of checksumed space */
  u_char  checksum;  /* Checksum                   */
}
pcmcia_cs_tuple_checksum_t;
```

PCMCIA_CS_TUPLE_VERS_1

```
typedef struct pcmcia_cs_tuple_vers_1_s
{
  u_char major;                      /* Major version          */
  u_char minor;                      /* Minor version          */
  char   manf_name[MAX_STRING_LEN]; /* Manufacturer name      */
  char   prod_name[MAX_STRING_LEN]; /* Product name           */
  char   lot_info[MAX_STRING_LEN];  /* Lot number             */
  char   prog_info[MAX_STRING_LEN]; /* Programming conditions */
}
pcmcia_cs_tuple_vers_1_t;
```

PCMCIA_CS_TUPLE_ALTSTR

```
typedef struct pcmcia_cs_tuple_altstr_s
{
  char escape[MAX_ESCAPE_SEQ_LEN]; /* Alternative string         */
                                   /* escape sequence            */
  char altstrings[MAX_ALTSTRINGS][MAX_STRING_LEN];
                                   /* Strings in alternative     */
                                   /* language (corresponding to */
                                   /* vers_1 or vers_2 tuples)   */
}
pcmcia_cs_tuple_altstr_t;
```

PCMCIA_CS_TUPLE_JEDEC
PCMCIA_CS_TUPLE_JEDEC_A

```
typedef struct pcmcia_cs_tuple_jedec_s
{
  int ndev;            /* Number of device structs   */
```

```
                      /* (corresponding to last     */
                      /* device tuple)              */
  struct
  {
    u_char manf;      /* Manufacturer number        */
    u_char units;     /* Number of mem units        */
    u_long unit_size; /* Mem unit's size            */
    u_long size;      /* Full size of used mem      */
  }
  dev[MAX_DEVICES];
}
pcmcia_cs_tuple_jedec_t;
```

PCMCIA_CS_TUPLE_CONFIG

```
typedef struct pcmcia_cs_tuple_config_s
{
  struct
  {
    u_int mask_size : 4; /* Size of Configuration      */
                         /* Registers presence mask    */
                         /* field in bytes (minus 1)   */
    u_int addr_size : 2; /* Size of Configuration      */
                         /* Registers base address     */
                         /* field in bytes (minus 1)   */
  }
  sizes;
  u_char index;          /* Last index                 */
  u_long base;           /* Base address               */
  u_char mask[16];       /* Conf. Reg. presence masks  */
  u_char subtuple[MAX_DATA_SIZE];
                         /* Optional additional data   */
  u_char size;           /* Size of subtuple data      */
}
pcmcia_cs_tuple_config_t;
```

PCMCIA_CS_TUPLE_CFTABLE_ENTRY

```
typedef struct pcmcia_cs_tuple_power_s
{
  struct
  {
    u_int pdown : 1;  /* Power down bit             */
    u_int peak  : 1;  /* Peak current bit           */
    u_int avg   : 1;  /* Average current bit        */
    u_int stat  : 1;  /* Static current bit         */
    u_int max_v : 1;  /* Maximum voltage bit        */
    u_int min_v : 1;  /* Minimum voltage bit        */
    u_int nom_v : 1;  /* Nominal voltage bit        */
  }
  select;
  struct
  {
    u_long value;     /* Value corresponding to     */
                      /* field in select structure  */
    u_char flags;     /* Flags corresponding to     */
                      /* field in select structure  */
  }
  values[7];
}
pcmcia_cs_tuple_power_t;
```

```
typedef struct pcmcia_cs_tuple_timing_s
{
  u_long  wait;         /* Max Wait time   */
  u_long  ready;        /* Max Ready time  */
  u_long  reserved;     /* reserved        */
  u_char  waitscale;    /* Wait scale      */
  u_char  rdyscale;     /* Ready scale     */
  u_char  rsvscale;     /* Reserved scale  */
  struct
  {
    u_int wait     : 1; /* Wait bit        */
    u_int ready    : 1; /* Ready bit       */
    u_int reserved : 1; /* Reserved bit    */
  }
  select;
}
pcmcia_cs_tuple_timing_t;

typedef struct pcmcia_cs_tuple_iospace_s
{
  struct
  {
    u_int range       : 1; /* Range bit                */
    u_int bus8_16     : 2; /* Bus width info           */
    u_int ioaddrlines : 5; /* Total number of address */
                           /* lines                    */
  }
  iospace_desc;
  struct
  {
    u_int len_size    : 2; /* Size of length field    */
    u_int addr_size   : 2; /* Size of address field   */
    u_int num_fields  : 4; /* Total number of range   */
                           /* fields (minus 1)        */
  }
  range_desc;
  struct
  {
    u_long base;          /* Start of the next I/O    */
                          /* Block                    */
    u_long length;        /* Length of the next I/O   */
                          /* Block                    */
  }
  range[16];
}
pcmcia_cs_tuple_iospace_t;

typedef struct pcmcia_cs_tuple_irq_s
{
  struct
  {
    u_int share : 1; /* Share bit              */
    u_int pulse : 1; /* Pulse bit              */
    u_int level : 1; /* Level bit              */
    u_int mask  : 1; /* Mask bit               */
    u_int vend  : 1; /* Vendor specific signal */
    u_int berr  : 1; /* Bus error signal       */
    u_int iock  : 1; /* I/O check signal       */
    u_int nmi   : 1; /* Non-maskable interrupt */
    u_int irqn  : 4; /* One of possible lines  */
  }
  irq_desc;
  u_short irq_mask;  /* IRQ lines mask         */
```

```
}
pcmcia_cs_tuple_irq_t;

typedef struct pcmcia_cs_tuple_mem_s
{
  struct
  {
    u_int host_addr  : 1; /* Host address bit           */
    u_int caddr_size : 2; /* Size of card address       */
    u_int len_size   : 2; /* Size of length             */
    u_int windows    : 3; /* The number of window       */
                          /* descriptors (minus 1)      */
  }
  mem_desc;
  struct
  {
    u_long length;        /* The length of the window   */
                          /* in units of 256 bytes      */
    u_long card_addr;     /* The address to be accessed */
                          /* on the card corresponding  */
                          /* to the host address        */
    u_long host_addr;     /* The physical address in    */
                          /* the host-address space     */
                          /* where the block of memory  */
                          /* must be placed             */
  }
  window[8];
}
pcmcia_cs_tuple_mem_t;

typedef struct pcmcia_cs_tuple_misc_s
{
  u_int pdown     : 1; /* Power down bit           */
  u_int read_only : 1; /* Read only bit            */
  u_int audio     : 1; /* Audio bit                */
  u_int max_twins : 3; /* Max Twin cards (minus 1) */
  u_int dma_width : 1; /* The DMA data transfer    */
                       /* width                    */
  u_int dma_req   : 2; /* DMA request signal       */
}
pcmcia_cs_tuple_misc_t;

typedef struct pcmcia_cs_tuple_cftable_entry_s
{
  struct
  {
    u_int interface  : 1; /* Interface bit              */
    u_int dflt       : 1; /* Default bit                */
    u_int entry_num  : 6; /* Value is to be written to  */
                          /* the Card Configuration     */
                          /* Register to enable the     */
                          /* configuration described in */
                          /* the tuple                  */
  }
  index;
  struct
  {
    u_int wait_req   : 1; /* WAIT# Signal support       */
                          /* required for Memory Cycles */
    u_int rdy_active : 1; /* READY Status Active        */
    u_int wp_active  : 1; /* Write Protect Status is    */
                          /* active                     */
    u_int bvd_active : 1; /* BVD1 and BVD2 signals are  */
```

```
                            /* active                      */
      u_int type       : 4; /* Interface type              */
    }
    interface;
    struct
    {
      u_int misc       : 1; /* Miscell. structure bit      */
      u_int memspace   : 2; /* Memspace structure          */
                            /* descriptor                  */
      u_int irq        : 1; /* IRQ structure bit           */
      u_int iospace    : 1; /* IO space structure bit      */
      u_int timing     : 1; /* Timing structure bit        */
      u_int power      : 2; /* Number of power structures */
    }
    select;
    pcmcia_cs_tuple_power_t   power[3];
                            /* Power structures            */
    pcmcia_cs_tuple_timing_t  timing;
                            /* Timing structure            */
    pcmcia_cs_tuple_iospace_t iospace;
                            /* IO space structure          */
    pcmcia_cs_tuple_irq_t     irq;
                            /* IRQ structure               */
    pcmcia_cs_tuple_mem_t     mem;
                            /* Memspace structure          */
    pcmcia_cs_tuple_misc_t    misc;
                            /* Miscellaneous structure     */
    u_char                    subtuple[MAX_DATA_SIZE];
                            /* Optional additional data    */
    u_char                    size;
  /* Size of subtuple data      */
  }
  pcmcia_cs_tuple_cftable_entry_t;
```

PCMCIA_CS_TUPLE_DEVICE_OC
PCMCIA_CS_TUPLE_DEVICE_OA

```
  typedef struct pcmcia_cs_tuple_device_o_s
  {
    int ndev;             /* Number of device structs */
    struct
    {
      u_int vcc_used : 2; /* Vcc voltage              */
      u_int mwait    : 1; /* MWait bit                */
    }
    info;
    struct
    {
      u_char type;        /* Type of device           */
      u_char wps;         /* Write protect switch     */
      u_char units;       /* Number of mem units      */
      u_long unit_size;   /* Mem unit’s size          */
      u_long speed;       /* Card’s speed             */
      u_long size;        /* Full size of mem         */
    }
    dev[MAX_DEVICES];
  }
  pcmcia_cs_tuple_device_o_t;
```

PCMCIA_CS_TUPLE_DEVICEGEO
PCMCIA_CS_TUPLE_DEVICEGEO_A

```
typedef struct pcmcia_cs_tuple_devicegeo_s
{
  int ndev;           /* Number of device structs */
  struct
  {
    u_char bus;       /* Card interface width     */
                      /* (2^(bus - 1))            */
    u_char erase;     /* Minimum erase block      */
                      /* size (2^(erase - 1))     */
    u_char read;      /* Minimum read block       */
                      /* size (2^(read - 1))      */
    u_char write;     /* Minimum write block      */
                      /* size (2^(write - 1))     */
    u_char partition; /* Minimal partition size   */
                      /* (2^(partition - 1))      */
    u_char hwil;      /* Hardware interleave      */
                      /* (2^(hwil - 1))           */
  }
  dev[MAX_DEVICES];
}
pcmcia_cs_tuple_devicegeo_t;
```

PCMCIA_CS_TUPLE_MANFID

```
typedef struct pcmcia_cs_tuple_manfid_s
{
  u_short code; /* Manufacture code */
  u_short card; /* Card info        */
}
pcmcia_cs_tuple_manfid_t;
```

PCMCIA_CS_TUPLE_FUNCID

```
typedef struct pcmcia_cs_tuple_funcid_s
{
  u_int func : 8; /* Type of card */
  u_int rom  : 1; /* ROM bit      */
  u_int post : 1; /* POST bit     */
}
pcmcia_cs_tuple_funcid_t;
```

PCMCIA_CS_TUPLE_FUNCE

```
typedef struct pcmcia_cs_tuple_funce_s
{
  u_char type;                /* Type of extended data */
  u_char data[MAX_DATA_SIZE]; /* Function information  */
  u_char size;                /* Size of extended data */
}
pcmcia_cs_tuple_funce_t;
```

PCMCIA_CS_TUPLE_SWIL

```
typedef struct pcmcia_cs_tuple_swil_s
{
  u_char interleave; /* Interleave factor */
}
pcmcia_cs_tuple_swil_t;
```

PCMCIA_CS_TUPLE_VERS_2

```
typedef struct pcmcia_cs_tuple_vers_2_s
```

```
{
  u_char  version;    /* Structure version       */
  u_char  comply;     /* Level of compliance     */
  u_short first_addr; /* Byte address of first   */
                      /* data byte in card       */
  u_char  rsv1;       /* Reserved                */
  u_char  rsv2;       /* Reserved                */
  u_char  vendor1;    /* Vendor specific         */
  u_char  vendor2;    /* Vendor specific         */
  u_char  copies;     /* Number of copies of CIS */
                      /* present on the device   */
  u_char  soft_vend[MAX_STRING_LEN];
                      /* Vendor of software that */
                      /* formatted the card      */
  u_char  card_info[MAX_STRING_LEN];
                      /* Informational message   */
                      /* about the card          */
pcmcia_cs_tuple_vers_2_t;
```

PCMCIA_CS_TUPLE_FORMAT
PCMCIA_CS_TUPLE_FORMAT_A

```
typedef struct pcmcia_cs_tuple_format_s
{
  u_char type;            /* Format type code           */
  struct
  {
    u_int type   : 3;     /* Error code type            */
    u_int length : 4;     /* Error code length          */
  }
  err_code;               /* Error detection method and */
                          /* length of error detection  */
                          /* code                       */
  u_long offset;          /* Byte address of the first  */
                          /* data byte in partition     */
  u_long size;            /* Number of data bytes in    */
                          /* this partition             */
  union
  {
    struct
    {
      u_short block_size; /* Block size                 */
      u_long  blocks;     /* Number of data blocks      */
      u_long  err_loc;    /* Location of the error      */
                          /* detection code             */
      u_char  bar;        /* Base Address Register      */
                          /* Indicator                  */
    }
    disk;                 /* Disk like regions          */
    struct
    {
      u_char flags;       /* Various flags              */
      u_char reserved;    /* Reserved                   */
      u_long addr;        /* Physical address at which  */
                          /* this memory partition      */
                          /* should be mapped           */
      u_long err_loc;     /* Location of the error      */
                          /* detection code             */
      u_char bar;         /* Base Address Register      */
                          /* Indicator                  */
    }
    memory;               /* Memory like regions        */
  }
  info;                  /* Additional information,    */
                         /* interpreted based on value */
                         /* of ’type’ field            */
}
pcmcia_cs_tuple_format_t;
```

PCMCIA_CS_TUPLE_GEOMETRY

```
typedef struct pcmcia_cs_tuple_geometry_s
{
  u_char sectors;   /* Sectors per track           */
  u_char tracks;    /* Tracks per cylinder         */
  u_char cylinders; /* Total number of cylinders   */
}
pcmcia_cs_tuple_geometry_t;
```

PCMCIA_CS_TUPLE_BYTEORDER

```
typedef struct pcmcia_cs_tuple_byteorder_s
{
  u_char byteorder; /* Byte order code   */
  u_char bytemap;   /* Byte mapping code */
}
pcmcia_cs_tuple_byteorder_t;
```

PCMCIA_CS_TUPLE_DATE

```
typedef struct pcmcia_cs_tuple_date_s
{
  struct
  {
    u_char seconds; /* Seconds                     */
    u_char minutes; /* Minutes                     */
    u_char hours;   /* Hours                       */
  }
  time;             /* The time at which the card */
                    /* was initialized             */
  struct
  {
    u_char day;     /* Day                         */
    u_char month;   /* Month                       */
    u_char year;    /* Year                        */
  }
  day;              /* The date the card was       */
                    /* initialized                 */
}
pcmcia_cs_tuple_date_t;
```

PCMCIA_CS_TUPLE_BATTERY

```
typedef struct pcmcia_cs_tuple_battery_s
{
  struct
  {
    u_char seconds; /* Seconds                     */
    u_char minutes; /* Minutes                     */
    u_char hours;   /* Hours                       */
  }
  rday;             /* The date on which the       */
                    /* battery was last replaced   */
  struct
  {
    u_char seconds; /* Seconds                   */
    u_char minutes; /* Minutes                   */
    u_char hours;   /* Hours                     */
  }
  xday;             /* The date on which the      */
                    /* battery should be replaced */
}
pcmcia_cs_tuple_battery_t;
```

PCMCIA_CS_TUPLE_ORG

```
typedef struct pcmcia_cs_tuple_org_s
{
  u_char type;      /* Data organization code   */
  u_char fs_info[MAX_STRING_LEN];
                    /* Text description of this */
                    /* organization             */
}
pcmcia_cs_tuple_org_t;
```

PCMCIA_CS_TUPLE_SPCL

```
typedef struct pcmcia_cs_tuple_spcl_s
{
  u_long id;        /* PCMCIA or JEIDA assigned */
                    /* value                    */
  u_char seq;       /* Tuple number in sequence */
  u_char data[MAX_DATA_SIZE];
                    /* The data component       */
  u_char size;      /* Size of data             */
}
pcmcia_cs_tuple_spcl_t;
```

EXAMPLE

```
pcmcia_cs_tuple_data_t tuple_data;

tuple.socket = 0;
tuple.code = PCMCIA_CS_TUPLE_VERS_1;
res = CardServices( GetFirstTuple, handle, (void *)0, sizeof( tuple ),
&tuple );
if ( res == PCMCIA_CS_SUCCESS )
{
  res = CardServices( ParseTuple, handle, &tuple, sizeof( tuple_data ),
 &tuple_data );

  if ( res == PCMCIA_CS_SUCCESS )
  {
    cprintf( "CARD DETECTED: v%d.%d\n", tuple_data.vers_1.major,
                                        tuple_data.vers_1.minor );
    cprintf( "               %s\n", tuple_data.vers_1.manf_name );
    cprintf( "               %s\n", tuple_data.vers_1.prod_name );
    cprintf( "               %s\n", tuple_data.vers_1.lot_info );
    cprintf( "               %s\n", tuple_data.vers_1.prog_info );
  }
}
```

RETURN CODES

PCMCIA_CS_SUCCESS	Request succeeded.
PCMCIA_CS_BAD_SOCKET	Specified socket is invalid.
PCMCIA_CS_UNSUPPORTED_SERVICE	Service is not supported.
PCMCIA_CS_BAD_ARG_LENGTH	ArgLength is invalid.

RegisterClient

The RegisterClient service registers a client with Card Services. ArgPointer must be a pointer to a pcmcia_cs_register_t structure. The client callback handler entry point is passed in Pointer. If the Handle argument is not NULL, the service places the new client handle into the location pointed by Handle.

The pcmcia_cs_register_t structure is defined as follows:

```
typedef struct pcmcia_cs_register_s
{
  int    attributes;  /* bit-mapped client attributes */
  int    event_mask;  /* notification events          */
  void * client_data; /* user-specific client         */
}
pcmcia_cs_register_t;
```

where:

attributes	Are bit mapped client attributes. Bit mapped client attributes are bitwise combinations of the followings constants: • PCMCIA_CS_ATTR_MEM - Memory client device driver • PCMCIA_CS_ATTR_MTD - Memory technology driver • PCMCIA_CS_ATTR_IO - I/O client device driver • PCMCIA_CS_ATTR_INSERT - If specified, the client receives an artificial PCMCIA_CS_EVENT_INSERTION event for every socket that contains a PC Card.
event_mask	Are events of which to notify the client - A bitwise combination of event codes.
client_data	Is user-specific data.

EXAMPLE

```
pcmcia_cs_register_t reg;

reg.attributes = PCMCIA_CS_ATTR_IO | PCMCIA_CS_ATTR_INSERT;
reg.event_mask = PCMCIA_CS_EVENT_INSERTION;
reg.client_data = NULL;
res = CardServices( RegisterClient, NULL, callback, sizeof( reg ), &reg );

if ( res == PCMCIA_CS_SUCCESS )
{
  cprintf( "Registration complete\n" );
}
```

RETURN CODES

`PCMCIA_CS_SUCCESS`	Request succeeded.
`PCMCIA_CS_UNSUPPORTED_SERVICE`	Service is not supported.
`PCMCIA_CS_BAD_ARG_LENGTH`	`ArgLength` is invalid.
`PCMCIA_CS_BAD_ATTRIBUTE`	Incorrect client type.

ReleaseConfiguration

`ReleaseConfiguration` service returns a PC Card and its socket to a simple interface and configuration zero. `ArgPointer` must be a pointer to a `pcmcia_cs_release_config_t` structure.

The `pcmcia_cs_release_config_t` structure is defined as follows:

```
typedef struct pcmcia_cs_release_config_s
{
  int socket; /* logical socket */
}
pcmcia_cs_release_config_t;
```

where:

`socket`	Is a logical socket.

EXAMPLE

```
pcmcia_cs_release_config_t release;

release.socket = 0;
res = CardServices( ReleaseConfiguration, NULL, NULL, sizeof( release ),
 &release );

if ( res == PCMCIA_CS_SUCCESS )
{
  cprintf( "Configuration released\n" );
}
```

RETURN CODES

`PCMCIA_CS_SUCCESS`	Request succeeded.
`PCMCIA_CS_UNSUPPORTED_SERVICE`	Service is not supported.
`PCMCIA_CS_BAD_ARG_LENGTH`	`ArgLength` is invalid.
`PCMCIA_CS_BAD_SOCKET`	Specified socket is invalid.

ReleaseIO

The `ReleaseIO` service releases the I/O addresses requested with the `RequestIO` service. Only the Card Services resource database is modified by a call to this service. No changes are made in the socket adapter. `ArgPointer` must be a pointer to a `pcmcia_cs_release_io_t` structure.

The `pcmcia_cs_release_io_t` structure is defined as follows:

```
typedef struct pcmcia_cs_release_io_s
{
  int            socket; /* logical socket */
}
pcmcia_cs_release_io_t;
```

where:

`socket`	Is a logical socket.

EXAMPLE

```
pcmcia_cs_release_io_t release;

release.socket = 0;
res = CardServices( ReleaseIO, NULL, NULL, sizeof( release ),
                    &release );

if ( res == PCMCIA_CS_SUCCESS )
{
  cprintf( "IO released\n" );
}
```

RETURN CODES

`PCMCIA_CS_SUCCESS`	Request succeeded.
`PCMCIA_CS_UNSUPPORTED_SERVICE`	Service is not supported.
`PCMCIA_CS_BAD_ARG_LENGTH`	`ArgLength` is invalid.
`PCMCIA_CS_BAD_SOCKET`	Specified socket is invalid.

ReleaseIRQ

The `ReleaseIRQ` service releases a previously allocated interrupt line. Only the Card Services resource database is modified by a call to this service. No changes are made in the socket adapter. The `ArgPointer` must be a pointer to a `pcmcia_cs_release_irq_t` structure.

The `pcmcia_cs_release_irq_t` structure is defined as follows:

```
typedef struct pcmcia_cs_release_irq_s
{
  int socket; /* logical socket */
}
pcmcia_cs_release_irq_t;
```

where:

`socket`	Is a logical socket.

EXAMPLE

```
pcmcia_cs_release_irq_t release;

release.socket = 0;
res = CardServices( ReleaseIRQ, NULL, NULL, sizeof( release ),
                    &release );

if ( res == PCMCIA_CS_SUCCESS )
{
  cprintf( "IRQ released\n" );
}
```

RETURN CODES

`PCMCIA_CS_SUCCESS`	Request succeeded.
`PCMCIA_CS_UNSUPPORTED_SERVICE`	Service is not supported.
`PCMCIA_CS_BAD_ARG_LENGTH`	`ArgLength` is invalid.
`PCMCIA_CS_BAD_SOCKET`	Specified socket is invalid.

RequestConfiguration

`RequestConfiguration` service configures the PC Card and socket. `ArgPointer` must be a pointer to a `pcmcia_cs_request_config_t` structure.

The `pcmcia_cs_request_config_t` structure is defined as follows:

```
typedef struct pcmcia_cs_request_config_s
{
  int socket;   /* logical socket                       */
```

```
      int vcc;      /* Vcc settings                         */
      int vpp;      /* Vpp settings                         */
      int int_type; /* interface type                       */
      unsigned char status;   /* Card Status register settings        */
      unsigned char pin;      /* Card Pin register settings           */
      unsigned char copy;     /* Card Copy register settings          */
      unsigned char option;   /* Card Option register settings        */
      int present; /* Card Configuration registers present */
    }
    pcmcia_cs_request_config_t;
```

where:

`socket`	Is a logical socket.
`vcc`	Is the Vcc setting. The voltage is expressed in tenths of a volt.
`vpp`	Is the Vpp setting. The voltage is expressed in tenths of a volt.
`int_type`	Is the interface type. Must be set to: • `PCMCIA_CS_INTERFACE_NONE` - Is for the simplest interface, featuring only card detection. • `PCMCIA_CS_INTERFACE_MEM` - Is for memory only interface. • `PCMCIA_CS_INTERFACE_IO` - Is for memory and I/O interface.
`status`	Is the Card Status register settings, if present.
`pin`	Is the Card Pin register settings, if present.
`copy`	Is the Card Socket/Copy register settings, if present.
`option`	Is the Card Option register settings, if present.
`present`	Specifies Card Configuration registers present. `present` is a bitwise combination of the following constants: • `PCMCIA_CS_PRESENT_STATUS` - The status register is present. • `PCMCIA_CS_PRESENT_PIN` - The pin register is present. • `PCMCIA_CS_PRESENT_COPY` - The copy register is present. • `PCMCIA_CS_PRESENT_OPTION` - The option register is present. Only those registers that are specified using this field are set.

EXAMPLE

```
pcmcia_cs_request_config_t req;

req.socket   = 0;
req.vcc      = 50;
req.vpp      = 50;
req.int_type = PCMCIA_CS_INTERFACE_MEM;
req.present  = 0;
res = CardServices( RequestConfiguration, NULL, NULL, sizeof( req ), &req
);

if ( res == PCMCIA_CS_SUCCESS )
{
  cprintf( "Configuration succeeded\n" );
}
```

RETURN CODES

PCMCIA_CS_SUCCESS	Request succeeded.
PCMCIA_CS_UNSUPPORTED_SERVICE	Service is not supported.
PCMCIA_CS_BAD_ARG_LENGTH	ArgLength is invalid.
PCMCIA_CS_BAD_SOCKET	Specified socket is invalid.
PCMCIA_CS_CONFIGURATION_LOCKED	The RequestConfiguration service has already been called for this socket but a matching ReleaseConfiguration has not.
PCMCIA_CS_NO_CARD	No PC Card in socket.

RequestIO

The RequestIO service reserves the specified I/O range for later assignment using the RequestConfiguration service. ArgPointer must be a pointer to a pcmcia_cs_request_io_t structure.

The pcmcia_cs_request_io_t structure is defined as follows:

```
typedef struct pcmcia_cs_request_io_s
{
  int      socket;      /* logical socket              */
  unsigned base_port1;  /* base port address for range */
  int      num_ports1;  /* number of contiguous ports  */
  int      attributes1; /* bit-mapped port attributes  */
  unsigned base_port2;  /* base port address for range */
  int      num_ports2;  /* number of contiguous ports  */
  int      attributes2; /* bit-mapped port attributes  */
}
pcmcia_cs_request_io_t;
```

where

socket	Is a logical socket.
base_port1	Is the base port number for I/O window 1.
num_ports1	Is the number of contiguous ports in I/O window 1.
attributes1	If set to PCMCIA_CS_IO_ATTR_8BIT, the I/O window has an 8 bit width.

`base_port2`	Is the base port number for I/O window 2.
`num_ports2`	Is the number of contiguous ports in I/O window 2.
`attributes2`	If set to `PCMCIA_CS_IO_ATTR_8BIT`, the I/O window has an 8 bit width.

EXAMPLE

```
pcmcia_cs_request_io_t req;

req.socket      = 0;
req.base_port1  = 0x3f0;
req.num_ports1  = 32;
req.attributes1 = 0;
req.base_port2  = 0x1f0;
req.num_ports2  = 4;
req.attributes1 = 0;
res = CardServices( RequestIO, NULL, NULL, sizeof( req ), &req );

if ( res == PCMCIA_CS_SUCCESS )
{
  cprintf( "I/O range request successful\n" );
}
```

RETURN CODES

`PCMCIA_CS_SUCCESS`	Request succeeded.
`PCMCIA_CS_UNSUPPORTED_SERVICE`	Service is not supported.
`PCMCIA_CS_BAD_ARG_LENGTH`	`ArgLength` is invalid.
`PCMCIA_CS_BAD_SOCKET`	Specified socket is invalid.
`PCMCIA_CS_CONFIGURATION_LOCKED`	The `RequestConfiguration` service has already been called for this socket but a matching `ReleaseConfiguration` has not.
`PCMCIA_CS_OUT_OF_RESOURCE`	Requested I/O window is already in use.

RequestIRQ

The `RequestIRQ` service reserves the specified IRQ line for later assignment using the `RequestConfiguration` service. The `ArgPointer` must be a pointer to a `pcmcia_cs_request_irq_t` structure.

The `pcmcia_cs_request_irq_t` structure is defined as follows:

```
typedef struct pcmcia_cs_request_irq_s
{
  int socket;       /* logical socket           */
  int assigned_irq; /* irq assigned to PC Card */
}
pcmcia_cs_request_irq_t;
```

where:

`socket`	Is a logical socket.
`assigned_irq`	Is the IRQ line.

EXAMPLE

```
pcmcia_cs_request_irq_t req;

req.socket       = 0;
req.assigned_irq = 14;
res = CardServices( RequestIRQ, NULL, NULL, sizeof( req ), &req );

if ( res == PCMCIA_CS_SUCCESS )
{
  cprintf( "IRQ request successful\n" );
}
```

RETURN CODES

`PCMCIA_CS_SUCCESS`	Request succeeded.
`PCMCIA_CS_UNSUPPORTED_SERVICE`	Service is not supported.
`PCMCIA_CS_BAD_ARG_LENGTH`	`ArgLength` is invalid.
`PCMCIA_CS_BAD_SOCKET`	Specified socket is invalid.
`PCMCIA_CS_CONFIGURATION_LOCKED`	The `RequestConfiguration` service has already been called for this socket but a matching `ReleaseConfiguration` has not.
`PCMCIA_CS_OUT_OF_RESOURCE`	Requested I/O window is already in use.

ServiceName

`ServiceName` service returns a character string corresponding to a specified request code. `ArgPointer` must be a pointer to a `pcmcia_cs_name_t` structure.

The `pcmcia_cs_name_t` structure is defined as follows:

```
typedef struct pcmcia_cs_name_s
{
  int    code;
  char * name;
}
pcmcia_cs_name_t;
```

where

`code`	Is the request code.
`name`	Is a pointer to a string containing the request name.

EXAMPLE

```
pcmcia_cs_name_t name;
name.code = GetConfigurationInfo;
CardServices( ServiceName, NULL, NULL, 0, &name );

/* should print GetConfigurationInfo */
cprintf( "code 0x%x, name %s\n", name.code, name.name );
```

RETURN CODES

`PCMCIA_CS_SUCCESS`	Request succeeded.
`PCMCIA_CS_UNSUPPORTED_SERVICE`	Service is not supported.

PC Card Support

PC Card Support is a facility that enables the use of PC Card (PCMCIA) devices in the LynxOS environment. Features supported by the PC Card subsystem include:

- Extendable support for several widely used PC Card devices. Supported cards are accessible as conventional ISA devices controlled by standard LynxOS device drivers.
- Simple and platform-independent means for developing new PC Card software (PC Card enablers and PC Card drivers).
- Support of preinstalled PC Card devices, thus allowing root file system to reside on a PC Card disk.
- Support of runtime insertion and extraction of PC Card devices.

Installing and Removing PC Card Support

.PC Card Support can be installed and removed after initial installation of LynxOS.

To install PC Card Support enter the following commands:

```
cd /sys/lynx.os
make install.pcmcia
```

To remove PC Card Support enter the following commands:

```
cd /sys/lynx.os
make uninstall.pcmcia
```

PC Card Support Architecture

The LynxOS PC Card subsystem is based upon the PCMCIA/JEIDA PC Card architecture specification. It has three main layers (see figure below). At the lowest level is Socket Services. The next level is Card Services. Layered on the top of Card Services are client device drivers. All these components are kernel entities. User mode applications interact with PC Card devices and the PC Card subsystem using interfaces defined by client device drivers.

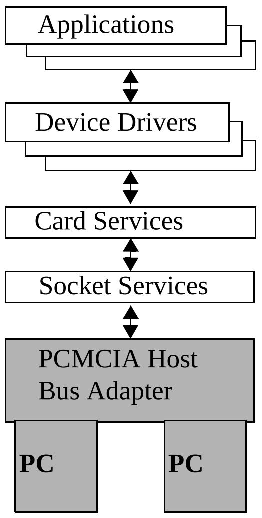


Figure 11-2: PC Card Support Architecture

Socket Services

Socket Services provides a standardized interface to manipulate PC Cards, sockets and adapters.

Host systems may have more than one PC Card adapter present. Each adapter has its own Socket Services. All instances of Socket Services are intended to support a single instance of Card Services. A socket service registers with Card Services and notifies it of status changes on PC Cards or in sockets.

By making all accesses to adapters, sockets, and PC Cards through the socket services interface, higher-level software is unaffected by different implementations of the hardware. Only hardware-specific socket services implementations must be modified to accommodate any different hardware implementations.

LynxOS PC Card Support implements socket services as an ordinary device driver. Refer to “Writing PC Card Socket Services” on page 257 for a detailed discussion of interfaces defined by Socket Services and for information on how to develop a new socket service.

Card Services

Above the Socket Services layer is the Card Services layer. Card Services coordinates accesses to PC Cards, sockets, and system resources among multiple clients. There is only one instance of Card Services in the system.

Card Services makes all accesses to the hardware level through the Socket Services interface. All Socket Services status change reporting is routed to Card Services. Card Services then notifies the appropriate clients. Card Services preserves for its clients an abstract, hardware-independent view of a card and associated system resources.

LynxOS PC Card Support implements the Card Services as an ordinary device driver. Refer to “Writing PC Card Socket Services” on page 257” for a detailed discussion of interfaces defined by Card Services.

Client Device Drivers

Client Device Drivers refers to all users of Card Services. In a LynxOS PC Card subsystem, users of Card Services are devices drivers that use the standardized API `CardServices()` to manipulate PC Cards, sockets, and adapters.

PC Card Enabler

LynxOS PC Card support includes a special client device driver called the PC Card Enabler. This driver responds to runtime insertion and removal events and provides the following services to the rest of the system:

- Card Configuration - Configured card acts as an ISA device.
- Automatic installation and deinstallation of static device drivers in response to card insertion and removal
- `ioctl`-based API to user mode applications - The API allows applications to monitor status of the PC Card subsystem.

Refer to `pcmcia_enabler(4)` man page for a detailed description of the PC Card Enabler.

PC Card Utilities

LynxOS PC Card support includes the following utilities:

- `pcmcia_info` - An information utility that displays information about all logical PCMCIA sockets and PC Card devices present in the system
- `pcmcia_shu` - A control utility used to prepare a socket for card removal
- `pcmcia_d` - A daemon that automatically handles dynamic driver installation and deinstallation in response to card insertion and removal.

For a complete discussion of each utility, refer to an appropriate man page.

Using a PC Card

When a supported PC Card is inserted before system boot, it is automatically configured by the PC Card Enabler. It can then be accessed as an ISA device using the existing device driver.

Supported Cards

The following PC Cards are supported by the PC Card Enabler and work with the specified standard LynxOS driver:

Table 11-3: Supported PC Cards

PC Card	Driver for x86	Driver for MPC860
3Com 3C589x EtherLinkIII Ethernet adapter	`if_3c5x9`	`if_3c5x9`
Adaptec SlimSCSI SCSI adapter	`sim1522`	`sim1522_8xx`
EigerStar ATA Hard Disk	`ide`	`ide`

Hot Swapping

Hot swapping is used to refer to the ability of PC Cards to be inserted and removed when power is applied to the machine and OS is running, and the ability of the system to automatically detect and react to configuration changes.

When a card is being inserted on a running system, it is handled by the PC Card Enabler exactly the same way as a card inserted to a socket prior to boot. If the card is supported, it is configured as specified in the PC Card Enabler configuration tables.

There are two ways to automatically install an appropriate device driver in response to card insertion.

One way is to use the PC Card Enabler static driver installation feature. It allows for an automatic call to the device driver `install` entry point function of a static driver that failed to install a major device at the boot time due to the absence of a device. Because the PC Card Enabler calls the `install` entry point after the PC Card device has been inserted and configured on the I/O bus, the driver can successfully create an appropriate major device and return success, thus making the PC Card device available for the device driver operations.

Information about statically linked device drivers supported by the PC Card subsystem is located in the configuration file `/sys/devices/pcmcia_enabler_info.c`. Please refer to `pcmcia_enabler(4)` man page for a detailed description of the PC Card Enabler facility.

The second way is to use `pcmcia_d` daemon program. This daemon program responds to card insertion events, and acts according to the configuration data in the configuration files. The key responsibility of `pcmcia_d` is to dynamically install an appropriate device driver, create a device node, and invoke an optional script configuring the PC Card for operation in the system. For instance, the script can mount an inserted PC Card disk as a LynxOS file system. Please refer to `pcmcia_d(1)` man page for a detailed description of the `pcmcia_d` daemon.

It is possible to remove a PC Card from a running system. Prior to removal, call the `pcmcia_shu` utility to prepare the socket for card removal. `pcmcia_shu` is responsible for ensuring that no device driver is accessing the device being removed. `pcmcia_shu` interacts with the PC Card Enabler and `pcmcia_d` to handle the removal request. An attempt to deinstall the driver is made. In addition, if the driver has been deinstalled by the `pcmcia_d` daemon, the daemon calls an optional user script that removes the device special node.

Request to remove a PC Card device may fail, for example, if the PC Card disk being removed is mounted as a file system. Appropriate steps must be taken to ensure that the PC Card removed is not used by any application. Removing a card without a successful card shutdown may cause a system crash.

Adding Support for a New PC Card

You can add support for a new PC Card by adding support to the PC Card Enabler configuration tables or by developing of a new client device driver.

Adding Support to PC Card Enabler

To add support for a new card, add a new entry to the PC Card Enabler configuration table. The required information includes the manufacturer and product identification numbers from the card `MANFID` tuple, Configuration Table Entry index, and the configuration to which the entry corresponds.

You can use the `pcmcia_info` utility to find out various information about the PC Card. For instance, if the PC Card device is inserted into the first PCMCIA socket, `pcmcia_info` creates a display similar to one shown below:

```
lynx1# pcmcia_info 0
Card in socket #0 : [ 0x0106 0x0000 ]

Current configuration:
I/O range 1: 0x1f0 0x8
I/O range 2: 0x3f6 0x1
IRQ: 46

+-------+-----------+-----------+-------+
| Index |   Range1   |   Range2   | Width |
+-------+-----------+-----------+-------+
|  0x01 |  N * 0x10 |    n/a    |   16  |
|  0x02 | 0x1f0 0x08 | 0x3f6 0x01 |   16  |
| *0x03 | 0x170 0x08 | 0x376 0x01 |   16  |
+-------+-----------+-----------+-------+
```

The first line indicates that there is a PC Card device found in the PCMCIA socket, and its manufacturer and product identification numbers are `0x0106` and `0x0000` respectively.

The table shows correspondence between a Configuration Table Entry and I/O ranges used by the particular configuration. Choose an entry with the configuration most appropriate for the host system.

To add a new entry to the PC Card Enabler configuration table, use the information obtained using `pcmcia_info` to define an entry corresponding to the new PC Card device. A detailed description of the PC Card Enabler configuration table format is available in `pcmcia_enabler(4)` man page.

Next configure the existing LynxOS driver capable of controlling the ISA device. Configure the driver to ensure that it uses I/O ranges and IRQ line identical to those specified in the PC Card Enabler configuration table.

Create New Device Driver

A new device driver should be a regular device driver using the Card Services API to detect, identify, and configure the PC Card prior to any normal mode accesses to the device. Refer to “Writing PC Card Socket Services” on page 257 for a detailed specification of the Card Services API.

Adding Support for a New PCMCIA Adapter

Adding support for a new PCMCIA adapter involves development of a new Socket Services device driver. No changes to other components of the PC Card Support software are necessary. Refer to “Writing PC Card Socket Services” on page 257 for a detailed specification of the Socket Services API and information on how to develop a new Socket Services driver.

Supported PCMCIA Adapters

The following PCMCIA adapters are supported by the LynxOS PC Card subsystem:

Table 11-4: Supported PCMCIA Adapters

PCMCIA Adapter	Architecture
Embedded RPXL823 PCMCIA Interface	RPXL823
i82365 compatible ISA/PCI PCMCIA Adapter	x86 adapters

Troubleshooting

This section provides troubleshooting tips that support the EtherLinkIII PC Card on the x86 platform.

1. Check that Socket Services information structure contains the correct information about the PCMCIA adapter(s) installed in your system:

   Open the `/sys/devices/pcmcia_ss_pci_info.c` file. Find the description of your adapter(s) in the `sspci_known_adapters` array. The current revision of the driver has only two entries - one for the O2Micro OZ6860 adapter, another for all other i82365-compatible adapters. All adapters other than OZ6860 are configured in the ISA legacy mode. If you are having problems with your adapter, try to increase the verbosity level by setting the `SSPCI_FLAG_VERBOSE` flag in the attributes field of the adapter entry. Additional debug information may help to understand the problem. Note that the debug output is sent to `COM2`. Another field that you might want to change in the Socket Services information file is the interrupt delivery mode. Refer to the next troubleshooting tip for details. Note the possible ISA locations table. You can add ISA port address to the `sspci_isa_locations` array, if you know that your adapter uses an alternate ISA port.

   Modify the file as necessary and rebuild the Socket Services device and the kernel.

2. Check the IRQ numbers and I/O addresses in the Enabler table and in the EtherLinkIII device information file are the same and are not in use by some other device:

Open `/sys/devices/pcmcia_enabler_info.c`. Find the following entry in the `pcmcia_enabler_card_table_cfg` structure:

```
{
  0x101, 0x589,       /* 3COM 3c589x cards    */

  #ifdef __x86__
    0x110, 0x10, 16,
  #else
    0x220, 0x10, 16,  /* 0x220-0x230          */
  #endif

  0x0, 0x0, 0x0,      /* unused               */
  0x1 | 0x40,         /* config index 1       */
                      /* level mode intrs     */
  #ifdef __x86__
    10
  #else
    9                 /* IRQ 9                */
  #endif

},
```

The first two fields identify the card. The third field is the base I/O address of the card. The last field is the IRQ number assigned to the card. Note that 32 is not added to the IRQ number.

Open `/sys/devices/if_3c589x_pcmcia_info.c`. Observe the following structure:

```
struct if_3c5x9_info if_3c589x_pcmcia_info = {
  0x110,
  32 + 10,
  0,      /* (EISA only) slot number, ISA set to  zero */
  0,      /* TP = 0, AUI = 1, BNC = 3 */
  3       /* bus type, ISA = 1, EISA = 2, PCMCIA = 3 */
};
```

The first field of the structure is the base I/O address of the device. The second field is the IRQ number. Note that here, 32 is added to the IRQ number. The assigned IRQ number must be the same in both files, and not in use by any other device. The I/O address must also be the same in both files. The I/O range used by the EtherLinkIII card must not overlap with I/O ranges used by other devices.

Unfortunately, there is no simple way to determine the IRQ numbers and I/O ranges are already in use. The device information files for all configured devices have to be examined. As a starting point, try the following configurations:

- For a laptop computer: I/O address = 0x110, IRQ = 10.
- For a desktop computer: I/O address = 0x220, IRQ = 9,12,15.

It is also possible that different values have to be used. Note that the I/O range must start at the 16 byte boundary. For example, 0x220 and 0x230 are valid, while 0x223 and 0x228 are not.

If no IRQ and base address combination works for the card, try to enable the PC Card I/O interrupts emulation mode in the Socket Services driver. To enable emulation, modify `/sys/devices/pcmcia_ss_pci_info.c`.

Find in the supported adapters table the entry containing information about the PCMCIA adapter, and add the `SSPCI_FLAG_EMU` flag to the attributes field.

To change the configuration edit the appropriate files, rebuild the device information files and the kernel. For example:

```
# cd /sys/devices
# vi pcmcia_enabler_info.c
```

Make necessary corrections.

```
# vi if_3c589x_pcmcia_info.c
```

Make necessary corrections.

```
# make install
# cd /sys/lynx.os
# make install
# cd /dev
# mknod -a /etc/nodetab
```

3. Shut down the host computer. Remove all PC Cards from the PCMCIA sockets. Insert the EtherLinkIII PC Card into PCMCIA socket #0. Boot.

   If the system hangs, go to troubleshooting tip 2 and change the values of IRQ number, I/O address, or both. Start with IRQ number the I/O address. IRQ 9,10,11,12,15 are good examples.

4. Execute:

   ```
   # ls -1 /dev/pcmcia* /dev/el_pc
   ```

   The following lines should be present in the output:

   ```
   /dev/el_pc
   /dev/pcmcia_enabler
   ```

   If any of the above nodes is not present, check `CONFIG.TBL` and make sure that all listed devices are installed. Rebuild the kernel if necessary.

Rebuild the contents of `/dev` directory from `/etc/nodetab` using the following commands:

```
# cd /dev
# mknod -a /etc/nodetab
```

5. Execute:

```
# devices | grep -i "pcmcia \| CardBus/i82365"
```

The output should be similar to the following:

```
35    char   36     0     db27d6d8       0     pcmcia CS
36    char   37     0     0              0     CardBus/i82365
SS
37    char   38     0     db280178       0     PCMCIA Enabler
```

Figure 11-3: devices Command Output

Possible mismatches are explained below:

- Some of the devices are not present. Check `CONFIG.TBL` and make sure that all listed devices are installed. Rebuild the kernel if necessary.
- All devices are present, but “PCI CardBus SS” is not installed (`(no dev)` value in device start address column). Make sure that a PCMCIA adapter is present in the system and that it is i82365-compatible. This package does not support other types of PCMCIA adapters. If you are using an adapter other than OZ6860, make sure that the appropriate ISA port is listed in the possible ISA locations table in the Socket Services information file. Also ensure that the device configuration files are listed in `CONFIG.TBL` and are in proper order. Rebuild the kernel if necessary.
- All devices are present, but “pcmcia CS” or “PCMCIA Enabler” is not installed (`(no dev)` value in device start address column). Make sure that device configuration files are listed in `CONFIG.TBL` and are in proper order. Rebuild the kernel if necessary.

6. Execute:

```
# pcmcia_info
```

You should see a table of the following format:

```
+----------+---------+------------+--------+--------+
| Socket # | Present | Configured |  Code  |  Card  |
+----------+---------+------------+--------+--------+
|        0*| Yes     | Yes        | 0x0101 | 0x0589 |
|        1 | No      | No         |  n/a   |   n/a  |
+----------+---------+------------+--------+--------+
```

Possible mismatches are explained below:

- Message “`failed to open file /dev/pcmcia_enabler`”. Rebuild the contents of `/dev` from `/etc/nodetab` using the commands:

  ```
  # cd /dev
  # mknod -a /etc/nodetab
  ```

- The table shown by the `pcmcia_info` utility is empty. Make sure that a PCMCIA adapter is present in the system and that it is i82365-compatible. This package does not support other types of PCMCIA adapters. If you are using an adapter other than OZ6860, make sure that the appropriate ISA port is listed in the possible ISA locations table in the Socket Services information file.

- The card is not marked as present. Check that the card is properly installed in the socket.

- The card is not marked as configured. There is no description for this card in Card Enabler’s information table. Check that Code and Card values provided by `pcmcia_info` match the values for the 3C589C card in the Card Enabler information file. These values should be equal to `0x0101` and `0x0589` respectively. If `pcmcia_info` gives different values, the wrong card is inserted. Also try different IRQ and I/O address settings (see Step 2).

7. Execute:

   ```
   # pcmcia_info 0
   ```

   This command gives more information about a PC Card in socket #0.

The sample output is given below:

```
Card in socket #0 : [ 0x0101 0x0589 ]

Current configuration:
I/O range 1: 0x110 0x10
I/O range 2: n/a
IRQ: 9

+-------+-----------+-----------+-------+
| Index |   Range1   |   Range2   | Width |
+-------+-----------+-----------+-------+
| *0x01 |  N * 0x10  |    n/a    |   16  |
+-------+-----------+-----------+-------+
```

Note the `Current configuration` section of the output. The values of I/O range 1 and IRQ should match the corresponding values in `if_3c589x_pcmcia_info.c`. If this is not so, correct `if_3c589x_pcmcia_info.c` or the card descriptions table in the Card Enabler information file as described in Step 2.

8. Execute:

```
# devices | grep el_pc
```

The output should be similar to the following:

```
38 char  39         0         db2802b8 0     el_pc
```

If there is no `el_pc` device listed, check that the line `I:3c589x_pcmcia.cfg` is present in `CONFIG.TBL` file.

If `el_pc` device is present but not installed (`(no dev)` value in the device start address column), check that IRQ value and I/O base address are the same in the output of `pcmcia_info 0` command and in the `if_3c589x_pcmcia_info.c` file.

Also make sure that the line `I:3c589x_pcmcia.cfg` has been placed in `CONFIG.TBL` after the three lines describing the PCMCIA devices (`I:pcmcia_cs.cfg`, `I:pcmcia_ss_pci.cfg`, `I:pcmcia_enabler.cfg`).

Try different IRQ and I/O address settings (see Step 2).

Writing PC Card Socket Services

Socket Services is defined as a device driver that interfaces to the Card Services module and implements hardware-specific details of programming a specific PCMCIA adapter. The interface between Card Services and Socket Services is defined by a set of driver entry points and data structures, described in the following sections.

Socket Services Overview

Socket Services is the lower layer in the PC Card Support subsystem. Socket Services provides a unified software interface to the PCMCIA sockets hardware. It masks the hardware implementation details, providing an abstraction layer that allows for development of higher-level software without explicit knowledge of the underlying hardware interfaces.

Socket Services handles the hardware as a number of objects of different types. A PCMCIA adapter is the hardware that connects a host system bus to PC Card sockets. A host system may have more than one adapter. There is one instance of Socket Services for each adapter present in the system. Socket Services reports the number of sockets and windows implemented by the adapter it services. An adapter has one or more sockets. Sockets are receptacles for PC Cards and the source of status change events. A range in the PC Card memory or I/O address space can be mapped into the host system space through a window. Most adapters provide a limited number of windows and each window has different mapping capabilities. Socket Services reports the characteristics of each window to the higher-level PC Card Support layers.

Socket Services Groups

The Socket Services interface can be divided into three functional groups of services.

- Adapter Services
- Socket Services
- Window Services

Adapter Services

Socket Services controls the adapter using the following `SS_GetInfo` service.

Socket Services

Socket Services controls sockets using the following services:

- `SS_GetSocket`
- `SS_SetSocket`
- `SS_InquireSocket`

Window Services

Socket Services controls windows using the following services:

- `SS_GetWindow`
- `SS_SetWindow`
- `SS_InquireWindow`

Socket Services Structure

LynxOS PC Card Support implements a Socket Services as an ordinary device driver. A Socket Services registers with Card Services at installation. As soon as the registration is complete, the Socket Services resources are available to Card Services and its clients. Socket Services drivers should be installed after Card Services, but before any client device drivers.

Header Files

All prototypes and constants needed to implement the Socket Services interface are defined in the following header files:

Table 11-5: Socket Services Header Files

Header File	Description
`/sys/dheaders/pcmcia_ss.h`	Contains definitions of the Socket Services interface.
`/sys/dheaders/pcmcia_cs_ss.h`	Contains prototypes for the Socket Services callback, registration entry point, and event notification entry point.

Registration

To register with Card Services, a Socket Services calls the `pcmcia_cs_register_ss` function, passing all necessary registration data as the parameters. The registration data consists of a callback entry point and user specific data to be passed to the callback:

```
id = pcmcia_cs_register_ss( callback, statics_ptr );
```

`pcmcia_cs_register_ss` returns a Socket Services identification number. The Socket Services identification number is passed back to Card Services when Socket Services notifies it of a status change event. If registration fails, `pcmcia_cs_register_ss` returns -1.

Event Notification

Socket Services intercepts status changes and reports them to Card Services. Status change detection can be interrupt-driven or polled, depending on the hardware features of the PCMCIA adapter. If the adapter supports interrupt-driven delivery of status change events, Socket Services should install an interrupt handler and process status change interrupts within the handler. If the adapter does not implement an interrupt for the status change events, Socket Services should use a polling based technique to detect status changes in the PCMCIA sockets.

Socket Services reports a status change event to Card Services by calling the `pcmcia_cs_event` entry point. Socket status, socket number, and Socket Services identification number obtained at registration time must be passed as the parameters. For example:

```
pcmcia_cs_event( status, socket, id );
```

Socket Services Callback

Card Services invokes a Socket Services through the callback interface. The callback entry point is provided by Socket Services at the registration time.

The callback routine has the following syntax:

```
int callback( void * stat, int service, int Number, void * ArgPointer );
```

where:

`stat`	Is the user-specific data registered through `pcmcia_cs_register_ss`.
`service`	Specifies the service code.

Number	Identifies an object on which to manipulate. Must be either a window or socket number, depending on the service.
ArgPointer	Is service-specific data.

Each service must return one of the following constants:

PCMCIA_SS_SUCCESS	Request succeeded.
PCMCIA_SS_UNSUPPORTED	Service or feature is not supported.
PCMICA_SS_BAD_VOLTAGE	Voltage specified for the SS_SetSocket request cannot be applied.

Socket Services Reference

SS_GetInfo

The SS_GetInfo service returns information about the PCMCIA adapter and the Socket Services. ArgPointer must be a pointer to a pcmcia_ss_information_t structure.

The pcmcia_ss_information_t structure is defined as follows:

```
typedef struct pcmcia_ss_information_s
{
  int    revision;    /* version           */
  int    sockets_num; /* number of sockets */
  int    windows_num; /* number of windows */
  char * name;        /* HBA name          */
  char * vendor;      /* vendor            */
}
pcmcia_ss_information_t;
```

where:

revision	Is the binary coded decimal (BCD) value of the SS version number.
sockets_num	Specifies the number of sockets.
windows_num	Specifies the number of windows.
name	Is the PCMCIA adapter name.
vendor	Is the vendor-specific string.

SS_InquireSocket

The `SS_InquireSocket` service returns the read-only information about the specified socket. `Number` must be set to the socket number. `ArgPointer` must be a pointer to a `pcmcia_ss_inquire_socket_t` structure.

The `pcmcia_ss_inquire_socket_t` structure is defined as follows:

```
typedef struct pcmcia_ss_inquire_socket_s
{
  int status;
}
pcmcia_ss_inquire_socket_t;
```

where:

`status`	Is the bit mapped socket status. A bitwise combination of the following constants: • `PCMCIA_SS_STATUS_DETECT` - PC Card is present in the socket. • `PCMCIA_SS_STATUS_CHANGE` - Status change event in I/O PC Card • `PCMCIA_SS_STATUS_BATTERY_DEAD` - Battery dead event in memory PC Card • `PCMCIA_SS_STATUS_BATTERY_LOW` - Battery low event in memory PC Card • `PCMCIA_SS_STATUS_READY` - Ready event in I/O PC Card

SS_SetSocket

The `SS_SetSocket` service configures the specified socket. `Number` must be set to the socket number. `ArgPointer` must be a pointer to a `pcmcia_ss_socket_t` structure.

The `pcmcia_ss_socket_t` structure is defined as follows:

```
typedef struct pcmcia_ss_socket_s
{
  int interface; /* interface type              */
  int irq;       /* assigned IRQ                */
  int vcc;       /* voltage applied to Vcc pin  */
  int vpp;       /* voltage applied to Vpp pins */
}
pcmcia_ss_socket_t;
```

where:

`interface`	Is the interface type. This field must be set to one of the following constants: • `PCMCIA_SS_INTERFACE_NONE` - Simplest interface, featuring only card detection • `PCMCIA_SS_INTERFACE_MEM` - Memory only interface • `PCMCIA_SS_INTERFACE_IO` - Memory and I/O interface.
`irq`	Specifies the IRQ line. A value of 0 means no steering. This field is meaningful only if `interface` is set to `PCMCIA_SS_INTERFACE_IO`.
`vcc`	Is the voltage applied to the Vcc pin. The voltage is defined in tenths of a volt.
`vpp`	Is the voltage applied to the Vpp pins. The voltage is defined in tenths of a volt.

SS_GetSocket

The `SS_GetSocket` service returns the configuration of the specified socket. `Number` must be set to the socket number. `ArgPointer` must be a pointer to a `pcmcia_ss_socket_t` structure.

SS_InquireWindow

The `SS_InquireWindow` service returns the read-only information about the specified window. `Number` must be set to the window number. `ArgPointer` must be a pointer to a `pcmcia_ss_inquire_window_t` structure.

The `pcmcia_ss_inquire_window_t` structure is defined as follows:

```
typedef struct pcmcia_ss_inquire_window_s
{
  int type;   /* supported window types */
  int socket; /* bit pattern            */

  /* io window characteristics */
  pcmcia_ss_io_win_chars_t io_chars;

  /* memory window characteristics */
  pcmcia_ss_memory_win_chars_t memory_chars;
}
pcmcia_ss_inquire_window_t;
```

where:

`type`	Specifies the type of the window. Must be a bitwise combination of the following constants: • `PCMCIA_SS_WINDOW_COMMON` - memory window • `PCMCIA_SS_WINDOW_ATTR` - attribute window • `PCMCIA_SS_WINDOW_IO` - I/O window
`socket`	Specifies sockets to which this window can be assigned. Each bit corresponds to a single socket. The least significant bit corresponds to the socket 0.
`io_chars`	Specifies window characteristics when used as an I/O window. Currently unused.
`memory_chars`	Specifies window characteristics when used as a common window. Currently unused.

SS_SetWindow

The `SS_SetWindow` service configures the specified window. `Number` must be set to the window number. `ArgPointer` must be a pointer to a `pcmcia_ss_window_t` structure.

The `pcmcia_ss_window_t` structure is defined as follows:

```
typedef struct pcmcia_ss_window_s
{
  int      socket;       /* socket index                 */
  int      type;         /* window type (IO/ATTR/COMMON) */
  unsigned base;         /* card base                    */
  int      size;         /* in bytes                     */
  int      data_width;   /* 8 or 16                      */
  int      access_speed; /* access speed for common type */
  unsigned offset;       /* window offset                */
  int      valid;        /* validity flag                */
}
pcmcia_ss_window_t;
```

where:

`socket`	Is the socket number.
`type`	Specifies the window type. Must be set to one of the following constants: • `PCMCIA_SS_WINDOW_COMMON` - memory window • `PCMCIA_SS_WINDOW_ATTR` - attribute window • `PCMCIA_SS_WINDOW_IO` - I/O window.
`base`	Specifies the host base address of the window.

`size`	Is the number of bytes in the window.
`data_width`	Is the data width in bits. Must be set to 8 or 16.
`access_speed`	Specifies the window access speed in nanoseconds. This field is used only if the `type` field is set to `PCMCIA_SS_WINDOW_COMMON`.
`offset`	Is the offset to the window.
`valid`	Is the window validity flag. If set to zero, the window is disabled.

SS_GetWindow

The `SS_GetWindow` service gets the configuration of the specified window. `Number` must be set to the window number. `ArgPointer` must be a pointer to a `pcmcia_ss_window_t` structure.

APPENDIX A *Porting Linux Drivers to LynxOS*

Due to the popularity of Linux, and the growing population of open source developers, a great deal of device driver code is freely accessible. Though it may not be practical for use directly on most real-time operating systems, many of these drivers can provide important benefits in understanding how a particular device must be manipulated.

For LynxOS developers, however, these device drivers are similar in structure to a LynxOS device driver, with the primary difference being the inherent real-time conditions. If a LynxOS developer needs a device driver and a Linux driver already exists, it is a relatively straightforward process to port it.

This appendix examines the differences and requirements of Linux and LynxOS at the device driver level.

GPL Issues

It is important to take note of GPL issues before using any GPL driver source code. Note that:

- Any modification to the Linux source must be made freely available to the rest of the world under the GPL license.
- You cannot distribute the driver statically linked to the LynxOS kernel. The ported driver must be dynamically installed, as LynxOS is not GPL code.
- No copyright text can be changed.
- If you use the driver source as a reference and write a completely new driver, there are no restrictions on what you may do with the final product, it is yours, not subject to the GPL.

Driver Installation

Driver installation between Linux and LynxOS systems are similar. The following table provides a comparison of static and dynamic installs for both systems.

Table A-1: Driver Installation Differences

Install Type	Linux	LynxOS
Static	Link Driver with kernel from the directory `linux/drivers` with `Makefile`.	Link driver with kernel from directory `sys/drivers` and `sys/devices` with `Makefile`

Dynamic	With `insmod`, execute `init_module()`
	With `mmod`, execute `cleanup_module()`
	`module_init()`
	`module_exit()`
		Use `drinstall` and `devinstall -c/-b`
		Or, use `drvinstall()` to install
		Use `drinstall`, `devinstall -u` or `drvuninstall()` to uninstall.

Using a Device

For both Linux and LynxOS, devices are accessed through a special file called a node, typically located in the /dev directory. Nodes can be created with the mknod utility. The following is an example listing of two devices. Note that the com1 device node is a character device (leading “c” character), and the scsi drive is a block device (leading “b” character):

```
crw--ww- 1 root 6,0    Sep 3 12:01  /dev/com1

brw----- 1 root 1, 16  Sep 29 19:58 /dev/sd0a
```

Once a node is opened with the `open()` call, it can be accessed with any standard file operation (`read()`, `write()`, `ioctl()`).

Major and Minor Numbers

Major and Minor numbers are used to identify a device. For LynxOS, the order of the driver `.cfg` pointed to in the `/sys/lynx.os/CONFIG.TBL` file specifies the major number of the device.

For Linux, drivers can be statically or dynamically assigned major and minor numbers, however the `proc/devices` directory should be checked for number availability.

Accessing a Device

The facilities for accessing a device are described below for both LynxOS and Linux:

Table A-2: Device Access Differences

Function	Linux Facilities	LynxOS Facilities
`mmap()`	`mmap()` from application	`mmap()` from application

I/O port access	`insb(), outsb(), readb/writeb`
	`insw(), outsw(), readw/writew`
	`instl(), outsl(), readl/writel`
		`__inb(), __outb()`
		`__inw(), __outw()`
		`__inl(), __outl()`

Physical to virtual address translation	`vremap()` can be used to map a physical address into a kernel address. The function `ioremap()` can also be used.	`permap()` can be used to map a physical address into a kernel virtual address.
PCI autoconfig access	PCI functions are used to handle PCI devices.	DRM (Device Resource Manager) is a set of functions used to access PCI devices.

Driver Entry Points

The driver APIs for both Linux and LynxOS are fairly standard. They provide entry points from user space to driver space through a series of system calls. The following table shows the correlating calls between Linux and LynxOS.

Table A-3: Linux and LynxOS Entry Points

Linux Call	LynxOS Call
`setup(), module_init()`	`install()`
`init(), init_module()`	`install()`

`cleanup_module(),`
`module_exit()`
	`uninstall()`

Table A-3: Linux and LynxOS Entry Points (Continued)

Linux Call	LynxOS Call
`open()`	`open()`
`release()`	`close()`
`read()`	`read()`
`write()`	`write()`
`ioctl()`	`ioctl()`
`select()`	`select()`
`lseek()`	`ioctl()`
`mmap() ioremap()`	`mmap()`
`readdir(), fsync(), fasync()`	`strategy()`
`check_media(), change(), revalidate()`	`N/A`

System Call Processing

The processing of system calls is similar on both LynxOS and Linux systems. The following table describes the differences.:

Table A-4: System Call Action Differences

Action	Linux	LynxOS
System call executed	When a task performs a system call, it is run in the task’s context. The state changes from user to system.	When a task performs a system call, it is run in the task’s context. The state changes from user to system.
Running process	A running process is called a task and can be a process or thread.	A running process is executed by at least the initial thread or the main thread.
Scheduling	The entities that get scheduled according to their priority are tasks	The entities that get scheduled according to their priority are user threads or kernel threads.

Table A-4: System Call Action Differences

Action	Linux	LynxOS
ISRs (Interrupt Service Request)	ISRs will run automatically in the current thread context when an interrupt occurs.	ISRs will run automatically in the current thread context when an interrupt occurs
ID Changes	One PID value for both threads and processes is used.	PID for processes and TID (Thread ID) for threads.)

Preemption

System calls can be preempted on both systems, but the degree of preemption is different

Table A-5: Preemption Differences

Linux	LynxOS
The running system call can be preempted by another task or by a slow interrupt.	The running thread can be preempted in the middle of its system call by another, higher priority thread, or by an interrupt.
The running system call can be preempted when it goes to sleep through the use of `interruptible_sleep_on()` option.	Preemption is implicit and can be disabled and restored with the functions `sdisable()` and `srestore()` (although there are non-reentrant areas in the kernel where this should not be done.)
The structure task_struct contains the priority value of the running task	The system call can get its priority with the function `_getpriority`().

Signal Handling

Signal handling also differs between the two systems

Table A-6: Signal Handling Comparison

Linux	**LynxOS**

The state of the task defines how systems are handled while blocking. The states `TASK_INTERRUPTABLE` and `TASK_UNINTERRUPTIBLE` are used to define the state of the task.	The flag's semaphore is used in the system call to define how signals are handled during blocking:
	`IGNORE_SIGS`
	`DELIVER_SIGS`
	`ABORT_ON_SIGS`

While blocking, these functions define if the task is interruptible:
`Interruptible_sleep_on()`
`Sleep_on()`
`Wake_up_interruptible()`
`Wake_up()`
	For interrupt handling during blocking, the `swait()` or `tswait()` calls are used to affect behavior.

Error Handling

LynxOS and Linux handle error returns from system calls differently

Table A-7: Error Handling Comparison

Linux	**LynxOS**
The error value returned by the last system call can be stored in a global variable called `errno`.	The error value of the last system call can be set with `pseterr()` or retrieved with `pgeterr()`. The errno value in user context will then contain the appropriate value.

Interrupts

When porting Linux device drivers to LynxOS, it is important to understand the differences in how both operating systems handle interrupt requests. Linux has a multi-stage mechanism used to prioritize tasks. LynxOS uses a similar mechanism, but it uses the priority of the interrupt as the basis for prioritization. Later processing within the kernel thread that handles the interrupt provides another level of prioritization.

How Linux Handles Interrupts

Linux provides two types of interrupt service routines: Slow and Fast. Slow interrupt routines can be interrupted by fast routines. Fast routines can only be interrupted if it is enabled in the ISR (Interrupt Service Routine). Linux uses the `cli()` and `sti()` calls to disable and restore interrupts.

How LynxOS Handles Interrupts

LynxOS provides a single interrupt routine, which is prioritized exclusively by the hardware. Interrupts can be interrupted by other interrupts of a higher priority only. LynxOS uses the functions `disable()` and `restore()` to disable and restore interrupts. In addition, LynxOS drivers can use kernel threads for devices with unbounded interrupt latency to create bounded interrupt response.

Registering Interrupts

Interrupts are registered differently on LynxOS and Linux systems.

Registering Interrupts for Linux

Linux uses `request_irq()` function to register both fast and slow interrupt routines. An example prototype of an ISR is:

```
void do_irq(int irq, void *dev_id, struct pt_regs *regs);
```

The function to clear the ISR is called `free_irq()`. Interrupts can also be linked with the `SA_SHIRQ` flag when calling `request_irq()`.

Registering Interrupts for LynxOS

LynxOS registers interrupts with the `iointset()` function. The prototype of the ISR is:

```
void do_irq(char *s);
```

ISRs are cleared with the `iointclr()` function. The address passed to the ISR is generally the address of the static structure of the device driver installed. Interrupts can be linked under LynxOS with `ioint_link()`.

Blocking and Non-Blocking I/O

Both Linux and LynxOS support devices that include blocking and non-blocking I/O. The following table describes the differences between how the two systems handle these features.

Table A-8: Blocking & Non-Blocking I/O Differences

Action	Linux	LynxOS
Ability to select blocking or non-blocking	Read and write can be blocking or non-blocking. This is set when the device is opened.	Read and write can be blocking or non-blocking. This is set when the device is opened.
Non-blocking setting	Set through `O_NONBLOCK` flag passed through read or write in struct of file.	Set through `O_NONBLOCK` flag passed through read or write in struct of file.
Behavior if data is not available	The function `interruptible_sleep_on()` can be set to wait synchronously for data.	A semaphore can be set to wait synchronously with the functions `wait()` or `tswait()`.
Behavior during wait	Task is put into the waiting queue.	The thread is put into a waiting state.
Behavior during data available	The interrupt can call `wake_up()` to allow the task to continue.	The ISR can post the semaphore with `ssignal()` and the thread continues according to its priority.

Bottom-Halves and Kernel Threads

This section outlines the behaviors used in handling prioritized interrupts.

Linux uses a *bottom-half handler* to process an interrupt that requires extensive processing, but is not time critical. Bottom half drivers are installed using `init_bh()` and removed with `remove_bh()`. An ISR can mark the bottom half of an ISR that needs to execute by using the function `mark_bh()`. Once marked, every bottom half handler runs automatically after slow interrupts occur, as well as whenever the scheduler invokes them.

LynxOS uses kernel threads to handle prioritized interrupt processing. Whenever an ISR is too long, its code can be placed into a kernel thread. The ISR can then signal the kernel thread to execute with an `ssignal()` semaphore. Kernel threads can be created with the `ststart()` call and removed with the `stremove()` call.

Kernel threads in LynxOS are scheduled for execution by prioritizing them at half a priority higher than the process which is using them. This is possible because although LynxOS has 256 user priorities, it really has 512 internal priorities to allow for this half-priority increase. The reason for this is to allow thread processing to complete before the user process that requested it runs.

Kernel Support

Memory allocation within the kernel is performed in similar ways.

Table A-9: Memory Allocation Differences

Action	Linux	LynxOS

Get a page	`unsigned long get_free_page(int priority)`
	
	`alloc_page`
		`char *get1page();`
		`void free1page(char *p);`

Get free memory	`void *kmalloc(size_t size, int priority)`
	
	`void * vmalloc(unsigned long size);`
		`char *sysbrk(long size);`
		`void sysfree(char *p, long size)`

Allocate contiguous physical memory	`kmalloc()` with priority set to `GFP_DMA` with `GFP_KERNEL` or `GFP_ATOMIC`	`char *alloc_cmem(int size);`
		`void free_cmem(char *p, int size);`

Kernel Timer Support

Kernel timer support is provided under both systems with the following calls:.

Table A-10: Kernel Timer Support Differences

Action	Linux	LynxOS
Adding a timer	`void add_timer(struct timer_list *timer);`	`int timeout(int (*func()), char *arg, int interval);`
Removing a timer	`int del_timer(struct timer_list *timer);`	`cancel_timeout(int num);`

Semaphore Support

Semaphores are available on both systems.

Linux Semaphore Support

Under Linux, a semaphore is waited on with the `down` system call:

```
void down(struct semaphore *sem);
```

It is signaled with the `up` call:

```
void up (struct semaphore *sem);
```

Tasks blocked that are waiting on a semaphore can be woken up with the `wake_up` call:

```
void wake_up(struct wait_queue *p)
```

LynxOS Semaphore Support

Under LynxOS, semaphores are waited on and signaled with the `swait()`, `ssignal()`, `ssignaln()` (for signaling multiple time) and `sreset()` using the following calls:

```
int swait(int *sem, int flag);
int ssignal(int *sem);
int ssignaln(int *sem, int count);
int sreset(int *sem)
```

Threads and processes waiting on a semaphore can be woken up in priority order with the `sreset()` call. Also, semaphores in LynxOS can be set to operate as priority inheritance semaphores to alleviate problems with priority inversion.

Address Translation

Address Translation for Linux

For Linux, when accessing a virtual user address from within a system call or any interrupt routine, the data needs to be copied from one address space to another. The functions used to copy to and from the kernel are:

- `put_user()` - Copies data from system space to user space
- `get_user()` - Copies data from user space to system space

The functions used to copy to and from user space to system space are:

- `copy_from_user()` - Copy from user space to system space
- `copy_to_user()` - Copies data from system space to user space

The address of the process context can be retrieved from the current pointer of the current process context.

To validate addresses, the function `access_ok()` can be used with the `VERIFY_WRITE` flag to check for write permission. The `access_ok()` function with the `VERIFY_READ` flag can be used to check for read permission.

Address Translation for LynxOS

LynxOS allows direct access to user address space from entry points, as well as from within the system call, without the need for address translation. Interrupts and kernel threads do need to translate a user address to the kernel virtual address space with `get_phys()`. In all cases, user code can never directly access the kernel address space. Additionally, the `currtptr` pointer within the current system call contains the address of the process context.

Address validation under LynxOS is performed with the `rbounds()` and `wbounds()` calls. Users can also use the `NOT_ALIGNED()` function to see if an address is valid for use as anything other than a character pointer (returns nonzero value if this is true).

Driver Problem Reporting

Linux allows you to report problems within device drivers with `sprintf()` and `vsprintf()` to send strings to the console device. LynxOS uses `cprintf()` and `kkprintf()` to perform this same function.

A convention is to use `cprintf()` for error message reporting and `kkprintf()` for debug output.

Communications with Applications

Both Linux and LynxOS allow signals to be sent to user applications from within kernel space. Linux provides `send_sig()`, which can send one of 32 different signals. LynxOS uses `_kill()` or `_killpg()` (for a group of processes). For LynxOS, 64 different signals are supported (required by the POSIX API.)

Scheduling Differences

LynxOS differs from Linux in the way that it schedules the handling of interrupts. There is also a difference in the way schedules and threads are processed. These differences can affect the way in which drivers must be written to respond to user applications.

Linux Scheduling

Linux is designed as a “fair share” scheduling model. Linux tasks can have a priority associated with them, but this is not used as an absolute determinant of the process priority. The “real” priority of a process (what the scheduler uses to determine what to schedule next) is completely dynamic on a Linux system. The scheduler keeps track of the processes that have been running, and which processes have been denied running. The scheduler then attempts to balance the execution time for each. For example, if a task is running for a length of time, the scheduler lowers its “real” priority, allowing other waiting tasks to run.

Linux also distinguishes between tasks that perform different kinds of activities and attempts to grant them CPU time accordingly.

For **interactive processes** (ones that interact with users), the wakeup time must be short. This is important, because these kinds of processes spend much of their time waiting for input from mice, command shells, etc. Typically, the average delay in waking these kinds of processes up must fall between 50 and 150 ms to keep up with users.

For **batch processes**, a rapid wakeup time is not required, as these typically run in the background. Because these processes can afford to wait, they are often the first ones to be penalized by the scheduler to maintain responsiveness for the interactive processes.

For **real-time processes**, extremely strong scheduling requirements are enforced. These processes can never be blocked by lower-priority processes and must always be responded to in a very short time. Also, the variance of the response time should be minimal. Examples of these kinds of tasks include sound applications, and data collection and control.

The Linux scheduler generally behaves in the following way:

1. Initialization, static priority is assigned by the user and the dynamic priority is equal to the static priority.
2. For each clock tick (occurring at 10 ms):

- The dynamic priority is decremented.
- The “goodness”value is computed (it is equal to the sum of the static and dynamic priorities).
  - If the dynamic priority decreases to 0, than the goodness value decreases to 0 as well.

3. When the scheduler is invoked, it gives the CPU time to the task with the highest “goodness” (`need_resched()`, `dynamic = 0`, block/yield)
4. When all tasks reach Dynamic = 0, all dynamic priorities are re-initialized to their static value and all tasks have the chance to run their time quantum.
5. When the real-time flag is set for a task, it implies that its goodness value is always be kept high.

At the heart of this scheduling algorithm, the notion of a “goodness” value for each task is what controls what tasks run when. As previously mentioned, Linux behaves differently based on the type of task it is running. The goodness value determines this behavior. Here are the different actions taken when the value of goodness changes (c is the value returned by the goodness() call):

- **c= -1000**

  This task must never be selected. This value is returned when the run queue list contains only the `init_task`.

- **c = 0**

  The task has exhausted its run time quantum. Unless this task is the first task in the run queue list and all the other runnable tasks have exhausted their quantum, it will not be selected for execution.

- **0 < c < 1000**

  The task is a conventional task that has not exhausted its quantum. Note that a higher value of **c** denotes a higher level of goodness.

- **c >= 1000**

  The task is a real-time process because the goodness value is very high.

LynxOS Scheduling

LynxOS uses a strict round-robin scheduler with fixed priority levels (there is a slight exception to this rule for priority inheritance scheduling). There are 256 priority levels in comparison to Linux's 99 (although Linux can also have negative priority levels). Kernel threads are scheduled in the global scheduling space along with user processes and user threads. Priority tracking and priority inheritance are also supported.

In general, LynxOS schedules processes and threads (tasks) in a strict priority sense. There is no other scheduling criteria for a process other than its priority. Tasks with a high priority ready to run are allowed to run immediately, preempting lower-priority running tasks. Also, the period of time that a high priority task takes to begin running is guaranteed to be bounded.

If a priority inheritance semaphore is used, the scheduler will alter the priority of the tasks involved by temporarily incrementing the priority of a high priority task that is waiting on a resource locked by a priority inheritance semaphore in the possession of a lower priority task. This keeps the high priority task from being denied by a medium priority task that preempts the lower priority task to keep it from completing its use of the resource.

The LynxOS scheduler runs each task within a priority level in turn for the period of time specified by that level's quantum (this is user-modifiable). If all the tasks within a particular priority level are waiting for I/O, tasks from the next lower priority queue are run.

Like Linux, tasks can voluntarily yield the CPU by using the `yield()` call.

Differences in Setting up a Driver

Setup

For Linux, the `setup()` function can be used to pass device-specific data to a driver for initialization. For LynxOS, the convention is to declare for every driver a structure that contains all the values that the device needs to be initialized with. The LynxOS `install()` entry point allocates memory for the driver and performs device initialization. The address of the driver `info` structure will then be passed automatically by the kernel to all the other entry points.

Installation

For Linux, a driver is registered within the kernel with the `register_chrdev()` or `register_blkdev()`. These functions are used by the Linux `init()` function.

The major number can be choose or the kernel will find the highest free available one. The `init()` call also should check to see if the device is present.

LynxOS allows the device driver to be performed either statically or dynamically. The major device number cannot be choose, it is dictated by the kernel that finds the highest free available one. The LynxOS `install()` routine checks to see if the device is present and available.

Device Access: open() and close()

Linux passes the `open()` call the following information:

```
int open(struct inode *inode, struct file *file);
```

Here, the `inode` value contains the node information for device access.

The `file` variable contains the access mode, position in the device and the functions that can be used in the inodes.

`open()` should return a 0 (for success) or an error value on failure.

The `release()` call is used to close the device:

```
void release(struct inode *inode; struct file *file);
```

LynxOS operates similarly, but the data structure for the device is passed to the `open()` and `close()` calls directly:

```
int open(char *s, struct file *file);
```

Here, `s` is passed by the kernel automatically and is the address for the data structure returned from install().

The file variable contains the access mode, position in the device and the major and minor numbers of the device.

LynxOS device drivers are closed with a `close()` call:

```
int close(char *s, struct file *file);
```

Both `open()` and `close()` should return `OK` or `SYSERR`.

Device Access: read() and write():

Reading and writing are performed with similar calls on both systems. For Linux, `read()` is called as follows:

```
int read(struct inode *inode, struct file *file, char
*buffer, int count);
```

This call should return the number of bytes read or an error.

`write()` is called as follows:

```
int write(struct inode *inode, struct file *file, char
*buffer, int count);
```

It should return the number of bytes written or an error.

Data in both cases needs to be copied from one address space (user) to another (system or kernel space of the driver).

LynxOS operates similarly, with the `read()` call declared as follows:

```
int read(char *s, struct file *file, char *buffer, int
count);
```

This call should return the number of bytes read or an error.

For the `write()` call:

```
int write(char *s, struct file *file, char *buffer, int
count);
```

This call should return the number of bytes written or an error.

The entry points can directly use the buffer address to access count data.

Device Access: Control

The `ioctl()` and `select()` calls are universal in both LynxOS and Linux, as well as the calls for device control. The `ioctl()` call allows control of a device and `select()` allows you to wait on multiple channels of the device.

For Linux, `ioctl()` is called as follows:

```
int ioctl(struct inode *inode, struct file *file,
unsigned int cmd, unsigned long arg);
```

This call should return a 0 or an error.

The `select()` call is used as follows:

```
int select(struct inode *inode, struct file *file, int
sel_type, slect_table *wait);
```

It should return a 0 or 1 when one of the devices becomes available.

LynxOS is similar, with the exception of passing the control structure first for `ioctl()`:

```
int ioctl(char *s, struct file *file, int cmd, char
*arg);
```

This should return `OK` or `SYSERR`.

For `select()`:

```
int select(char *s, struct file *file, int which, struct
sel *ffs);
```

Return value should be `0` or `SYSERR`.

APPENDIX B Porting UNIX Drivers to LynxOS

This appendix discusses the similarities and differences between device drivers under LynxOS and UNIX. It is intended to serve two main purposes:

- To provide some guidelines to developers wishing to port existing UNIX drivers to LynxOS in order to reuse existing code.
- To provide a pedagogical stepping stone for developers who are already experienced with UNIX drivers.

The material that follows describes a feature of a UNIX device driver and points out the corresponding feature in a LynxOS device driver. This appendix supplements the more detailed coverage of LynxOS device drivers that can be found in previous chapters of this manual. Certain LynxOS features not used in a UNIX driver are, consequently, mentioned very briefly only. The versions of the UNIX kernel referred to in this chapter are, for the most part, SVR3.2 and SVR4.

Kernel and Driver Design Goals

A frequently asked question is whether it would be possible to achieve source-level or even binary compatibility between UNIX and LynxOS drivers. While this--with some effort--might be technically feasible, the result would probably not be acceptable for designers of real-time systems.

This is because the fundamental differences in the design goals of LynxOS as compared to the UNIX kernel. The latter was designed for multi-user time-sharing systems, while LynxOS was designed specifically for hard real-time systems. These differences in design goals influence the choice of kernel data structures and algorithms, including those used in device drivers.

The differences are also seen in the services provided by the kernel to device drivers. The LynxOS kernel provides many services that meet specific requirements of real-time systems. These features would not be found in a UNIX driver. On the contrary, a UNIX driver may use some services that would result in a detrimental effect on a real-time performance.

Another significant difference is preemptability. The UNIX kernel was originally written to be uninterruptible, though some UNIX kernels now exist that are preemptive to some extent. The LynxOS kernel, including device drivers, is fully preemptive. This has a major influence on the way a driver is written.

Different design goals can also be noted at the level of the drivers themselves. UNIX drivers are generally designed to make the most efficient use of I/O devices, thereby maximizing throughput. This goal leads to the use of specific driver techniques such as the chaining of I/O requests, processing of interrupts within an interrupt handler, and the starting of the next I/O operation from within the interrupt handler. In contrast, a LynxOS driver must be designed to have a minimal impact on real-time performance, respecting the relative priorities of the tasks that are using the devices. The way in which interrupts are handled is probably the largest difference between a UNIX and a LynxOS driver.

Given these differences, both at the kernel and driver level, it is clear that in order to respect real-time demands, a port is preferred in providing compatibility.

Porting Strategy

Porting a UNIX driver can be broadly divided into three stages as follows:

- Stage One
    - Driver interface with kernel
    - Driver interface with application
    - The U structure
    - Reentry and synchronization
- Stage Two
    - System threads and priority tracking
- Stage Three
    - Dynamic installation
    - POSIX programming model

The first stage allows the developer to reach a point where a working LynxOS driver can be tested for functionality. While enabling the re-use of a driver in a relatively short time, this initial port does not take advantage of the real-time aspects of LynxOS, and the driver could have a detrimental effect on the system response time. In order for the driver to conform to the real-time characteristics of LynxOS, the implementation of Stage Two is absolutely necessary. The features in Stage Three are optional but may be advantageous in certain situations.

Driver Structure

Overall Structure

A LynxOS and a UNIX driver are quite similar in overall structure. Each consists of a number of entry points, including an initialization routine and an interrupt handler. A LynxOS driver has, in addition, one or more kernel threads.

Table B-1: UNIX v/s LynxOS Structure

LynxOS Driver	UNIX Driver
Initialization	Initialization
Entry points	Entry points
Interrupt handler	Interrupt handler
Kernel threads	

Global Variables

A UNIX driver typically makes widespread use of global variables, which is the most common way for the driver entry points to share information. A LynxOS driver can and should be written without the use of any global variables. The LynxOS kernel provides an elegant means to communicate driver state between entry points. Use of this mechanism is essential to allow dynamic install and uninstall of a driver.

Major and Minor Device Numbers

There is an important difference in the way UNIX and LynxOS use major device numbers. Under UNIX, the major device number is used to distinguish between different drivers. The minor number distinguishes between different devices controlled by the same driver. Under LynxOS, each driver has a unique driver ID, though this number is never used by the driver code. Different devices controlled by the same driver are identified by different major numbers (as opposed to the minor number in UNIX). The use of the minor device number is defined entirely by the driver. LynxOS driver IDs and major numbers are allocated automatically during a kernel build.

Driver Interface with Kernel

The interface between the UNIX kernel and a driver is defined by the driver service calls, the `init` entry point, and the interrupt handler.

Driver Service Calls

The services provided by a kernel to device drivers can be grouped into several functional classes:

- Memory Management
- Synchronization
- DMA Transfers and Raw I/O
- Block I/O
- Driver Debugging

Memory Management

This section describes the functions used for allocating memory and for translating memory addresses.

Memory Allocation

Functions used for the allocation of memory for the driver's internal use are as follows:

Table B-2: Internal Use Memory Allocation Functions

LynxOS	UNIX
`sysbrk`, `sysfree`, `get1page`, `free1page`, `alloc_cmem`, `free_cmem`	`kmem_alloc`, `kmem_free`

The functions `sysbrk` and `sysfree` are the nearest equivalent to UNIX `kmem_alloc` and `kmem_free`. The UNIX function `kmem_alloc` can sleep while waiting for free space. The LynxOS functions never sleep, instead, they return `SYSERR` if the memory request cannot be satisfied immediately.

Address Translation

The functions required for converting virtual to physical addresses are as follows:

Table B-3: Virtual to Physical Address Conversion Functions

	LynxOS	UNIX
User virtual to physical	`mmchain, mmchainjob`	`vtop`
Kernel virtual to physical	`mmchainjob (job 0) addr – PHYSBASE`	`kvtophys`

Note that `mmchain` returns a kernel virtual address. To convert this to a physical address, the constant `PHYSBASE` must be subtracted.

Synchronization

In non-preemptive UNIX kernels, synchronization is a fairly straightforward matter. But in a fully preemptive kernel such as LynxOS it is much more complex. This can represent a significant portion of the porting effort. For more information, see Chapter 4, "Synchronization."

DMA Transfers and Raw I/O

Setting up DMA transfers requires the following kernel services:

- Memory locking
- Split transfer into physically contiguous pieces
- Virtual to physical address translation

The following code fragments illustrate typical SVR4 driver code for performing a DMA transfer to user space.

UNIX

```
read (dev, uio)
dev_t dev;
struct uio *uio;
{
    physiock (mybreak, 0, dev, B_READ, nblocks, uio);
}

mybreak (bp)
struct buf *bp;
{
    dma_pageio (mystrategy, bp);
}

mystrategy (bp)
struct buf *bp;
{
    physaddr = vtop (bp->b_addr, bp->b_proc);
    /* start DMA transfer */
}
```

The key functions in the previous code fragment are:

`physiock`	Faults in and locks memory pages.
`dma_pageio`	Breaks transfer blocks into 512 byte blocks and calls `strategy` routine.
`mystrategy`	Converts user virtual address to physical address, sets up and initiates DMA transfer (user written).

LynxOS

```
read (s, f, buff, count)
struct statics *s;
struct file *f;
char *buff;
int count;
{
    struct dmachain *array;
```

```
        int np, nc, pid, i;

        pid = getpid ();
        np = npages (buff, count);
        array = (struct dmachain *) sysbrk (np * sizeof
                  (struct dmachain));
        mem_lock (pid, buff, count);
        nc = mmchain (array, buff, count);
        for (i = 0; i < nc; i++) {
            /*
             * Do DMA transfer at physical address
             * array[i].address, length array[i].count
             */
            array[i].address -= PHYSBASE;
                                  /* convert to physical address */
            do_dma (&array[i]);   /* user supplied routine */
        }
        sysfree (array, np * sizeof (struct dmachain));
        mem_unlock (pid, buff, count, TRUE);
    }
```

The key functions in the previous code fragment are:

`mem_lock`	Faults in and locks pages.
`mmchain`	Converts virtual address range to list of kernel virtual addresses. These are converted to physical addresses using `PHYSBASE`
`mem_unlock`	Unlocks memory pages.

Note that whereas UNIX uses the block interface (`strategy` entry point) for raw I/O, LynxOS uses the character interface read and `write` entry points.

Block Input/Output

strategy Entry Point

Both UNIX and LynxOS block drivers have a `strategy` entry point that is called by the kernel's block buffering I/O subsystem to perform transfers to block devices.

Table B-4: Strategy Entry Point Comparison

LynxOS	UNIX

`strategy (s, bp)`
`struct statics *s;`
`struct buf_entry *bp;`
	`strategy (bp)`
	`struct buf *bp;`

As with other entry points, the LynxOS strategy routine is passed the address of the device’s statics structure as the first argument.

buf Structure

This data structure defines the buffers that are used to hold the data blocks from a block device. In LynxOS, this structure is of type `struct buf_entry`. The correspondence between the fields is shown below.

Table B-5: buf Data Structure Comparison

LynxOS struct buf_entry	UNIX struct buf
`int b_status`	`int b_flags`
`struct buf_entry *av_forw`	`struct buf *av_forw`
`struct buf_entry *av_back`	`struct buf *av_back`
`int b_device`	`o_dev_t b_dev`
	`unsigned b_count`
`char *memblk`	`caddr_t b_addr`
`long b_number`	`daddr_t b_blkno`
	`char b_oerror`
	`unsigned int b_resid`
	`clock_t b_start`
	`struct proc *b_proc`
	`struct page *b_pages`
	`long b_bufsize`
	`int (*b_iodone)()`
	`struct vnode *b_vp`
	`int b_error`
	`dev_t b_edev`

The symbolic constants used to specify bits in the `b_flags` field are shown below.

Table B-6: b_flags Field Comparison

LynxOS	UNIX
B_BUSY	B_BUSY
B_DONE	B_DONE
B_ERROR	B_ERROR
	B_PAGEIO
B_PHYS	B_PHYS
B_READ	B_READ
B_WANTED	B_WANTED
B_ASYNC	B_ASYNC

Block I/O Support Routines

UNIX provides a number of support routines for block device drivers.

biowait	Suspend, waiting for I/O completion
biodone	Wakeup process and release buffer
brelse	Put buffer back on free list

The following code fragment shows how these routines are typically used in the `strategy` entry point and interrupt handler of a UNIX driver.

UNIX

```
xx_strategy (bp)
struct buf *bp;
{
    /* start transfer on device ... */
    ....
    /* if transfer is asynchronous, return, else wait
       for completion */

    if (bp->flags & B_ASYNC)
        return (0);
    biowait (bp);
}

xx_intr ()
{
    if (error_condition)
        bp->b_error |= B_ERROR;
    biodone (bp);                  /* wake up process */
}
```

LynxOS does not provide the `biowait` or `biodone` routines, but the code to implement the required functionality is straightforward, as shown below.

LynxOS

```
strategy (s, bp)
struct statics *s;
struct buf_entry *bp;
{
    /* start transfer on device ... */

    swait (&s->devsem);  /* wait for device completion */
    bp->b_status |= B_DONE;
               /* set bits to indicate transfer status */
    if (s->error)
        bp->b_status |= B_ERROR;
    if (bp->b_status & B_ASYNC) {
              /* if async transfer, release buffer ... */
        ssignal (&bp->b_rwsem);
        brelse (bp);
    } else
        ssignal (&bp->b_rwsem);
                       /* ... else wakeup waiting task */
}
```

Driver Debugging

UNIX

`printf`	Print message on system console (uses polling)
`uprintf`	Print message on user terminal (uses driver)

LynxOS

`kprintf`	Print message on debug console (uses polling)
`cprintf`	Print message on system console (uses driver)

Initialization Routine

Although both UNIX and LynxOS drivers have an initialization routine, the way in which they are used differs in some important ways. By convention, the UNIX routine is called *xxx*`init`, in LynxOS `xxxinstall`.

UNIX

- Initialization is called once during bootup.
- Initializes all hardware and software.
- Device-specific information is kept in statically allocated structures.
- Maximum number of supported devices is hardcoded.
- Limited number of configuration parameters

LynxOS

- Initialization routine is called for every major device.
- Device structure allocated dynamically.
- Number of supported devices not limited.
- User-defined configuration parameters

Probing for Devices

One of the tasks usually performed by the initialization routine is to test for the presence of a device. UNIX drivers must handle bus errors. In LynxOS this is handled automatically. Typical UNIX and LynxOS code is illustrated below:

UNIX init Routine

```
#define MAX_CONT 4 /* no. of supported controllers */
struct csb csb[MAX_CONT]; /* controller status blocks */

xx_init ()
{
    for (i = 0; i < xx_ccnt; i++) {
        if (setjmp (u.u_tsav) == 0) {
            u.u_nofault = TRUE;
            ......             /* touch the device here */
            u.u_nofault = FALSE;
            ......  /* Initialize hardware and software */
        } else
            xx_addr[i] = 0;      /* device not present */
    }
}
```

LynxOS install Routine

```
xx_install (info)
struct xxinfo *info;
           /* user defined configuration parameters */
```

```
{
    ....   /* Touch device here */
           /* If we get here, we know device is present */
    s = (struct statics *) sysbrk (sizeof (struct
            statics s));
    ....   /* Initialize software and hardware */
    return (s);
}
```

Interrupt Handling

In SystemV, the details of a device's interrupt capabilities are defined statically in configuration files external to the driver. The name of the interrupt handler is `xxx_intr`, where `xxx_` is the specified driver prefix.

Because LynxOS supports dynamic driver installation and deinstallation, attaching and detaching an interrupt handler is done within the driver code using the functions `iointset()` and `iointcl()`. This is done in the `install()` and `uninstall()` entry points. The device's interrupt vector is normally passed to the install routine in the device information structure.

For x86

```
iointset (32 + info->vector, intr_handler, s);
```

U Structure

Unlike most UNIX kernels, LynxOS does not have a `U` structure. The following paragraphs discuss the most commonly used members of this structure and how the equivalent functionality is implemented in a LynxOS driver.

u_base, u_count, u_offset

Older versions of UNIX used these fields to specify the details of a data transfer in the `read`/`write` entry points. The driver modifies these during the course of the transfer. The return value received by the application is the initial `u_count` value minus its final value. More recent implementations of UNIX have replaced them with a `uio` structure.

In a LynxOS driver, the user buffer address and size are passed directly as arguments to the driver entry point. An important difference from UNIX is that the value returned to the application is the value returned by the driver entry point. The seek position on the device is specified by the field `position` in the file structure. The driver is responsible for setting this at the end of a transfer.

u_fmode

This field holds the file mode flags. Its main use is in the read/write entry points to test for non-blocking I/O. It is also used to test, for example, if an application is trying to read from a device opened in write only mode.

In LynxOS, the file mode is held in the `access_mode` field of the file structure.

u_error

This field contains an error code, which is copied to the application's `errno` variable.

A LynxOS driver specifies an error code with the `pseterr()` function.

u_segflg

This field indicates whether a data transfer is to or from user or kernel space. It is necessary to know this because the user process and kernel have separate virtual address spaces.

In LynxOS, the user process and kernel exist within the same virtual space, so this functionality is not required.

u_procp

This field is a pointer used to process table entry for the current process. UNIX device drivers seldom need to access this field explicitly. In LynxOS, each process is identified by a unique job number which can be accessed in the driver top-half routines to provide similar functionality. The function `getpid()` can also be used to find the process ID number.

u_tsav, u_nofault

These are used for trapping bus errors, typically in the `init()` routine.

In the install routine of a LynxOS driver, bus errors are handled automatically. Elsewhere in a driver, the routines `noreco()` and `recoset()` must be used to catch bus errors.

Reentrance and Synchronization

Critical Code Regions

Accesses to shared data structures and hardware registers must be serialized. The synchronization mechanisms used in a UNIX driver depend very much on whether the driver is preemptive. SVR4 driver code is not preemptive, though synchronous preemption is possible if a driver calls `sleep()`. Drivers written for such kernels only need to synchronize with the interrupt level routines. This is done with the `spl`*n* and `spl`*x* functions. The LynxOS equivalent of these functions are `disable()` and `restore()` although there is an important difference. The LynxOS functions disable and restore all interrupts, but interrupt nesting is not possible.

Drivers under LynxOS are fully preemptive. Appropriate synchronization must be added to make the driver reentrant.

Event Synchronization

This type of synchronization involves waiting for an event (buffer free, transfer complete, data ready, and so on) to occur.

LynxOS	`swait/ssignal`
UNIX	`sleep/wakeup`

The UNIX `sleep()` function specifies a priority, which is assigned to the process when it wakes up. LynxOS uses fixed scheduling priorities. A task priority can only be changed on request from the user application. Both UNIX `sleep` and LynxOS `swait()` use an argument to specify how signals are handled during the time the task is blocked. It is difficult to find an exact correspondence in behavior in all cases.

UNIX sleep Priority

`<= PZERO`	Signals are ignored. Use the symbolic constant `SEM_SIGIGNORE` with `swait`.
`> PZERO`	Signals are delivered but `sleep` never returns. The nearest equivalent with `swait` is to use the symbolic constant `SEM_SIGRETRY`. However, the `swait` is automatically restarted and eventually returns.
`> PZERO | PCATCH`	`sleep` is aborted and returns 1 on receipt of a signal. The LynxOS equivalent is to use the symbolic constant `SEM_SIGABORT`. However, `swait` returns a non-zero value (not necessarily 1).

Another important difference is that `wakeup()` is stateless. It can only wake tasks that are blocked on the event at the time that `wakeup()` is called. On the other hand, `ssignal()` has a counter associated with it. This difference can have an influence on driver design. More care is needed with synchronization in the stateless case. Though this problem is normally solved by the fact that a UNIX driver is not preemptive.

Driver Interface with User Applications

The driver interface with the application covers the following topics:

- Driver entry points
- Accessing user address space
- Returning errors

Entry Points

There are a number of general remarks that can be made that apply to all entry points.

- In a LynxOS driver the first argument to all entry points is a pointer to the `statics` structure allocated by the `install` routine.
- LynxOS does not use the UNIX `cred_t` credentials structure.

- In LynxOS, the device number is passed only to the `open` entry point. Other entry points can access the device number and access the mode in the `file` structure.

```
    int flag = f->access_mode;
    int dev = f->dev;
```

Major and Minor Device Numbers

As discussed above, there is an important difference in the way UNIX and LynxOS use the device numbers. Typically, a UNIX driver uses (part of) the minor number to index into an array containing state variables for each device, as illustrated below.

UNIX

```
/* number of supported controllers */
#define MAX_CONT 4

/* controller status blocks */
struct csb csb[MAX_CONT];

/* number of configured controllers */
extern int xxx__ccnt;

xxx_open (dev, mode, otyp, cred)
dev_t *dev;
int mode;
int otyp;
cred_t *cred;
{
    struct csb *csbp;
    int cntlr;

    cntlr = getminor (*dev) & 0xf;
    if (cntlr >= xxx__ccnt || cntlr >= MAX_CONT)
        return (ENXIO);
    csbp = &csb[cntlr];
```

This code is unnecessary in the LynxOS driver because the address the controller’s status block is passed as an argument to the entry points.

LynxOS

```
xxx_open (s, dev, f)
struct statics *s;
int dev;
struct file *f;
```

NOTE: UNIX drivers usually use the term controller status block and use a statics structure. They are more or less the same thing.

open/close

Table B-7: open/close Comparison

UNIX	LynxOS

`open (dev,mode,otyp,cred)`
`dev_t *dev;  /* SVR4 */`
`int mode;`
`int otyp;`
`cred_t *cred;`
 	
`close (dev,mode,otyp,cred)`
`dev_t dev;`
`int mode;`
`int otyp;`
`cred_t *cred;`
	`open (s, dev, f)`
	`struct statics *s;`
	`int dev;`
	`struct file *f;`
	
	`close (s, f)`
	`struct statics *s;`
	`struct file *f;`

As shown in the listing above:

- LynxOS passes the device number to open, like SVR3. SVR4 passes a pointer to the device number.
- LynxOS does not have an equivalent of the `otyp` field.
- The LynxOS kernel only calls the `close` entry point on the last close of a device.

read/write

Table B-8: read/write Comparison

LynxOS	UNIX

`read (s, f, buff, count)`
`struct statics *s;`
`struct file *f;`
`char *buff;`
`int count;`
	`read (dev, uiop, credp)`
	`dev_t dev;`
	`uio_t *uiop;`
	`cred_t *credp;`

The UNIX `uio` structure specifies a list of user buffers. Earlier UNIX kernels used the `clist` data structure for character storage.

In a LynxOS driver, the user buffer is specified by `buff` and `count`. The entry point is called once for each buffer in scatter/gather I/O (`readv`/`writev`). LynxOS does not use the `clist` data structure.

The following code fragments compare typical `write` entry point logic used to transmit all user data. Note that in LynxOS, the driver is responsible for positioning the seek pointer (`f->position`). Another important difference is that the UNIX driver returns the number of bytes not transmitted.

Table B-9: write Comparison

LynxOS	UNIX
`for (i = 0; i < count; i++)`	`while ((c = uwritec (uio)) >=0)`
`  transmit (buff[i]);`	`  transmit (c);`
`f->position += count;`	`return (0);`
`return (count);`	

ioctl

Table B-10: ioctl Comparison

LynxOS	UNIX
`ioctl (s, f, cmd, arg)`	`ioctl(dev, cmd, arg, mode,`
`struct statics *s;`	`      cred, rval)`
`struct file *f;`	`dev_t dev;`
`int cmd;`	`int cmd;`
`int arg;`	`int arg;`
	`int mode;`
	`cred_t *credp;`
	`int *rval;`

If `arg` is a pointer, the LynxOS driver must check the validity of the address with `rbounds()` and `wbounds()`.

select

Table B-11: select Comparison

LynxOS	UNIX
`struct statics {`	`extern int selwait;`
`...`	`struct proc *selr, *selw;`
`    int space_free;`	`select (dev, rw)`
`    /* for output */`	`dev_t dev;`
`    int data_ready;`	`int rw;`
`    /* on input */`	`{`
`    int *rsel_sem, *wsel_sem;`	`switch (rw) {`
`};`	`    case FREAD:`
`select (s, f, which, ffs)`	`        selr = u.u_procp;`
`struct statics *s;`	`        break;`
`struct file *f;`	`    case FWRITE:`
`int which;`	`        selw = u.u_procp;`
`struct sel *ffs;`	`        break;`
`{`	`  }`
`switch (which) {`	`  return(0);`
`    case SREAD:`	`}`
`        ffs->iosem = &s->data_ready;`	`/*Data Input */`
`        ffs->sel_sem = &s->rsel_sem;`	`if (selr) {`
`        break;`	`        selwakeup (selr, coll);`
`     case SWRITE:`	`        selr = 0;`
`        ffs->iosem = &s->space_free;`	`}`
`        ffs->sel_sem = &s->wsel_sem;`	`/* Data Output */`
`        break;`	`if (selw) {`
`    case SEXCEPT:`	`        selwakeup (selw, coll);`
`        return (SYSERR);`	`        selw = 0`
`}`	`}`
`return(OK);`	
`}`	
`s->data_ready = 1;`	
`disable (ps);`	
`if (s->rsel_sem)`	
`        ssignal (s->rsel_sem);`	
`restore (ps);`	
`s->space_free = 1;`	
`disable (ps);`	
`if (s->wsel_sem)`	
`        ssignal (s->wsel_sem);`	
`restore (ps);`	

Accessing User Space

UNIX

The currently executing user process and the kernel may have separate virtual address spaces. In this case, kernel service routines are used to transfer data to and from user space. These routines usually handle invalid user addresses.

LynxOS

The current user process and the kernel exist in same virtual space. The kernel (including drivers) can access the whole of the virtual space. Therefore, drivers can transfer data to and from user space directly using a pointer.

The following code fragments illustrate how data might be transferred from user space in an `ioctl` entry point.

Table B-12: ioctl Data Transfer Example

LynxOS	UNIX

`if (rbounds (useraddr) < nbytes) {`
`pseterr (EFAULT);`
`    return (SYSERR);`
`}`
`while (nbytes--)`
`        *kernaddr++=*useraddr++;`
	`char *useraddr, *kernaddr;`
	`int nbytes;`
	`if (copyin (useraddr, kernaddr,`
	`nbytes) == -1)`
	`    return (EFAULT);`

Returning Errors to User Application

UNIX

Earlier versions used the `u_error` field in the **u** structure. SVR4 uses the entry point return value.

LynxOS

Uses the `pseterr` function and return the value `SYSERR`.

The following code fragments illustrate how a driver returns the error `EIO`:

Table B-13: Returning Errors

LynxOS	UNIX SVR3	UNIX SVR4

`pseterr (EIO);`
`return (SYSERR);`
	`u.u_error = EIO;`
	`return;`
		`return (EIO);`

LynxOS Kernel Threads

When using kernel threads, interrupt processing is performed by a preemptive, prioritized task. This is essential in order to maintain deterministic system response times. Using the UNIX interrupt architecture, where all interrupt processing is done in the interrupt handler itself, will lead to a degradation of the system's real-time performance.

Dynamic Installation

LynxOS supports the dynamic installation and deinstallation of drivers. This greatly facilitates the driver development and debugging phases as a kernel rebuild and reboot is not necessary each time the driver is modified. If the port has been done correctly, the only addition required to support dynamic installation is the declaration of the `entry_points` structure.

POSIX Programming Model

The LynxOS implementation of the POSIX.1 and POSIX.1b features permit much simpler driver design for supporting asynchronous I/O, non-blocking I/O, and synchronous I/O multiplexing and polling.

Asynchronous I/O

The complexity of handling asynchronous transfers is hidden from the application and driver developer. The POSIX API provides services to the application developer, and the driver sees only synchronous requests. Therefore, code to handle asynchronous transfers can be removed from a UNIX driver if the LynxOS version is only intended for use with POSIX conforming applications.

Synchronous I/O Multiplexing and Polling

This functionality is provided by the `select` system call at the application level and the `select` entry point in a driver. The POSIX standard does not define a `select` function. So, if the LynxOS driver is only intended for use with POSIX conforming applications, the `select` entry point can be removed.

APPENDIX C Sample Device Driver

Header Files

ptrinfo.h

```
/* ptrinfo.h */
struct ptrinfo {
  int port;
};
```

prtioclt.h

```
/* ptrioctl.h */
  #define PTRSTATUS 500
  struct ptrstatus {
  int chars;     /* characters printed */
  int lines;     /* lines printed      */
};
```

Driver Code

```
/* ptrdrvr.c - using threads */
#include <kernel.h>
#include <mem.h>
#include <sys/file.h>
#include <errno.h>
#include <sys/ioctl.h>
#include <conf.h>
```

```
#include <st.h>
#include "ptrinfo.h"
#include "ptrioctl.h"

/* ports  */

#define PP_DATA   0     /* data port offset */
#define PP_STATUS   1   /* status port offset */
#define PP_CONTROL  2   /* control port offset */

/* status bits */

#define PP_BUSY     0x80   /* printer busy */
#define PP_PE       0x20   /* out of paper */
#define PP_SLCT     0x10   /* printer is selected */
#define PP_ERROR    0x08   /* printer detected error */

/* control bits */

#define PP_IENABLE 0x10     /* interrupt enable */
#define PP_SLCTIN  0x08     /* select printer */
#define PP_INIT    0x04     /* start printer */
#define PP_AUTOLF  0x02     /* auto line feed */
#define PP_STROBE  0x01     /* strobe printer */

#define port_in(addr) __inb(addr)
#define port_out(data,addr) __outb(addr,data)

typedef unsigned short ptype;

#define STACKSIZE PAGESIZE

struct qentry {
char c;          /* character */
int pri;         /* its priority */
};

struct ptrstatics {
ptype datap;             /* data port address */
ptype controlp;          /* cntrl port address */
char control;            /* control bits */
int irq;                 /* IRQ number */
int closing;             /* closing device */
int close_sem;           /* sempahore for close */
int expecting;           /* expecting an int.? */
int nextnl;              /* output a ’\r’ next? */
int chars;                /* printed since open */
int lines;                /* printed since open */
int qlen;                 /* characters in queue */
struct qentry *q;         /* the queue itself */
int head;                 /* head of queue */
int tail;                 /* tail of queue */
int qdata;                /* data in the queue */
int free_sem;             /* free queue space */
int stid;                 /* thread id */
int int_sem;              /* interrupt semaphore */
int qsem;                 /* queue protection */
struct priotrack pt;      /* pri. tracking */
int curpri;               /* current priority */
int prio_sem;             /* pri. trking sem */
};

/*
static port_in(), port_out();

asm {
  port_in:           /* byte = port_in(port) *
  mov EAX, 0
  mov EDX, 4[ESP]
  in  AL, DX
  ret

  port_out:         /* port_out(byte, port) *
  mov EDX, 8[ESP]
  mov EAX, 4[ESP]
  out DX, AL
  ret
}
*/

#define PERR (char *) SYSERR

char *ptrinstall(info)
struct ptrinfo *info;
{
  struct ptrstatics *s;
  static void ptrint(), ptrthread(), ptruninstall();
  int i;

/* probe for the printer */

  port_out(1, info->port+PP_DATA);
```

```
    if (port_in(info->port+PP_DATA) != 1)
      return PERR;
    s = (struct ptrstatics *)
    sysbrk((long)sizeof *s);
    if (!s) return PERR;
      s->q = (struct qentry *)
    sysbrk((long)info->qlen * sizeof(struct qentry));
    if (!s->q) {
      sysfree(s, (long)sizeof *s);
    return PERR;
  }

  /* initialize statics */

    s->datap = info->port + PP_DATA;
    s->controlp = info->port + PP_CONTROL;
    s->control = PP_SLCTIN | PP_INIT;
    s->irq = info->irq;
    s->expecting = 0;
    s->lines = s->chars = 0;
    s->closing = s->close_sem = 0;
    s->nextnl = 0;
    s->free_sem = s->qlen = info->qlen;
    s->qdata = s->head = s->tail = 0;
    s->int_sem = 0;
    s->qsem = -1;
    bzero(&s->pt, sizeof(struct priotrack));
    s->curpri = 0;
    s->prio_sem = -1;

    /* initialize printer */

    iointset(32+s->irq, ptrint, s);
    port_out(PP_SLCTIN, s->controlp);
    for (i = 0; i < 100; i++) ;
      port_out(s->control, s->controlp);

    s->stid = ststart(ptrthread, STACKSIZE,
                      s->curpri, "ptr thread", 1, s);
    if (s->stid == SYSERR) {
      ptruninstall(s);
      return PERR;
    }
  return (char *) s;
  }
```

```
int ptruninstall(s)

     struct ptrstatics *s;
{
  if (s->stid != SYSERR) stremove(s->stid);
  iointclr(32+s->irq);
  sysfree(s->q, (long)s->qlen *
     sizeof(struct qentry));
  sysfree(s, (long)sizeof *s);
}

int ptropen(s, devno, f)
     struct ptrstatics *s;
     int devno;
     struct file *f;
{
  if (f->access_mode & FREAD) {
    pseterr(EINVAL);
    return SYSERR;
  }
  if (minor(devno)) {
    pseterr(ENXIO);
    return SYSERR;
  }
  return OK;
}

int ptrclose(s, f)
     struct ptrstatics *s;
     struct file *f;
{
  swait(&s->qsem, SEM_SIGIGNORE);
  if (s->expecting) {
    s->closing = 1;
    ssignal(&s->qsem);
    swait(&s->close_sem, SEM_SIGIGNORE);
    s->closing = 0;
  } else {
    ssignal(&s->qsem);
  }
  s->lines = s->chars = 0;
  return OK;
}
int ptrselect()
{
  return OK;
}

/*  assumes:
 **    data in the queue
 **    queue access disabled
 */

void send(s)
     struct ptrstatics *s;
{
  int prio;
  char c;

  if (s->nextnl) {
    c = ’\r’;
    prio = s->nextnl;
    s->nextnl = 0;
    s->lines++;
  } else {
    c = s->q[s->head].c;
    prio = s->q[s->head++].pri;
    s->head %= s->qlen;
    s->qdata--;
    ssignal(&s->free_sem);
    s->nextnl = c == ’\n’? prio : 0;
    s->chars++;
  }

  port_out(c, s->datap);
  port_out(s->control | PP_STROBE, s->controlp);
  port_out(s->control | PP_IENABLE, s->controlp);

  if (!s->nextnl) {
    swait(&s->prio_sem, SEM_SIGIGNORE);
    priot_remove(&s->pt, prio);
    if (prio == s->curpri) {
      prio = priot_max(&s->pt);
      if (prio != s->curpri) {
   s->curpri = prio;
   stsetprio(s->stid, (prio<<1)+1);
      }
    }
    ssignal(&s->prio_sem);
  }
}
```

```
void ptrint(s)
     struct ptrstatics *s;
{
  ssignal(&s->int_sem);
}


void ptrthread(s)
     struct ptrstatics *s;
{
  for (;;) {
    swait(&s->int_sem, SEM_SIGIGNORE);
    swait(&s->qsem, SEM_SIGIGNORE);
    if (s->qdata || s->nextnl) {
      send(s);
    } else {
      s->expecting = 0;
      /* disable ptr interrupts: */
      port_out(s->control, s->controlp);
      if (s->closing) ssignal(&s->close_sem);
    }
    ssignal(&s->qsem);
  }
}

/* This function ptrthread() is really the only difference 
between this
   driver and the others. It is a kernel thread used by the 
driver to send
   characters out to the printer. */

int ptrwrite(s, f, buff, count)
     struct ptrstatics *s;
     struct file *f;
     char *buff;
     int count;
{
  int i = count, myprio;
  myprio = _getpriority();
  swait(&s->prio_sem, SEM_SIGIGNORE);
  priot_addn(&s->pt, myprio, count);
  if (myprio > s->curpri) {
    s->curpri = myprio;
    stsetprio(s->stid, (myprio<<1)+1);
  }
  ssignal(&s->prio_sem);

  while (i--) {
    while (swait(&s->free_sem, SEM_SIGABORT)) {
      swait(&s->prio_sem, SEM_SIGABORT);
      priot_removen(&s->pt, myprio, i+1);
      if (s->curpri == myprio) {
   s->curpri = priot_max(&s->pt);
   if (s->curpri < myprio) {
     stsetprio(s->stid,(s->curpri<<1)+1);
   }
      }
      ssignal(&s->prio_sem);
      deliversigs();

      priot_addn(&s->pt, myprio, i+1);
      if (myprio > s->curpri) {
   s->curpri = myprio;
   stsetprio(s->stid, (myprio<<1)+1);
      }
      ssignal(&s->prio_sem);
    }

    swait(&s->qsem, SEM_SIGIGNORE);

    s->q[s->tail].c = *buff++;
    s->q[s->tail++].pri = myprio;
    s->tail %= s->qlen;
    s->qdata++;
    if (!s->expecting) {
      send(s);
      s->expecting = 1;
    }
    ssignal(&s->qsem);
  }
  return count;
}

int ptrioctl(s, f, command, arg)
     struct ptrstatics *s;
     struct file *f;
     int command;
     char *arg;
{
  switch (command) {
```

```
  case PTRSTATUS:
    if (wbounds((int)arg) < sizeof(struct ptrstatus)) {
      pseterr(EFAULT);
      return SYSERR;
    }
    ((struct ptrstatus *)arg)->chars = s->chars;
    ((struct ptrstatus *)arg)->lines = s->lines;
    break;

  case FIOPRIO:

  case FIOASYNC:

    break;

  default:

    pseterr(EINVAL);
    return SYSERR;
  }

  return OK;
}

#include <dldd.h>

static struct dldd entry_points = {
  ptropen, ptrclose, 0, ptrwrite,
  ptrselect, ptrioctl,
  ptrinstall, ptruninstall, (char *) 0
  };
```

Index

Symbols

#include directive 9
/dev directory 18

A

Accessing Hardware 49–60
address
  converting virtual to physical 43
  device 39
  flash memory base in kernel virtual space 202
  kernel direct mapped 39
  kernel virtual 38
  physical 39
  user virtual 38
  validating 46
address space management 157, 170
address translation 43
  creating and removing 164
  UNIX drivers 289
address types, LynxOS 38
aliasing user virtual address 48
application
  access to devices and drivers 18
  example 182
  managing Hot Swap 175
  signal handler 70
  using DRM facilities 172

B

base address, flash memory 197
block device 3
  raw data transfer capability 3
  uninstalling dynamic 152
  UNIX driver 291
block device driver
  buffers 3
  characteristics 3
  dynamic installation 150
  entry point functions 22
  file system support 3
  install() 23
  ioctl() 29
  ioctl() example 30
  kernel buffering 3
  strategy() 32
  UNIX I/O support routines 293
blocking 64
  with interrupts disabled 74
bridge device, inserting 172
BSP, relationship to DRM 159
buf_entry data structure 33
buffer, block device driver 3
buffers 12
bus architecture 160
bus error handler 4, 12
bus errors 154
bus layer
  of a bus node 161
  of a bus/device node 161
  relationship to DRM 159

C

callback routine, writing 201
  flash virtual base 202
  MTD specific data 203
  operation code 201
  operation parameter 202
  return code 203
  synchronization 203
Card Services 205
  access to PC cards 206
  advanced client services 208
  bulk memory services 208
  calling conventions 209
  client callback 211
  client device drivers 206, 247
  client services 207
  client utilities 207
  detecting presence of 210
  event notification 210
    event code 211
    event logical socket 211
    events supported 212
  groups 206
  header files 209
  Initialization 206
  interface 209
  logical sockets 206
  Overview 206
  reference of supported services 217
    AccessConfigReg 218
    DeregisterClient 219
    ErrorName 219
    GetCardServiceInfo 220
    GetConfigurationInfo 221
    GetFirstTuple 224
    GetNextTuple 226
    ParseTuple 227
      tuple codes 227
    RegisterClient 237
    ReleaseConfiguration 238
    ReleaseIO 239
    ReleaseIRQ 240
    RequestConfiguration 240
    RequestIO 242
    RequestIRQ 243
    ServiceName 244
  resource management 207
  special services 208
CardServices() function 209
  client device drivers 210
    adding to ISA driver 216
    deregistering 215
    identifying PC card 212
    PC card configuring 213
    registration 210
    sample PC card enabler 215
  parameters 209
  synopsis 209
changing priority of an I/O task 29
character device 3
  uninstalling dynamic 152
character device driver
  characteristics 3
  dynamic installation 149
  entry point functions 22
  install() 23
  ioctl() example 30
  MTD 190
  read() 27
  write() 28
clock interrupt handler 12, 80, 85, 98
close() 21, 26, 80
  example 27
code, blocking 64
code, example device driver 307
code, nesting 65
code, partitioning 64
code, protecting 64, 68
combining synchronization mechanisms 73
components, device driver, summary 14
concurrent operating system 63
CONFIG.TBL 144
configuration file, network driver 137
consumer tasks 71
contacting LynuxWorks xv
Contents iii
context switch time 102
context switch time, kernel threads 103
controlling interrupts 116
copying data from a buffer 28
copying data from a device 27
Copyright Information ii
counting semaphore 67, 72
  to manage free list 75
cprintf() 153
critical code regions, UNIX driver 298

D

data availability indication 79
data packet
  freeing mbuf 125
  input 132
    copying data to mbufs 133
    determining packet type 133
    enqueueing packet 134
    processing 132
    statistics counters 134
  output 130
    ether_output function 130
    kernel thread processing 132
  passing 122
  queues 127
    input 127
    output 127
data structures
  device information 4
  dldd 4
  entry_points 22
  for user task priority tracking 114
  functionality 8
  statics 4
data transfer, limited size 115
data, holding from block device 292
deadlock 64, 66, 75
debug terminal 153
Debugging 152
  bus errors 154
  device driver service routines 153
  hanging the system 153
  overhead 153
  SKDB 154
definition
  device driver 1
  device installation 3
  interrupt handlers 10
  interrupts 10, 79
  kernel semaphore 67
  kernel threads 11, 80
  synchronization 63
  timeout 12
device
  access mode 92
  flash, linear 189
  Insertion/Removal 171
  installation 3
  interrupt management with DRM 169
  interrupt signal 83
  probing for 155, 165
  readiness indication 79
  resource ID 170
  resource management with DRM 170
  setting parameters 28
  state 28
  testing for presence 155
device address 39
Device Address Space Management 157, 170
device and driver configuration file, creating 143
Device Defaults
  x86 95
device driver
  adding to ISA driver 216
  Card Services 206
  components 4
  components, summary 14
  data structures 4
  definition 1
  development and installation 13
  Example, LynxOS 307
  Example, PCI 177
  exclusive access 105
  hardware access 49
  installation 141
  installation attributes 4
  interrupt-based 88
    close() 92
    declaration for ioctl 89
    device information declaration 88
    device information definition 88
    install() 90
    interrupt handler 93
    ioctl() entry point 94
    open() 92
    send() 92
    source code 89
    statics structure 90
    uninstall() 91
    write() 94
  multiple access 105, 107
  network 119
  PC card client driver 205
  PC card enabler 215
  shared resources 4
  Socket Services 247
  static versus dynamic installation 141
  supporting multiple major devices 15

types 3
UNIX initialization routine 294
using DRM facilities 168
with interrupt handler code 86
Device Driver Basics 1–19
device I/O to drivers 157, 170
device ID 14, 17
Device Identification 157, 168
device information block 15
device information data structure 4, 23
deallocation 9
example 9
function 8
instantiating 5
parameters included 8
device information declaration 88, 143, 150
device information definition 88, 143, 150
Device Interrupt Management 157, 169
device registers, reading writing 51
Device Resource Manager 55, 56
Device Resource Manager (DRM) 157–188
device special files 18
device tree 158
adding nodes 171
building 165
DRM bus nodes 158
DRM device nodes 158
navigating 168, 171
devices command 19
devices Command Output 18
devinfo.h header file 143
direct mapped kernel virtual address 48
direct memory access 38
disabling interrupts 61, 62
disabling preemption 61, 62, 65
dldd data structure 4
format 146
functionality 10
dldd.h include file 145
DMA buffers, obtaining 40, 41
DMA controller 38, 39, 43, 79
DMA transfer 38, 79
paging 38, 41
UNIX drivers 290
documents, LynxOS xiii
documents, online xiii
drinstall() system call 150
driver and device configuration tables 143
driver code
attaching to end of kernel image 142
installation 143, 145
linking with kernel 141
driver ID 14, 17
driver response time 102
drivers command 19
drivers command output 17
DRM 55, 56
bus layer relationship 160
components 159
concepts 158
device tree 158, 171
Hot Swap management facilities 176
identifying device handle 165
Initialization 164
Interface Specification 166
module view 159
Node States 162
nodes 160
bus 158
device 158
fields 161
life cycle state 161, 163
read and write to bus layer-defined resources 170
registering interrupt handlers 81
resource allocation 165
service routines 160, 164
device insertion/removal 171
summary 164
services 157
device address space management 170
device I/O to drivers 170
device identification 168
Device Interrupt Management 169
using from applications 172
sysctl() interface 172
using from device drivers 168
dynamic installation 141, 142
advantages 142
block driver 150
character driver 149
device and driver uninstallation 151
dr_install() 22
drinstall() 22
drivers 10
entry point functions 22
error messages 152
import file 148
procedure 145

All Platforms 149
device information definition and declaration 150
device installation 151
driver source code 145
node creation 151
PowerPC 148
x86 147
uninstall() 24

E

eieio() 55
Entry Point Functions 21–??
entry point functions 4, 6
addresses 22
close() 6
install() 6
ioctl() 6
naming 7
open() 6
read() 6
registering names 22
select() 6
sharing information 287
strategy() 6
uninstall() 6
usage guidelines 22
watchdog 136
write() 6
entry points, network driver 128
error conditions, handling 70
error messages, dynamic installation 152
ether_output function 130
Ethernet address, network driver 129
Ethernet header 120, 129, 133
event synchronization 67, 69
event synchronization semaphore 70, 73
Example Application 182
Example device driver 13
driver code 307
header files 307
PCI device 177
Example PC card enabler 215
Example, interrupt-based printer device driver 88
exception handling 13
Exclusive access 105
interrupt handler 106
kernel thread 106
kernel thread priority 113
priority tracking 112
top-half entry point 105

F

file.h header file 26, 27, 28, 29, 30, 35
fixed priority scheduler 69
flash cache 197
flash device, linear
access to partition 199
base address 197
mapping to contiguous kernel virtual space 197
MTD callback routine 190, 200
partition information 199
partitioning 200
sector size 195
size 195
two layer model for access 189
Memory Technology Driver 189
user interface 189
Flash ID, registration data 194
flash information block 195
flash_mgr module 189
deregistering with 201
flash partition data 199
registering with 190
free list
accessing from interrupt handler 74, 76
managing with counting semaphore 75
manipulating 73
functionality, data structures 8
functions, mbuf 126

G

GNU C compiler, installing driver with 147

H

handling, signals 69, 76
hardware access 49

platform-specific 49
PowerPC systems 55
ISA bus 55
PCI support facility 55
x86 systems 51
reading writing device registers 51
hardware exception conditions 79
hardware interrupt, acknowledging 101, 106
header files
Card Services 209
devinfo.h 143
example device driver 307
file.h 26, 27, 28, 29, 30, 35
kernel data structure 119
kernel.h 51, 67
mem.h 45
port_ops_x86.h 51
Socket Services 259
st.h 114
Hot Swap 158, 169, 249
PC Card Enabler 249
pcmcia_d daemon 249
Hot Swap management applications 175

I

I/O address 8, 23
I/O polling 30
I/O port address 39
I/O system calls 2
ifconfig utility 128, 134
ifnet, network driver 129
import file, dynamic installation 148
install() 21, 23, 40, 41, 80
block device driver 23
character device driver 23
example 23
functionality 23
network driver 128
finding interface name 128
initializing Ethernet address 129
initializing ifnet structure 129
Installation and Debugging 141–155
installation attributes, device driver 4
installation, UNIX driver 305
installing, driver source code 143, 145
instantiating device information structure 5
Intel 8259 interrupt controller 84
interrupt 10
and real-time response 80
controlling 116
definition 79
disabling 61, 62
dispatch time 102
handlers 10
latency 101
levels 83
management 157
managing with DRM 169
pending 80
priority 103
processing in interrupt handler v/s kernel thread 117
response time 102
restoring 62
sharing 82
unbounded disable times 71
using kernel threads 11
vector
interrupt handler stack 82
sharing 81
values, PowerPC 83
values, x86 82
Interrupt and Timeout Handling 79–99
interrupt controller 83
master and slave 84
programming 84
interrupt dispatcher 81
interrupt handler 10
accessing free list 74, 76
accessing user space 47
asynchronous execution 47
clearing 80
declaring and resetting 80
implementing 86
location in device driver 86
processing interrupts within 116
programming considerations 88
registering 23
registering with DRM 81
specifying 80
UNIX driver 296
interrupt handlers, LynxOS 80
interrupt service routine 4, 79
registering 169
setting and clearing 164
interrupt state, saving on a task basis 75
interrupt vector 79

interrupt vector number 81
interrupt vector space 83
interrupt vector table 80
interrupt vector, clearing 81
invoking a device driver 2
ioctl() 21
  block device driver 29
  example, block device driver 30
  example, character device driver 30
  network driver 134
    SIOCSIFADDR 134
    SIOCSIFFLAGS 135
  validating addresses to 46
ioctl() system call 28
iointclr() 80, 81
iointset() 80, 81
IP multicasting support 138
  ether_multi structure 138
IRQ
  interrupt priority 85
  level 8, 23
  sharing 82
IRQ device default configuration 95
ISA bus access, x86 55
ISA device physical address 55
ISA interrupts 83
ISR 79
  uses 79

K

kernel address space 39
kernel buffering 3
kernel data structure 119
  header files included 119
  host Internet address 121
  socket addresses 120
  symbolic constants 119
kernel direct mapped address 39
kernel rebuilding 145
kernel semaphore 61, 67
  definition 67
  functions 68
  incrementing 62
  memory allocation 67
  signaling 63, 67
  state 67
  using as counting semaphore 76
  using for mutual exclusion 68
  value 67
  wait queue 67
kernel support functions 13
  exception handling 13
  memory management 13
  synchronization 13
kernel thread 4, 11, 103
  accessing user space 47
  context switch time 103
  creating 104
  data packet output 132
    starting transmission 132
    statistics counters 132
  definition 80
  execution context 47
  network device 136
  priority 103
  processing incoming packets 132
  processing interrupts within 117
  scheduling priority 80, 104
  signaling 106
  stack size 104
  static thread priority 109
  structure 105
  switching to 47
  TCP/IP module 119
Kernel Threads and Priority Tracking 101–118
kernel virtual address 38
kernel.h header file 51, 67
keyboard 85
kkprintf() 153

L

Linux device drivers
  GPL issues 267
Linux device drivers, porting to LynxOS
  Accessing a device 269
  blocking & non-blocking I/O 274
  bottom halves and kernel threads 274
  driver setup differences 280
  Entry points 269
  Error Handling 272
  Installation 268
  Interrupts 272
  kernel support 275
  memory address translation 276

Preemption 271
scheduling differences 278
Signal Handling 272
System call processing 270
Linux Drivers
Porting 267
LynuxWorks, contacting xv
LynuxWorks, Inc. ii
LynxOS device driver components 4, 5
LynxOS device driver interface model 2
LynxOS driver
major number 288
priority tracking 34
LynxOS entry point functions summary 6
LynxOS features not used in UNIX drivers 285
LynxOS File System and strategy() Interface 33

M

macros, mbuf 125
major and minor device declarations 144
major and minor devices 16
major device
device information block 15
single controller card 15
major number 15
LynxOS drivers 288
major() 26
malloc() 40
mapping
device names to driver names 19
LynxOS virtual addresses to hardware 49
to entry point functions 7
user space address to virtual addresses 48
virtual to physical memory 44
mem.h header file 45
mem_unlock() 42
memlock() 41
memory
allocating 39, 40, 41
allocating for kernel semaphore 67
allocation functions 39
kernel access 37
locking 41
management 13
user accessible space 37
memory locking 41
mem_lock() 41
mem_unlock() 41, 42
Memory Management 37–60
Memory Management Unit 37
memory mapping, virtual to physical 38
memory operations, Card Services 208
memory-mapped device 50
minor device
examples of usage 16
file descriptor 26
initialization 25
single channel or sub-device 15
synchronization between 26
minor number 15
Flash ID 194
flash partition number 199
minor() 26
mknod utility 19
mmap() 21, 35
MMU 37
bypassing 39
MTD 189
callback routine 190, 200
flash virtual base 202
MTD specific data 203
operation code 201
operation parameter 202
return code 203
synchronization 203
writing 201
character device interface routines 190
control register windows 198
deregistering with flash_mgr 201
Device Control Registers 198
Device Info String 195
flash base address 197
Flash ID 194
flash memory size 195
interface overview 190
partition information 199
registering with flash_mgr 190
registration attributes 196
registration data 191
write, read, erase flash device 196
multicast address ranges 138
Multiple access 105, 107
non-atomic requests 115
priority tracking 114
kernel thread 115
top-half entry point 114
with kernel threads

interrupt handler 108
kernel thread 108
top-half entry point 108
without kernel threads 107
multiplexing 30
mutex mechanisms 75
mutual exclusion 13, 68
disable()/restore() mechanism 74

N

naming entry point functions 7
network device
kernel thread 136
local RAM 133
resetting hardware 136
Network Device Drivers 119–140
network driver 119
communicating with TCP/IP module 121
device configuration file 129, 137
entry points 128
hardware independent interface 121
hardware initialization 128
ioctl() 134
SIOCSIFADDR 134
SIOCSIFFLAGS 135
IP multicasting support 138
packet input 132
packet output 130
ether_output 130
kernel thread processing 132
packet queues 127
priority tracking 137
reset entry point 136
statics data structure 127
watchdog entry point 136
network interface
bring up or down 135
checking operation 136
initializing Ethernet address 129
initializing ifnet structure 129
name 128
packet queues 127
setting IP address 134
node creation 151
nodetab file 144, 145
non-atomic requests 115
non-deterministic environment 63

O

online documentation xiii
open() 21, 25, 40, 41, 80
example 26
open() system call 26
OSBASE constant 37

P

packet queues 127
paging 38, 41
PC Card
adding support for new 250
new driver 251
PC Card Enabler 250
client drivers 205
Configuration Register 218
configuring by client device drivers 213
Hot Swap 249
hot-swap capable driver 216
identification by client device drivers 212
IRQ 223
manipulating 205, 247
removing from running system 250
Socket Services
writing 257
supported by Enabler 249
using 248
PC Card Enabler 248
configuration tables 250
static driver installation 249
PC Card package architecture, LynxOS 205
PC Card Support
architecture 246
Card Services 247
features 245
installing 246
Socket Services 247
Troubleshooting 252
utilities 248
PCI address 39
PCI devices, managing 56
PCI support facility, PowerPC 55
PCMCIA Adapter
adding support for new 251
supported 252

PCMCIA/JEIDA PC Card architecture
    specification 205, 246
pcmcia_d daemon, dynamic driver
    installation 250
permap() 50
permissions for LynxOS virtual memory
    access 50
PHYSBASE constant 39
physical address 39
    ISA devices 55
physical device layout 158
physical memory 39
    allocating 40
    allocating contiguous 41
PID, getting for task 42
port_ops_x86.h header file 51
Porting Linux Drivers to LynxOS 267
Porting UNIX Drivers to LynxOS 285–305
PowerPC
    ddld structure 146
    dynamic driver installation 148
    interrupt vector values 83
    ISA Bridge 55
    PCI-to-ISA bridge 55
    subtractive decoding 55
    VME-to-PCI bridge 55
PowerPC hardware access 55
    PCI support facility 55
PowerPC Kernel Access Virtual Memory Map,
    LynxOS 57
PowerPC User Access Virtual Memory Map,
    LynxOS 58
preemption 13
    restoring 62
    UNIX driver code 298
preemption, disabling 61, 62, 65
preemptive operating system 63
pre-mapped region 50
printf() 153
priority
    interrupts v/s tasks 103
    inversion 69
    kernel thread 104, 109
    UNIX driver 298
    user v/s kernel 111
priority inheritance 29, 69
    semaphores 69
priority levels, user applications 111
priority scheduler, fixed 69
priority tracking 11, 29, 109
    driver-level requests 116
    exclusive access driver 112
    multiple access driver 114
    network driver 137
    w_count 34
probing for devices 155
processes
    awakening 62
    number of, waiting on a semaphore 63
producer process 71
protecting critical code sections 64, 68

Q

queues 12
    in device driver code 86
    using 86

R

race conditions 64, 66, 75
rbounds() 154
    values returned 46
read() 21, 27
    example 27
read() system call 27, 33
reading writing device registers, x86 51
real-time clock 85
real-time response 101, 103
    and interrupts 80
    and signal handling 76
rebuilding kernel 145
Reference manuals xiii
requests, driver-level 116
reset Entry Point 136
resource
    allocation 72
    availability 8, 23
    required by device driver 6, 21
resource pool management 72

S

scatter/gather transfers 41
scheduling with static thread priorities 109

select() 21, 30
  example 31
select() system call 30
semaphores 12
  event synchronization 70
  priority inheritance 69
serial device, data transfer 116
service routines, DRM 164
shared resources 12
  accessing 63
shared static data structures and queues, allocating 40
sharing IRQs 82
signal handler, application 70
signal handling 69
signal handling and real-time response 76
signaling, kernel thread 106
signaling, semaphore 67
signals and tasks 67
Simple Kernel Debugger 154
SKDB 154
sleep() 298
Socket Services 205
  access to PC Cards 206, 247
  callback entry point 260
  event notification 260
  Groups 258
    Adapter Services 258
    Socket Services 259
    Window Services 259
  header files 259
  identification number 260
  information structure 252
  invoking 260
  overview 258
  PC Card, writing 257
  reference 261
  registering with Card Services 260
  SS_GetInfo 261
  SS_GetSocket 263
  SS_GetWindow 265
  SS_InquireSocket 262
  SS_InquireWindow 263
  SS_SetSocket 262
  SS_SetWindow 264
  status change reporting 206
  status changes 260
  structure 259
Special Note formats xv
sreset() 63
  caution when using 71
  handling error conditions 70
  synchronizing with calls to ssignal() 71
  using with event synchronization semaphores 70
  variable length transfers 71
st.h header file 114
stack size, kernel thread 104
state, semaphore 67
static installation 141
  advantages 141
  code organization 142
  entry point functions 22
  procedure 142
    CONFIG.TBL 144
    device and driver configuration file 143
    driver source code 143
statics data structure 4, 23, 24, 25, 26, 27, 28, 29, 30, 32
  example 9
  function 9
  network driver 127
statistics counters, data packet input 134
statistics counters, data packet output 132
strategy() 21, 32
  UNIX driver 291
struct arpcom 120
struct ether_header 120
struct ifnet 121
  network interface attach routine 130
struct in_addr 121
struct mbuf 122
  adding or removing data 123
  allocating 124
  clusters 125
  commonly used functions 126
  commonly used macros 125
  copying incoming packets to 133
  freeing 125
  packet header 131
  position and size of data 123
struct sockaddr 120
struct sockaddr_in 120
structure, interrupt-based driver 87
structure, kernel thread 105
symbolic constants, kernel data structure 119
Synchronization 61–77
synchronization 12, 298
  between minor devices 26

by flash_mgr for flash device 203
combining mechanisms 73
manipulating free list 73
definition 63
event 67, 69
mutual exclusion 13
preemption 13
UNIX driver 289, 298
synchronization functions
disable() 62
pi_init() 63
restore() 62
scount() 63
sdisable() 62
signaln() 63
sreset() 63
srestore() 63
ssignal() 62
swait() 62
sysbrk() 40
sysctl() interface
accessing DRM 172
parameters 174
sysfree() 40
system calls, I/O 2
system halts 12
system management service routines 169
system response time 107
SystemV, interrupt capabilities 296

T

task
completion time 102
consumer 71
getting PID 42
indefinite blocking 69
preemption 62
priority 103
response time 102
semaphore signaling 67
signal handler 67
virtual address space 37
TCP/IP module
calling driver start routine 131
communicating with driver 121
kernel threads 119
transmitting packets 131
Technical Support xv
timeout 12
cancelling 97
interrupts 80
interval 97
lengthy processing 98
using for events 70
timeout handler 4, 12, 79, 97
algorithm 97
algorithm with kernel threads 98
calling 98
timeout ID 97
timeout() 97
transfers, direct memory 38
Typographical Conventions xiv

U

U structure 296
u_base, u_count, u_offset 296
u_error 297
u_fmode 297
u_procp 297
u_segflg 297
u_tsav, u_nofault 297
uninstall() 21, 24, 80
example 25
functions 24
uninstallation, dynamic 151
UNIX driver
address translation 289
block I/O support routines 293
buf structure 292
debugging 294
DMA transfers and raw I/O 290
event synchronization 298
handling interrupts 286
initialization routine 294
probing for devices 295
installation method 305
interface with kernel 288
driver service calls 288
interface with user applications 299
accessing user space 303
entry points 299
error return 304
ioctl 302
major and minor numbers 300

open/close 301
read/write 301
select 303
interrupt handling 296, 305
kernel and driver goals 285
LynxOS features not used 285
memory management functions 288
porting strategy 286
porting to LynxOS 285
POSIX model 305
reentrance and synchronization 298
sleep() function 298
source/binary compatibility 285
structure 287
global variables 287
synchronization 289
transfers to block devices 291
U structure 296
user and kernel priorities 111
user space address, mapping to virtual addresses 48
user space, accessing from interrupt handler and kernel threads 47
user virtual address 38
aliasing 48

V

validating addresses 46
variable length transfers 71
vector, interrupt 79
clearing 81
interrupt handler stack 82
sharing 81
values, PowerPC 83
values, x86 82
virtual address conversion 45
Virtual Address Model, LynxOS 37
virtual map
memory-mapped device 50
PowerPC kernel access 57
PowerPC user access 58
remapping during context switches 37
x86 kernel access 52
x86 user access 53
virtual memory
LynxOS access permissions 50
mapping to physical memory 44
pre-mapped region 50
VME address 39
VME interrupts 83
vmstart command 41

W

watchdog entry point 136
disabling 136
wbounds() 154
write() 21, 28
example 28
write() system call 28, 33
Writing Flash Memory Technology Drivers (MTDs) 189–203
Writing PC Card Client Drivers 205–265

X

x86
devices 51
dynamic driver installation 147
interrupt vector values 82
Kernel Access Virtual Memory Map 52
User Access Virtual Memory Map 53
x86 hardware access 51
ISA bus 55
reading writing device registers 51